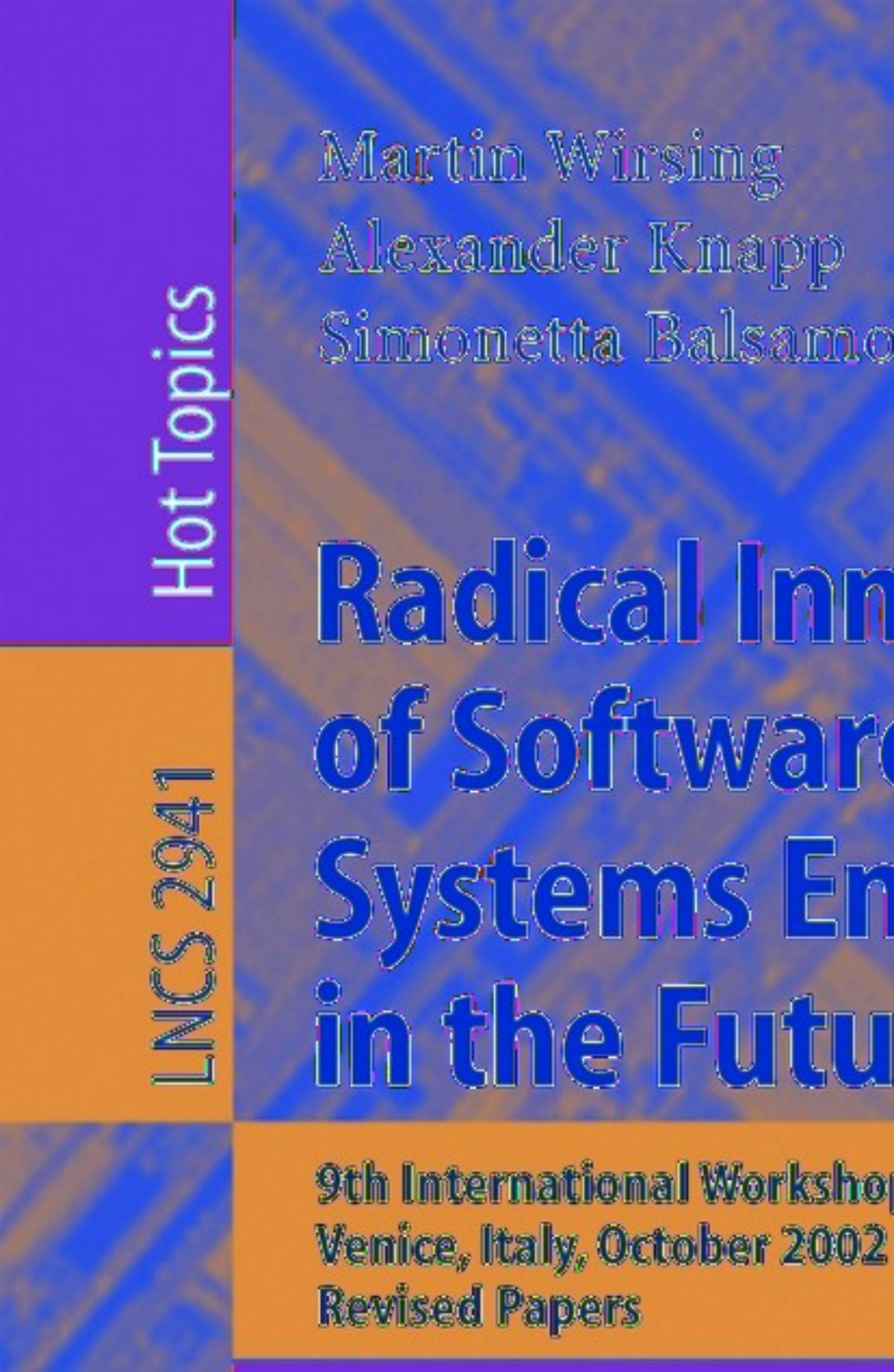

Hot Topics
LNCS 2941
Martin Wirsing
Alexander Knapp
Simonetta Balsamo
Radical Inn
of Softwar
Systems En
in the Futu
9th International Worksho
Venice, Italy, October 2002
Revised Papers

Lecture Notes in Computer Science 2941
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Hong Kong
London
Milan
Paris
Tokyo

Martin Wirsing Alexander Knapp
Simonetta Balsamo (Eds.)

Radical Innovations of Software and Systems Engineering in the Future

9th International Workshop, RISSEF 2002
Venice, Italy, October 7-11, 2002
Revised Papers

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Martin Wirsing
Alexander Knapp
Ludwig-Maximilians-Universität München, Institut für Informatik
Oettingenstraße 67, 80538 München, Germany
E-mail: {wirsing,knapp}@informatik.uni-muenchen.de

Simonetta Balsamo
Università Ca' Foscari di Venezia, Dipartimento di Informatica
Via Torino 155, 30172 Mestre-Venezia, Italy
E-mail: balsamo@dsi.unive.it

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress.

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

CR Subject Classification (1998): D.2, D.3, F.3

ISSN 0302-9743
ISBN 3-540-21179-9 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik
Printed on acid-free paper SPIN: 10991596 06/3142 5 4 3 2 1 0

Preface

This volume contains the papers from the workshop "Radical Innovations of Software and Systems Engineering in the Future." This workshop was the ninth in the series of Monterey Software Engineering workshops for formulating and advancing software engineering models and techniques, with the fundamental theme of increasing the practical impact of formal methods.

During the last decade object orientation was the driving factor for new system solutions in many areas ranging from e-commerce to embedded systems. New modeling languages such as UML and new programming languages such as Java and CASE tools have considerably influenced the system development techniques of today and will remain key techniques for the near future. However, actual practice shows many deficiencies of these new approaches:

- there is no proof and no evidence that software productivity has increased with the new methods;
- UML has no clean scientific foundations, which inhibits the construction of powerful analysis and development tools;
- support for mobile distributed system development is missing;
- for many applications, object-oriented design is not suited to producing clean well-structured code, as many applications show.

As a consequence there is an urgent need for discussing the role of object-orientation and new "post object-oriented" software engineering and programming techniques. The aims of the workshop, continuing the effort to bring together pragmatic and foundational research in software engineering, were threefold:

- to discuss the actual problems and shortcomings in software and systems engineering, to evaluate potential or partial solutions that have been proposed, and to analyze why some ideas were or were not successful;
- to propose and discuss in a proactive way radically new innovations in software and systems engineering and to present visionary and explorative perspectives and bold ideas for the modeling language, the programming language, the system development method, and the system development process of tomorrow;
- to show how the wealth of past foundational research in software engineering can be uplifted to handle the new problems posed, among others, by the different levels of component and system granularity, the heterogeneity of components, the use of distribution and communication, and the request for appropriate human-interface support.

The workshop program consisted of 36 invited talks by distinguished scientists and practitioners in the field. The participants were invited to submit

a written version of their talks for possible inclusion in these proceedings. All submissions underwent a careful refereeing process by the steering committee and the programme committee. This volume contains the final versions of the 24 contributions that were accepted.

Our sincere thanks go to all the referees who helped reviewing the submissions.

We would like to thank David Hislop for his continuous support of the Monterey workshop series and in particular for helping us organize this workshop in Europe. The financial support of the US Army Research Office[1], the US National Science Foundation, the Münchener Universitätsgesellschaft, and the University of Venice is gratefully acknowledged. Marta Simeoni and several other members of the Computer Science Department of the University of Venice provided invaluable help throughout the preparation and organization of the workshop.

Munich, November 2003 M. Wirsing, S. Balsamo, A. Knapp

[1] The views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

Organization

Steering Committee

Egidio Astesiano	University of Genova, Italy
Manfred Broy	Technical University of Munich, Germany
David Hislop	US Army Research Office, USA
Luqi	US Naval Postgraduate School, USA
Zohar Manna	Stanford University, USA

Programme Committee

Mikhail Auguston	New Mexico State University, USA
Barrett Bryant	University of Alabama at Birmingham, USA
Daniel M. Berry	University of Waterloo, Canada
Valdis Berzins	US Naval Postgraduate School, USA
Rance Cleveland	SUNY at Stony Brook, USA
Peter Freeman	Georgia Institute of Technology, USA
Jeannette Wing	Carnegie Mellon University, USA
Purush Iyer	North Carolina State University, USA
Simonetta Balsamo	Università di Venezia, Italy
Swapan Bhattacharya	Jadavpur University, India
Marie-Claude Gaudel	University of Paris-Orsay, France
Carlo Ghezzi	Politecnico di Milano, Italy
Axel van Lamsweerde	Université Catholique de Louvain, Belgium
Oscar Nierstrasz	Bern University, Switzerland
Martin Wirsing	University of Munich, Germany

Additional Referees

Michael Barth	Philipp Meier	Gianna Reggio
Peter Braun	Markus Pister	
Alexander Knapp	Alexander Pretschner	

Sponsoring Institutions

US Army Research Office
US National Science Foundation
Münchener Universitätsgesellschaft

Table of Contents

Architecture Specific Models: Software Design on Abstract Platforms*
(The P2P Case)

Egidio Astesiano, Maura Cerioli, and Gianna Reggio

DISI, Università di Genova, Italy
{reggio,cerioli,astes}@disi.unige.it

Abstract. We address in general the problem of providing a methodological and notational support for the development at the design level of applications based on the use of a middleware. In order to keep the engineering support at the appropriate level of abstraction, we formulate our proposal within the frame of Model Driven Architecture (MDA).
We advocate the introduction of an intermediate abstraction level (between PIM and the PSM), called ASM for Architecture Specific Model, which is particularly suited to abstract away the basic common architectural features of different platforms. In particular, we consider the middlewares supporting a peer-to-peer architecture, because of the growing interest in mobile applications with nomadic users and the presence of many proposals of peer-to-peer middlewares.

Introduction

There are three sources of inspiration and motivation for the work presented in this paper.

Engineering Support for Middleware based Software. The use of a middleware, encapsulating the implementation of low-level details and providing appropriate abstractions for handling them in a transparent way, improves and simplifies the development and maintenance of applications (see, e.g., [3]). But, the direct use of middleware products at the programming level without an appropriate software engineering support for the other development phases is dangerous. Hence, as it is argued in [6], the software engineering support, at the design level at least, has to take middleware explicitly into account, providing middleware-oriented design notations, methods and tools. The main aim of this paper is to show how well-established software engineering design techniques can be tailored to provide support for middleware-oriented design.

Platform Independent versus Platform Specific Modeling. Relying on the features of a particular platform too early during the design phase results in rigid models that cannot be reused for further implementations based on different

* Partially supported by Murst - Programma di Ricerca Scientifica di Rilevante Interesse Nazionale Sahara.

M. Wirsing et al. (Eds.): RISSEF 2002, LNCS 2941, pp. 1–15, 2004.

technologies. The so called MDA (for *Model Driven Architecture*), a technique proposed by the OMG, advocates the initial use of a *Platform Independent Model* (PIM) to be refined into one or more *Platform Specific Models* (PSM) that are its specializations taking into account the features of the particular technology adopted. The PSM will be adapted or even completely replaced accordingly to the frequent changes in the technology. However, in our opinion, the gap between PIM and PSM can be too wide. Indeed, there are clearly architectural choices, like the level and paradigm of distribution, that are a step down toward the implementation, having fixed some of the details, but are still quite far away from a specific platform, because they can be realized by several different middlewares. Therefore, we advocate the introduction of an intermediate level, called ASM for *Architecture Specific Model*. Devising an ASM implies to define a basic abstract paradigm for the chosen architecture providing conceptual support for the features of the different middlewares for that kind of architecture. Since our proposal is made in the context of the MDA and due to the relevance and success of development methodologies based on object-orientation in general and in particular supported by the UML notation, technically the ASM level is presented by a UML-*profile*. While many profiles have been proposed, few are middleware-oriented (the best known of them is the one for CORBA) and they are all supporting the PSM level, by providing a notation for the features of one particular middleware. We do not know of any UML-profile supporting the development of software based on P2P middlewares, and this provides further motivation to our work.

Decentralization and Mobility. The use of mobile devices (mobile computing) supporting applications for nomadic users is becoming popular. That kind of application needs an appropriate underlying architecture. As it is easily understood and explicitly advocated by many researchers (see, e.g., [2]), particular advantages seem to be provided by the so-called peer-to-peer (shortly P2P) architecture, namely a network of autonomous computing entities, called *peers*, with equivalent capabilities and responsibilities. Recently many proposals have been put forward for P2P architectures, by researchers and companies, mainly under the form of P2P middlewares (see, e.g., [11,16,7,9,8,15]). We have been particularly stimulated by the proposal of PeerWare [4], that has been analyzed and used as a paradigmatic example within the project SALADIN (`http://saladin.dm.univaq.it/`). Here, we present what can be considered an *abstraction of most current proposals, that does not pretend to be the definitive choice, also because the P2P paradigm is very recent and our main aim is methodological in advocating the MDA/ASM approach.*

Our notation supports the design of an application by a set of diagrams describing the system architecture, that is the peers and how they are grouped to share resources. Such peers and groups are typed and each such type is described by a diagram. In the case of a peer type, the diagram describes the activity and the resources used and provided by instances of that type, whereas for a group type it just describes the resources shared in such group. A most notable feature of our approach is that the middleware is naturally integrated into the

object-oriented paradigm, by describing it as an object, one for each peer, whose operations correspond to its primitives. The same pattern can be specialized to provide profiles for the PSM level, by specializing the classes representing the middleware objects and the resources and adding/removing features. As a consequence of the introduction of the ASM level, also the mapping of a PIM to a PSM is factorized in two steps, from PIM to ASM and from ASM to PIM.

In this paper, in Sect. 1, after presenting the basic ideas of MDA, we introduce a running example to illustrate the basic concepts and techniques. This is first done outlining the related PIM. In Sect. 2 we introduce our abstract P2P architecture paradigm showing informally how the example application can be mapped onto a P2P architecture. Then, in Sect. 3 we illustrate our profile by showing its structure and its use in defining an ASM for the example application. Finally, in the last section we draw some conclusions, discuss the relationships to existent work, and give some hints on future directions of research.

1 Introducing and Illustrating MDA and PIM

1.1 Model Driven Architecture (MDA)

The Model Driven Architecture (MDA) proposed by OMG, see [10,14], defines an approach to system specification that separates the specification of system functionality from the specification of its implementation on a specific technology platform. Any MDA development project starts with the creation of a *Platform Independent Model* (*PIM*), expressed in UML. The PIM expresses only business functionality and behavior undistorted, as much as possible, by technology. Because of its abstraction, it retains its full value over the years, requiring change only when business conditions mandate. Then, one or more *Platform-Specific Models* (*PSM*) are given, implementing the PIM. Ideally, the PSM are derived from the PIM by a mapping that converts the run-time characteristics and configuration information that we designed in the application model in a general way into the specific forms required by the target platform. Guided by standard mappings, automated tools may perform as much of these conversions as possible. To simplify the definition of the mapping, the PSM should be expressed in UML, possibly enriched by linguistic tools to represent the platform ingredients. This is usually done by a UML profile (extension/variant of the basic notation).

1.2 A Worked PIM Example

In this paper we will use as running example a P2P development of an application for handling the orders of a manufacture company: ORDS. Such company stores the products in a number of warehouses, each one serving some locations denoted by their zips, handles automatically the orders, and, to send the invoices to the clients, uses special mail centers. The orders are collected by salesmen, who may also verify the status of old orders. The case study we present here is a simplified toy version, with a minimal amount of features to illustrate our points.

Our abstract design of the ORDS system (modeled by a PIM, named ORDS-PIM), presented in this section, has been made following a simple method developed by our group. Such method assumes that the designed system is built, besides by ≪datatype≫ representing pure values, by objects of three kinds of classes and provides stereotypes to denote them:

– ≪boundary≫ taking care of the system interaction with the external world,
– ≪executor≫ performing the core activities of the system,
– ≪entity≫ storing persistent data (e.g., a database).

The first two stereotypes specialize active classes, while ≪entity≫ specializes passive classes.

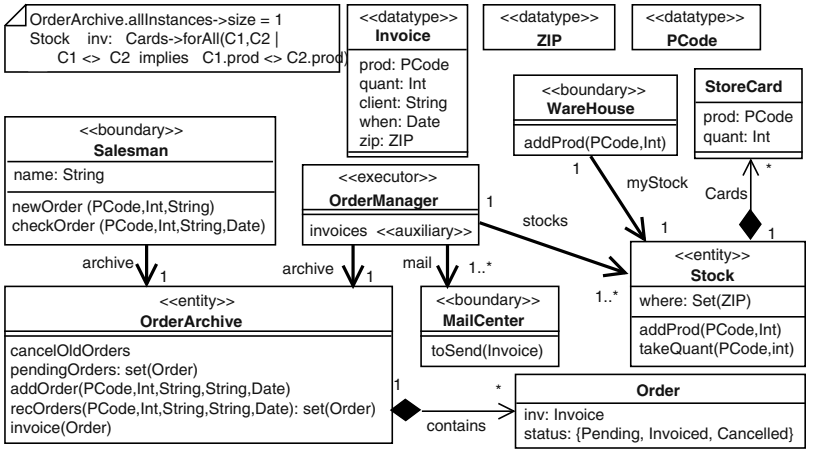


Fig. 1. ORDS-PIM Static View

The static view of the ORDS-PIM, in Fig. 1, shows that the designed system has some interfaces (≪boundary≫ classes) for interacting with the entities of its context: the salesmen, the warehouse workers, which refill the stocked products, and the mail centers, which receive the data to prepare the paper mails. The ≪executor≫ class OrderManager models how the orders, collected by the salesmen, are automatically processed. The persistent data used by the system, as the order archive and the stocks stored in the warehouses, are modeled by the ≪entity≫ classes. The invariants on these classes, reported in the same picture, require that there exists a unique OrderArchive, and each stock contains at most one card for each product.

The behaviours of the active classes (≪boundary≫ and ≪executor≫) purely consist in reacting to time deadlines or to signals received from outside; they can be given by trivial statecharts that can be found in [13], together with the definition of the operations.

The method requires all the classes to be fully encapsulated, that is, their instances may be accessed only by calling their operations. Moreover, all the dependencies among classes due to an operation call, have to be made explicit

by an ≪access≫ association, whose stereotype is offered by the method as well. Therefore, the ≪access≫ associations (visually presented by thick lines) fully depict the interactions among the system constituents.

2 A P2P Paradigm

At the moment no standard P2P paradigm seems to be established, not even in the practice. Not only several P2P-oriented middlewares have been proposed (see, e.g., [4,11,7,9,8], the Jxta Initiative (`http://www.jxta.org/`)), or the Bayou System (`http://www2.parc.com/csl/projects/bayou/`), based on different approaches, but several distributed applications have been developed directly implementing the P2P infrastructure for each of them (see [13] for a short list of some such systems).

In the spirit of our proposal for the introduction of an Architecture Specific level within the MDA approach, we present an abstract P2P paradigm for (mobile) distributed systems, trying to abstract as much as possible from those aspects that have different instantiations in the various proposals. Thus, a model based on the paradigm presented here is flexible enough to be implemented using different P2P middlewares and hence is not platform specific. For a more detailed discussion on the features of the proposed P2P paradigm and its relationship with existing P2P middlewares see [13].

2.1 P2P Paradigm Basic Concepts

For us a *P2P system* is a network of autonomous computing entities, called *peers*, with equivalent capabilities and responsibilities, distributed each on a different host. Each peer may initiate a communication/cooperation and contributes to the activity of the system by offering resources, accessing the resources of the other peers, and performing private activities. *Resource*, here, is used in a broad sense and includes, for instance, (persistent) data, services, ports and sockets. Consequently "accessing a resource" may include making queries and updating data, calling a service, sending a message on a port, or publishing an event.

In our paradigm, we assume that resources may be accessed both in a nominated way, by their (physical or logical) address, or in a property oriented fashion as the result of a query operation for all the resources satisfying a given property. Moreover, to prevent unauthorized access to some resources we assume that the peers are parted in possibly overlapping *groups*, that create a secure domain for exchanging data, disregarding the actual realization of such boundaries. Then the resources of a peer are divided among the groups it belongs to, and the resources relative to a group will be accessible only to the members of that group. In other words, the local space of each peer is partitioned into a *private* and, for each group the peer is member of, a *public* part, that may be accessed by all and only the members of that group. The private part includes all the activity of the peer and is, hence, the only part where the actual access of the (public and external) resources takes place. The public parts offer the resources needed for the group collaboration.

Since we want to address the mobile case, peers are not required to be permanently available to the groups of which they are members. Thus, while the *membership* of peers to groups is statically decided, their *connection to* a group dynamically changes. Each peer may decide to explicitly join or leave a group changing, as a side effect, the available resources of that group by adding or removing those resident on the peer itself. The capability of (joining) leaving a group is more flexible than simply (connecting to) disconnecting from a network, because it allows to selectively (share) hide the part of the peer resources public on that group. As the presence of peers on a group may dynamically change[1], at any given time a peer connected to some group may access only those resources offered by the peers that are currently connected to such group.

So far, we have discussed the P2P paradigm abstract features. Now, we add a somewhat orthogonal assumption, that the P2P paradigm be supported by some *middleware*, offering a common framework where coordination and cooperation of peers is supported, and the changing network topology is hidden. Then a peer will be a host where, besides the private and the public parts, a software component realizing the middleware services (*middleware component*) resides on. Such middleware component has the absolute control of the resources, that can be accessed only through the middleware primitives. There is just one basic access primitive for this level of middleware, from which other operations can be derived, by specializing some of its parameters. A call of such primitive, named perform, corresponds to selecting a community of resources and performing an action on each of them. Such community of resources is the union of the public communities for some group on a set of peers. Both the group and the set of peers are parameters of the call, as well as the action to be performed.

The middleware is also the unique responsible for group management, in the sense that joining and leaving groups, finding the (currently connected) members of a group are services of the middleware, realized by the "middleware component" resident on the peer and offered to the (private part of the) peer.

2.2 Mapping ORDS into the P2P Paradigm Architecture

In this section we informally present how to translate the ORDS-PIM, defined in Sect. 1.2, into a distributed P2P system following the paradigm of Sect. 2.1. Then, that system will be designed using the UML profile of Sect. 3, getting the P2P oriented ASM for the ORDS case.

To start, we need a rough description of the distributed structure of the system, that is which computers will be available, their users, and their connections to the network. In this case, we will also introduce some mobility aspects, to show how they are handled in the proposed approach. Let us assume that

[1] Following the approach in [1,9,5], we assume that the middleware masks unannounced disconnections, by mechanisms like caching and some reconciliation policy. That is, we regard the groups as realized on an everywhere available connection net, and trust the middleware to solve the problems due to this idealization.

– The company is structured in different branches.
– Each salesman, branch, warehouse and mail center owns a computer.
– The computers of the salesmen are portable and can be connected to Internet by means of a modem, while all the others are assumed to be workstations stably connected by Internet.

The first step to derive a P2P model from the PIM is to devise the peers composing the system. In this case, obviously, we have a peer for each salesman, each warehouse, each branch, and each mail center.

The next step requires to deploy the ORDS-PIM active objects on such peers. In this case the deployment of the ≪boundary≫ objects (Salesman, MailCenter and WareHouse) is quite obvious, because their external users own a peer each. The OrderManager objects could be in principle be deployed on any host, as they interact equally with all the boundary elements. We decide to deploy an OrderManager on each branch peer, to exploit their computational capabilities.

The groups allow to discipline the cooperation among the peers, which should cooperate only if some active objects deployed on them were already cooperating within the ORDS-PIM, that is if there is an ≪access≫ association, between their classes (see Fig. 1). We have *direct cooperations*, when active objects access each other, like for instance OrderManager accessing MailCenter, or *indirect cooperations* when different active objects access the same passive object, like for instance Salesman and OrderManager sharing OrderArchive. Thus, in a first approximation, we can devise three groups for supporting the cooperations in Fig. 1: Mail (among branches and mail centers), Product (among branches and warehouses), and Company (among branches and salesmen).

The objects of ≪entity≫ classes that are accessed only by objects already deployed on a unique peer, will be deployed on the same, while those that are accessed by objects deployed on different peers need to be shared on some group. In principle, they could be deployed on any peer, but in order to minimize communications and to maximize availability, it may be more convenient to deploy them on one of the peers involved in the sharing. Thus in our example, the unique OrderArchive may be deployed either on the branches (together with the order managers) or on the salesman peers. Since the former are more stably connected and more powerful it seems sensible to use the branch peers to store the OrderArchive. Furthermore, we choose, as reasonable in a P2P setting, a distributed realization of the order archive, splitting it in several subarchives, one for each branch, that will contains the orders handled by the manager deployed on such peer. Analogously, the Stock objects could be deployed on the WareHouse or on the Branch peers. But, since each stock entity represents the status of the corresponding warehouse and may be accessed by several branches we deploy Stock objects on WareHouse peers.

As a final step, all the calls of the operations of the objects that are now resources shared on some group, have to be converted into appropriate calls of the middleware services.

3 A Profile for Peer-to-Peer ASM

In this section we will present, as a UML profile (see [12]), a visual object-oriented notation to model the P2P oriented ASMs following the paradigm illustrated in the previous section.

Since we integrate our P2P paradigm within an object-oriented paradigm, it is most natural to consider, as resources to be *shared*, standard *objects*. Thus resources, being objects, hence naturally typed by their classes, are implicitly provided with a precise interface. Moreover, we may use the OO typing mechanism at a higher level to classify the peers and groups constituting the system.

There are two key points in our profile. First, we represent the middleware component on each peer as an object of a class predefined by the profile, offering as operations the middleware primitives. Thus, the access to its primitives is disciplined by the standard mechanism of operation call (see Sec. 3.4 and 3.5).

Second, a model will consists of several UML (sub)models describing, respectively, the system architecture, the kinds of involved peers and groups. The *P2P Static View* describes the architecture of the system at the static level. It presents the types for the peers and groups building the system. The *Architecture Diagram* describes the actual instances of the peer and group types building the system and the memberships among them. The *Resource Community Model* describes the resources of the system shared on the groups of a type. The *Peer Model* describes the private part resident on the peers of a type, and which resources they offer and expect to find on each group they belong to.

That splitting of the modeling into different views corresponds to a separation of concerns during the design phase, providing means for the specification of each part of the system in isolation, as far as possible, making explicit the assumptions on the outside world. Moreover, the consistency among these views provides a useful tool to check that the intuitions about the resources needed for the performance of some group activity meet the description of the same activity from the viewpoint of the involved partners.

3.1 P2P Static View

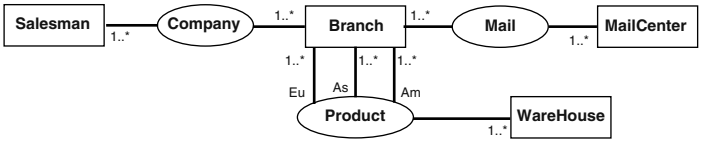
The P2P Static View presents the types of peers and groups used in the P2P system and for each peer type, the possible groups its instances belong to.

In a P2P Static View only classes of the stereotype ≪peer≫ or ≪group≫, that are classes without attributes and operations, and associations of stereotype ≪member≫ may be used.

≪member≫ are oriented associations going from a ≪peer≫ class, represented by a box icon, into a ≪group≫ class, represented by an oval icon, where the multiplicity on the group end is always 1 (and hence it is omitted). In this way the association names allow to uniquely identify the groups a peer of this type belongs to. If there is only one anonymous association from a peer type into a group type, then it is implicitly named as the group type itself.

As discussed in Sect. 2.2, the peers of our P2P realization of ORDS may be classified in four types: Salesman, Branch, MailCenter and WareHouse, while the

groups are of three types: Company, Product and Mail. In order to show a case where several instances of the same group type exist and different associations for the same peer types, let us decide that different groups of type Product represent a continent each. Each warehouse will serve (zip codes within) one continent and is, hence, connected to one Product group, while the branches need to be connected with all the groups of that type, in order to deal with orders for each locations. This classification will restrict the search space for a warehouse capable of handling an order. Such peer and group types are summarized below

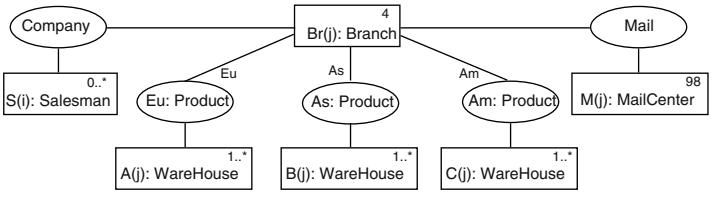


3.2 Architecture Diagram

The Architecture Diagram describes the structure of the modeled P2P system by stating which are the peers and the groups building the system and the memberships among them. It is a collaboration at the instance level satisfying the following constraints.

- The instances, represented as ClassifierRole, must belong to the types presented in the P2P Static View (and are visually depicted using, as previously, the box and the oval icons).
- The links, all belonging to the ≪member≫ associations present in the P2P Static View, are labeled with the corresponding association name.

If the number of the peers and groups composing the system is not determined a priori, or they are too much to be shown on a diagram, we can attach to the object icons multiplicity indicators expressing the number of instances.



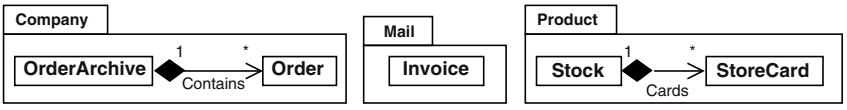
The Architecture Diagram of the ORDS-P2P, above, shows that there is one group of type Company (Mail) connecting all peers of appropriate types, and three groups of type Product, and moreover that each warehouse is member of one of them while each branch is member of each such group. In this case, we have 4 branches, 98 mail centers, and any number of salesmen and warehouses.

3.3 Resource Community Model

A Resource Community Model describes the types of the resources shared by the members of any group of a given type. It simply consists of a UML package

named as the group type itself containing at least a class diagram, where the new special OCL basic types PeerId, GroupId and ResourceId may be used. These types corresponds to peer, group and resource identities to be used as arguments and results of the middleware primitives. They are used as a bridge among the different views composing a UML-P2P model. Since intuitively the resources are manipulated by the peer internal activity through the middleware, no calls of the middleware primitives are allowed within a resource community model.

The models of the resource communities of the ORDS-P2P are presented below, and each of them consists of the shared data needed to realize indirect cooperations as discussed in the previous section. The details of the definition of each class are omitted, because they are as in the ORDS-PIM.



3.4 The Middleware

We introduce the middleware in our profile by defining a UML interface, P2PMW, whose operations correspond to the services that it offers. Notice that the primitives provided by the (abstraction of the) middleware presented here are not intended to match directly any existing middleware. Indeed, we are aiming at a profile for the support of an intermediate level of design where the platform has yet to be decided, and only the architecture paradigm of the system has already been fixed. We may complement the P2PMW interface with a UML class realizing it, say P2PMWclass, which will be then part of the profile definition. The UML description of the operations of P2PMWclass will give an abstract semantic description of the middleware primitives; whereas the attributes of P2PMWclass will give an abstract view of the information on the network managed by the middleware and on the current activities of the middleware itself.

The middleware primitives use the special types PeerId, GroupId and ResourceId, already introduced, and also RemoteAction, defining what can be done on a selected community of resources. RemoteAction is a specialization of UML action (Action is the corresponding meta-class), adding two new actions:

- returnCopy that will make a deep copy of its argument in the private community of the calling peer and return its identity,
- returnRef that will give back the identifier (element of the special type ResourceId) of the argument. Such resource identifier, whenever used in a remote action in the correct resource community will identify the original resource.

A remote action, as any UML action, allows to model both querying and imperative updating over a community of objects. To describe a query or an update on some particular resources in a community, we can use directly the identifiers of such resources, corresponding to a direct reference to the resources in a named style of middleware. But, it is as well possible to find them indirectly, for example by using an OCL expression of the form C.allInstances->select(...) for

selecting all resources of class C satisfying some condition, achieving in this way the anonymous style of resource lookup favoured by some middleware.

A RemoteAction must be statically correct in the context of the communities on which it will operate; in particular no references to the caller environment may appear in it. That constraint on one side allows remote evaluation, and on the other bans interlinks between private and public communities (of different groups) and among the communities local to different peers, as the users cannot exploit (private) local object identities when assigning values to the attributes of (possibly remote) public objects through code execution.

The primitives of P2PMW are:

- mySelf():PeerId returns the identifier of the peer, where the middleware component is resident.
- join(GroupId) connects to the given group.
- leave(GroupId) disconnects from the given group.
- isConnected(GroupId):Boolean checks if there is an active connection to the given group.
- whoIsConnected(GroupId):Set(PeerId)] returns the set of the identifiers of the peers currently connected to the given group.
- members(GroupId,PeerType):Set(PeerId) given g and PT, returns the set of the identifiers of the peers of type PT that are members of g[2].
- perform(GroupId,Set(PeerId),RemoteAction):Set(OclAny) given g, ps, and ra, executes ra in all the public communities for group g of those peers in ps that are currently connected. Then, it collects the results of any returnCopy and returnRef actions, producing a, possibly empty, object collection and yields it as result.

3.5 Peer Model

A Peer Model describes the software required by the modeled P2P system on the peers of a given type. Hence, in a UML-P2P model there will be a Peer Model for each peer type present in the P2P Static View.

A Peer Model for a peer type PT whose instances have the capability of being member of group of the types G1, ..., Gk consists of the following parts:

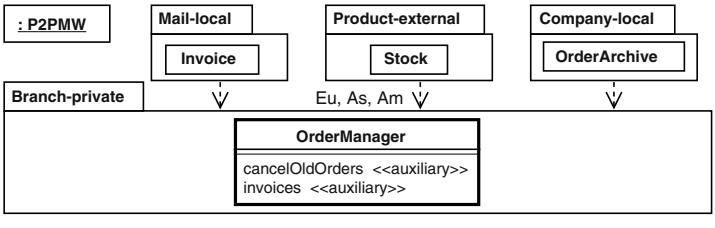
- :P2PMW, a denotation of the used middleware component, that is a *ClassifierRole* for the interface defined in Sect. 3.4, to recall that it is possible to use its operations in the private part.
- for each group type GT in {G1, ..., Gk}:
  - a package GT-public defining the static structure of the public resource community made available to any group of type GT the peers of type PT belong to.
  - a package GT-external defining how the peers of type PT view the external resource communities provided by the other members of any group of type GT they belong to.

[2] Since memberships are statically fixed, the result of this operation is a constant, needed only to give the peers a linguistic means to access the group members.

Each of the two packages for GT defines a resource community by a class diagram that is a subset of the one of the GT Resource Community Model. If any of these two packages is empty (because the peers do no offer resources or do not require resources on such group), then it will be omitted in the diagram.

– a package, named PT-private, that describes the part of the system resident on the peers of that type, parameterized by the references to the groups it is member of (determined by the ≪member≫ associations in the P2P Static View). Such parameters are depicted over the lines connecting the packages corresponding to their types.
A private part is any UML model, where the calls of the middleware object operations may appear, and that implicitly imports all the packages describing the resource communities that the peer may access. There are some obvious static correctness requirements: a peer may only join/leave a group of which it is member; the calls to the primitive perform on some resource community has to be correct w.r.t. the "type" of the community itself. The type of its community and of the communities of different peers on a group of type GT is given by the two packages public and external for GT.

Example: Branch Peer Model. Here we illustrate the use of the Peer Models on the example of the Branch peer type, shown below. The other Peer Models of (a slightly different version of) the ORDS-P2P example can be found in [13], as well as the full details of the Branch Peer Model, that we omit here for brevity.



In Sect. 2.2 we decided to deploy one object of the ≪executor≫ class OrderManager on each branch peer; thus we put the corresponding class in the private part of the Branch peer model. Moreover, we decided to realize the unique database of orders in a distributed way, deploying the orders managed by each branch on its peer. Thus, since the OrderArchive has to be shared with salesman, we will have its class in the Company-public package. Analogously, as the branches need to access the warehouse catalogues, we will have the corresponding class Stock in the Product-external package. Notice that in the Branch Peer Model there is no information about the actual deployment of the Stock objects; it is simply assumed that they are provided by peers on the group Product. Thus, a change in the design of the actual location of those objects would not affect the design of this kind of peers.

The OrderManager has to be slightly modified in order to accommodate the P2P realization and in particular the accesses to the shared data have to be mapped onto calls to the middleware primitives. Thus, for instance the PIM

version of the invoices method has to be modified by using the middleware to access the order archive, the stock and the mail. We leave as comments the PIM version of the shared resource access, to better show the difference.

```
method invoices()
  { os = perform(Company,mySelf,
      returnRef OrderArchive.allInstances.pendingOrders());
    \\ PIM version: os = archive.pendingOrders()
    for O in os do {
      z = O.inv.zip;
      g = z.zip2area();
      ss = perform(g,any,
        if Stock.allInstances ->select(S.where->includes(z))<>{}
                                                    then{returnRef mySelf};);
      \\ PIM version: ss = astocks->select(S.where->includes(O.ZIP));
      done = False;
      while(ss <> {} and not done) do
        { done = perform(g,ss->first,
            Stock.allInstances.takeQuant(O.prod,O.quant););
          \\ PIM version: done = ss->first.takeQuant(O.prod,O.quant);
          if done then { perform(Company,mySelf,OrderArchive.allInstances.invoice(O));}
          else ss = ss - ss.first } }
          \\ PIM version: archive.invoice(O);
          \\ Moreover, in the PIM version it performed mail.toSend(O.inv); while
          \\ here we leave that responsibility to the mail centers; see comment below
```

Notice that, since the warehouses are now classified by their area, we need to know how the ZIP codes are mapped onto the the group identifiers in order to perform a search on the correct group. This is realized by zip2area of the ZIP class with result type GroupId, that is, hence, added to the class. Notice that the OCL operation .allInstances refers to those instances in the community determined by the enclosing call of perform, so for instance the first occurrence in the previous method refers to the public community for the group Company of the peer itself (if it is currently connected, otherwise it is empty).

A further difference in the definition of invoices in the distributed case is that we move the responsibility for calling the toSend operation is moved from the OrderManager to the MailCenter, that will periodically perform a query for invoiced orders, send the corresponding mail and update their status.

3.6 Static Correctness of the Overall Model

Besides the standard UML constraints on the static correctness, we impose some further conditions, following the principle that a strict typing helps the design of systems, by allowing an early, if rough, check of consistency.

As the different parts of a model correspond to different views of the same system, besides the static correctness of each view, already stated before, we have to check for consistency among them. This has already been partially done. For instance, the form of each peer model depends on the P2P Static View and

the classes admissible in each public or external package of some group type GT have to appear in the Resource Community Model for GT. Now, we check the GT-public and GT-external packages for GT from all the Peer Models of the possible members one against the other in order to be sure that if a peer expects some resources from the community, some (other) peer is offering it on the *same* group. Hence, we require that for each group in the Architecture Diagram and each peer member of it, the GT-external package of the type of that peer is included in the union of the GT-public packages of the types of that group members. This condition does not guarantee that all the expected cooperations will really take place, as it is possible that the peers providing some resource are not connected at the same time as the peers needing such resource. But, this kind of failure is due to the dynamic configurations of the groups (to the actual presence of the members) and cannot be checked statically.

4 Conclusions

From the experience of some projects dealing with significant case studies, we have been impressed by the gap existing between the proposed rigorous techniques for software development from scratch and the use of the various kinds of middleware in the practice without any methodological support. In the context of the MDA (*Model Driven Architecture*), we have proposed the introduction of an intermediate level, between PIM and PSM, called ASM (*Architecture Specific Model*) and we have illustrated our proposal in the case of a P2P architecture where distribution and mobility are fully encapsulated by some middleware. The middleware is naturally integrated into the OO paradigm of UML, describing its software components present on each host as objects whose operations correspond to its primitives. The idea of representing the middleware components as objects is general and powerful enough to be reused whenever designing a UML profile to model applications using some middleware.

In order to complete the MDA picture, we need to define in the general case how to transform a PIM into an ASM presented by using UML-P2P. In this paper, we have just sketched how to handle the case of the ORDS application. The mapping will be given by a set of systematic guidelines.

Moreover, to fill the MDA landscape towards the PSM for the special case of P2P architecture, we further need to define the notations to express the PSM, that are UML profiles for specific P2P middlewares, such as PeerWare [4], Jxta [15], Xmiddle [9] et cetera, and guidelines to translate an ASM into the corresponding PIM. We can define such profiles following the way we have defined UML-P2P, just replacing the abstract generic ingredients with the more specific ones supported by that particular middleware.

Acknowledgments

We acknowledge the benefits of many discussions with the colleagues of the Sahara project, and especially the designers of PeerWare, G.P. Cugola and G.P. Picco.

References

1. A. Arora, C. Haywood, and K.S. Pabla. JXTA for J2ME™-Extending the Reach of Wireless With JXTA Technology. Technical report, Sun Microsystems, Inc., 2002. Available at `http://www.jxta.org/project/www/docs/JXTA4J2ME.pdf`.
2. D. Balzarotti, C. Ghezzi, and M. Monga. Supporting configuration management for virtual workgroups in a peer-to-peer setting. In *Proc. SEKE 2002*. ACM Press, 2002.
3. J. Charles. Middleware Moves to the Forefront. *Computer*, 32(5):17–19, 1999.
4. G. Cugola and Gian P. Picco. PeerWare: Core Middleware Support for Peer-to-Peer and Mobile Systems. Manuscript, submitted for publication, 2001.
5. A. Demers, K. Peterson, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The Bayou Architecture: Support for Data Sharing amoung Mobile Users. Technical report, Xerox Parc, Santa Cruz, CA, US, 1994.
6. W. Emmerich. Software Engineering and Middleware: A Roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*, pages 117–129. ACM Press, 2000.
7. Jatelite-System. Jatelite White Paper. Available at `http://www.jatelite.com/pdf/jatelite_en_whitepaper.pdf`, 2002.
8. G. Kortuem, J. Schneider, D. Preuitt, T.G.C. Thompson, and Z. Segall S. Fickas. When Peer-to-Peer comes Face-to-Face: Collaborative Peer-to-Peer Computing in Mobile Ad hoc Networks . In *Proceedings of 1st International Conference on Peer-to-Peer Computing (P2P 2001)*. IEEE Computer Society, 2002.
9. C. Mascolo, L. Capra, S. Zachariadis, and W. Emmerich. XMIDDLE: A Data-Sharing Middleware for Mobile Computing. *Wireless Personal Communications*, 21:77–103, 2002.
10. OMG Architecture Board MDA Drafting Team. Model Driven Architecture (MDA). Available at `http://cgi.omg.org/docs/ormsc/01-07-01.pdf`, 2001.
11. A. Murphy, G. Picco, and G-C. Roman. Developing Mobile Computing Applications with Lime. In M. Jazayeri and A. Wolf, editors, *Proceedings of the 22th International Conference on Software Engineering (ICSE 2000), Limerick (Ireland)*, pages 766–769. ACM Press, 2000.
12. OMG. White paper on the Profile Mechanism – Version 1.0. Available at `http://uml.shl.com/u2wg/default.htm`, 1999.
13. G. Reggio, M. Cerioli, and E. Astesiano. Between PIM and PSM: the P2P Case. Available at `http://www.disi.unige.it/person/ReggioG/`, 2002.
14. J. Siegel and the OMG Staff Strategy Group. Developing in OMG's Model-Driven Architecture (MDA). Available at `ftp://ftp.omg.org/pub/docs/omg/01-12-01.pdf`, 2001.
15. Sun-Mycrosystem. Jxta Initiative. WEB site `http://www.jxta.org/`, 2000.
16. Xerox-Parc. The Bayou Project. WEB site `http://www2.parc.com/csl/projects/bayou/`, 1996.

Tight Structuring for Precise UML-Based Requirement Specifications*

Egidio Astesiano and Gianna Reggio

DISI, Università di Genova, Italy

Abstract. On the basis of some experience in the use of UML, we believe and claim, contrary to a recent wave for allowing almost total freedom as opposed to disciplined methods, that a tighter and more precise structuring of the artifacts for the different phases of the software development process may help speed-up the process, while obviously making easier the consistency checks among the various artifacts. To support our claim we have started to investigate an approach, that, though being compliant with the UML notation and a number of UML-based methods, departs from them both in the basic philosophy, that follows the "tight and precise" imperative, and in the technical solutions for structuring the various artifacts.
Building on some previous work concerning the structure of the requirement specification artifacts, here we complete upwards and improve our proposal, investigating the link between the analysis of the problem domain and the requirement capture and specification. To that purpose we propose a rather new way of structuring the problem domain model and then the link with the system, that encompasses the most popular current approaches to domain modelling. Then we exploit both the domain model and our frame for capturing and specifying the requirements. From our construction we can derive rigorous guidelines for the specification tasks, in a workflow that allows and suggests iteration and incrementality, but in a way that is not just based on the single use cases and takes more care of the overall construction. The various concepts and constructions are illustrated with the help of a small case study.

1 Introduction

In recent years, we have seen the introduction and the acceptance of use-case driven approaches combined with object-oriented techniques, particularly in connection with visual notations such as UML [19]. This is the case of software development process models such as RUP (the Rational Unified Process [14]), Catalysis [6] and COMET [9]. In the last three years we have made some experiments in the use of UML-based and use case-driven techniques and of some related methods, both in teaching and by personal involvement. Those experiments were concerned especially with the early development phases, requirement capture and specification and then design.

* Partially supported by the Italian National Project SAHARA (Architetture Software per infrastrutture di rete ad accesso eterogeneo).

M. Wirsing et al. (Eds.): RISSEF 2002, LNCS 2941, pp. 16–34, 2004.

Because of that experience, we have become increasingly convinced that, to be more effective in terms of both productivity and quality, those approaches need to be improved and complemented especially in two directions. The first is a tighter and more systematic structuring of the artifacts based on precise guidelines for their building. The goal we want to achieve by that is twofold: first, cut experimentally endless discussions on the structural choices and thus making the process much faster; second, provide a better support to the consistency checks among the different artifacts. Indeed, it is well known that consistency is one of the hot problems in multiview modelling approaches [2]. Of course tight and precise structuring is not enough for consistency checks, as long as we want (and we much need) to go beyond pure syntactic checks (see [7] also for references). Thus, there is a second sense of our "precise" qualification that does not refer to the structural aspects, but to the semantics of the single constructs. Indeed, another principle we follow is the use of constructs with an unambiguous well-defined semantics. Altogether, by the tight structuring and the use of semantically well-defined constructs, we here provide an example of what we call well-founded method, as a modern and more viable approach that still embodies the basic sound principles of the "explicitly formal" methods (see [5] for a perspective and a rationale of well-founded methods).

In this paper, we continue the investigation and update the initial proposal first presented in [1]. In that paper, we have outlined some new ideas about the structure of the Requirement Specification artifacts. Here we complete upwards and improve that proposal, investigating the link between the analysis of the problem domain and the requirement capture and specification. Indeed one of our assumptions, backed by our own experience, is the neat separation between the problem domain and the system (as much advocated in the work of some pioneers, such as M.Jackson's [10]). To that purpose, we propose a rather new way of structuring the problem domain model (PDM) and then the link with the system. Our proposal, centered on two views, the Conceptual View and the Work Case View, in a structural sense encompasses the two most popular current approaches to domain modelling, namely conceptual modelling and business modelling, and can be reduced as a specialization to each of those. Then we propose a "system placement" activity, supported by a System Placement Diagram, to relate the system to the domain and, by that, to locate the system boundary.

This paper is mainly aimed at presenting the structural aspects of our approach; we present both its rationale and the technical aspects, illustrated with the help of a small running case study. However, the fact that we illustrate the structure with the end artifacts, should not induce to underestimate the relevance of the methodological aspects in the building of the artifacts. Indeed, in the development we make an ample use of iteration and feedback, as it is unavoidable in any sensible method. But, also for lack of room, we only touch that issue, just providing some methodological guidelines for the workflow, while not presenting the various iterations we have followed when handling the case study.

In the first section, we present the rationale and our way of structuring the problem domain. In the second, we outline the transition from the prob-

lem domain model to the requirement capture and specification, by exploiting our particular way of structuring the requirement artifacts. Then, after some methodological hints on the workflow, we discuss the relation to other and future work. Throughout the paper we illustrate our approach by means of a small case study, shown in Fig. 1.

We have to develop a system AL_L *to handle algebraic lotteries. Our lotteries are said "algebraic" since the tickets are numbered by integer numbers, the winners are determined by means of an* order *over such numbers, and a client buys a ticket by selecting its number. Whenever a client buys a ticket, he gets the right to another free ticket, which will be given at some future time, fully depending on the lottery manager decision. The number of a free ticket is generated by the set of the numbers of the already assigned tickets following some* law.
Thus, a lottery is characterized by an order *over the integers determining the winners and a* law *for generating the numbers of the free tickets. To guarantee the clients of the fairness of the lottery, the order and the law, expressed rigorously with algebraic techniques, are registered by a lawyer before the start of any lottery.*
The system will be then realized as an on-line system, where the tickets must be bought and paid on-line using credit cards with the help of an external service. Possible clients must register with the lottery system to play; and clients access the system in a session-like way. An external service takes care of the authentication of the clients.

Fig. 1. The AL_L case study

2 Modelling the Problem Domain

2.1 Method Rationale

The distinction between the (problem) domain and the (solution) system has been recognized and accepted long time ago in the software engineering community (see, e.g., [10]). The problem domain consists of those aspects of the real world that are relevant for the system to be developed for providing a solution to the problem under consideration. For instance, in the case of a system for handling a lift the relevant domain aspects concern how the lift works, that is in which way the calls can be made, whether the cabin doors are opened/closed by the users, and the most typical habits of the users (e.g., a user immediately leaves the cabin once the doors are open). Of course, the separation line between domain and system depends on the way the problem is stated. For example for the algebraic lottery, the problem domain aspects concern how the clients buy the tickets, how and when the winners are drawn and so on. Instead, in our formulation of that problem, the way the tickets are sold to the clients (e.g., by using Internet and credit cards, by clerks using cash) is a choice to be made when devising the system and thus should appear in the requirements and not in the problem domain.

More or less, any development method requires to model the problem domain either explicitly, by a specific task, or implicitly in the requirement specification

task. We prefer to separate the domain modelling from the requirement definition and to present the result in a specific document (the PDM), because,

- that separation helps get a more abstract unbiased description of the system that we denote by System;
- the resulting PDM may be reused for many different System, thus extending to the early phases of the development the MDA philosophy [12].

Currently, in the literature and also in the practice, there are two main ways to present a PDM:

as a conceptual model: the PDM is a conceptual model of the entities present in the domain, in this case it is usually represented by a (UML) class diagram, where the classes correspond to such entities, the associations to their mutual relationships and, if allowed, the attributes to some characteristics of such entities. Sometime, some limited behavioural aspects are given by sequence/ collaboration diagrams.

as a business model: the PDM is the description of a business, intended as an organized offer of functionalities (business use cases) to outside entities interacting with it (business actors), and with an internal structure (business object model based on business workers and business items). Clearly, in this case actors, workers and items correspond to entities present in the real world, and are not parts of the System to be developed. This technique has been introduced recently in the RUP development method [14].

In our opinion, the conceptual model approach is not satisfactory in the cases where the entities in the domain are highly interactive and autonomous (e.g., participants in a meeting), or the most relevant aspects of the domain are naturally presented as workflows (e.g., handling an order in an e-commerce system). The business model approach overcomes the above limits, and is quite satisfactory whenever it is possible to naturally determine the "business organization"; however, it is problematic in the cases where the domain is static (e.g., the domain for a word processor concerning texts, paragraphs, documents, ...) or when, trying to find the "business organization", we fix too early the boundaries of the System to be developed. Here we propose a somewhat more general technique trying to avoid the negative aspects of both the above approaches, and such that both are particular subcases of our one.

2.2 Problem Domain Model: A Proposal

The structure of a PDM is shown in Fig. 2. We propose to model the various entities present in the domain by the Conceptual View, a UML class diagram, but where the classes may be also active, thus with a dynamic behaviour; even more we allow to model the autonomous features of their behaviour. Then, the most relevant cooperation among such entities may be modelled in the Work Case View part that consists of workflows of a special kind, named *work cases.*

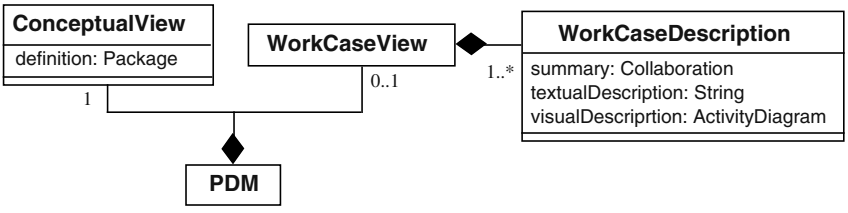
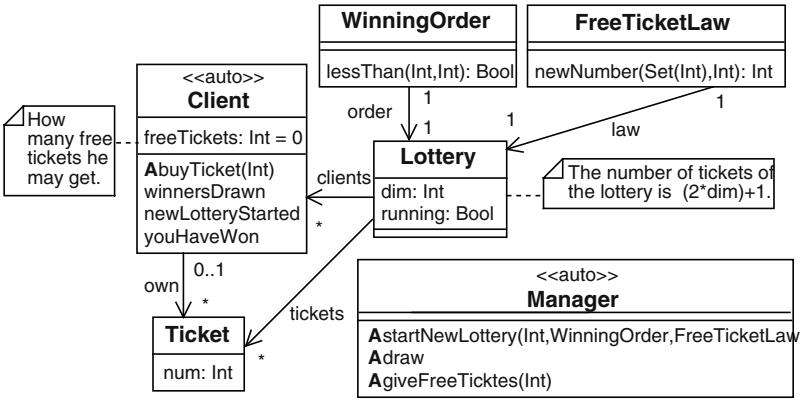


Fig. 2. PDM Structure

Conceptual View. The Conceptual View, a UML package containing at least a class diagram, makes explicit which are the entities appearing in the domain (they are modelled by objects whose classes appear in such package) and their mutual relationships, if any (modelled by associations among the corresponding classes). The other elements of the class diagram, such as attributes, operations and constraints, and the other diagrams in the package (as state charts defining the operations) may be used to model relevant aspects of such entities. In that package we may use the active class stereotype ≪auto≫ to indicate those domain entities capable of autonomous behaviour (i.e., they are not just reacting to external stimuli). An autonomous action of such entities is modelled by the self call of operations of the stereotype ≪A≫ (denoted by identifiers starting with a bold capital **A**). Below we show the Conceptual View of the AL_L case study.



context C: Client inv: C.freeTickets >= 0
context WO:WinningOrder inv: "x < y iff WO.lessThan(x,y)" is a total order
context newNumber(asTks,j):
pre: {-j, ..., +j} - asTks <> {}
post: asTks->excludes(result) and -j <= result and result <= j
context L: Lottery inv:
All the tickets in L.tickets have different numbers and
L.dim = 5000 * k with K >= 1 and L.tickets.num = {-L.dim ... L.dim}
Lottery.allInstances->size= 1 and Manager.allInstances->size= 1

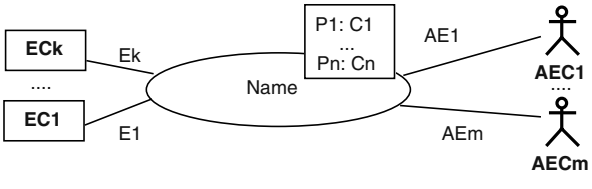
The autonomous entities appearing in that domain are the clients, which may buy the tickets, and a manager, which starts the lotteries, draws the winners and

gives out free tickets. Because these activities are autonomous, we model them by means of four ≪auto≫ operations; whereas we model other non-autonomous activities, such as to be informed of winning a prize, by plain operations (e.g., youhaveWon). In the domain there are also passive entities, describing the current lottery and its tickets. The constraints attached to the class diagram model relevant aspects of the domain entities (e.g., there is at most one running lottery, or the *winning orders* are total orders on the integer numbers).

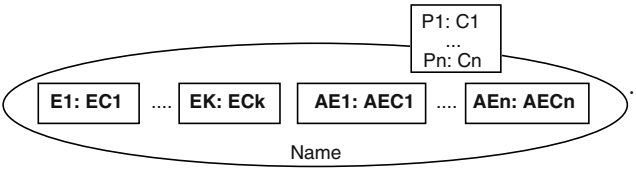
Work Case View. Technically, a work case is a variant of the UML collaboration, thus it allows to represent some cooperative effort among some entities present in the domain. As a collaboration, it has a name, precisely defines (the roles of) the participants, and may have some parameters. But we prefer to model the behaviour of a collaboration by means of an activity diagram expressing the causal relationships among actions made by the participants, instead of by a set of interactions (message exchanges). In this way we can just describe the causal/temporal relationships among relevant actions made by the participants without necessarily presenting such actions as messages sent by someone to some other one. The reason of this choice is to keep the description of a work case quite abstract avoiding to introduce spurious objects (just to have someone calling some operations) or to make particular choices about who calls who.

Notice that there is a clear difference between our work cases and the RUP business use cases. Indeed all the participants in a work case are modelled by the roles of the work case (recall it is a variant of a UML collaboration), whereas in a business use case there exists a business organization, which is a special implicit participant interacting with all the other ones (business actors), and in general the latter do not interact each other.

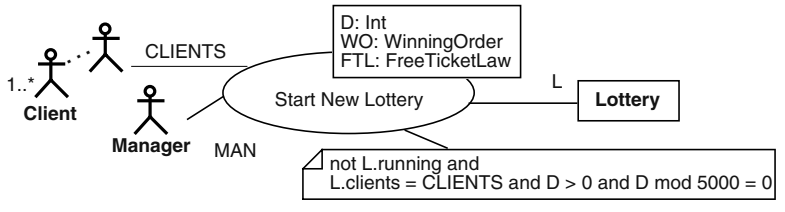
The description of a work case consists of three parts (see Fig. 2). The main part is a, possibly parameterized, UML collaboration, where its roles correspond to all the participants in the work case. Since we do not describe the work case behaviour by a collaboration diagram, we prefer to visually represent the corresponding collaboration in the following way:



where Name is the work case name; P1, ... Pn are the parameters; AE1, ... AEm, E1, ... Ek are the roles (to be played by domain entities) of the participant in the work case; C1, ... Cn, AEC1, ... AECm, EC1, ... ECk are classes appearing in the Conceptual View. We distinguish the class of the autonomous entities by using the icon  . Using the standard UML notation it should be represented by



Since we can attach to the collaboration icon a constraint, we can state, if any, which conditions the participants and the parameters of the work case have to satisfy. Then, a work case description contains a textual description made by using the natural language. It must start with a sentence of the form "When ..." expressing under which conditions the considered domain entities may take part to the work case, and must consist of sentences where the subjects are autonomous participants and where the object complements are participants. The last part of a work case description is a visual presentation of its behaviour by means of a UML activity diagram. The action-states of such diagram can be only calls of the operations of the work case participants, and the conditions properties on the states of the work case participants. In Fig. 3 we present one work case of the AL_L PDM (the remaining ones: Buy Ticket, Draw Winners and Give Free Tickets can be found in [3]). This work case is quite simple; it just says



textual When no lottery is running, the manager may start a new one giving the dimension of the lottery (a natural greater than 0 and multiple of 5000), the law for generating the numbers of the free tickets (a function which given a set of integers finds a new number not belonging to it) and a total order on integers, which will be used to find the winners.
All clients, will be informed of the new lottery.
Then, a lottery is running and is characterized by the data given by the manager, and all its tickets are available.

behaviour

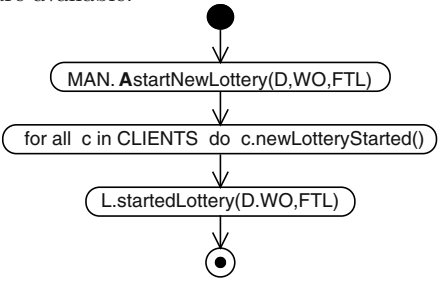


Fig. 3. AL_L PDM: Work Case Start New Lottery

under which conditions the manager my start a new lottery and what happens when he does that (the clients are informed, and the characteristics of the new lottery are recorded by the Lottery domain entity).

To keep the presentations of the behaviour views of the work cases simple and quite readable, we strongly suggest defining appropriate additional operations, similarly to those used in [19] for presenting the "well-formedness rules". For example, in the work case Start New Lottery we have used the operation startedLottery to describe the update of the Lottery domain entity.

context Lottery::startedLottery(D:Int,WO: WinningOrder,FTL:FreeTicketLaw)
post: self.running and self.dim = D and
self.winningOrderr = WO and self.FreeTicketLaw = FTL and
self.availableTickets.num = { -D ... D}

3 Capturing and Specifying Requirements

3.1 System Placement

Once we have given the PDM, the next step of the development of the System is "to place it" in the domain by making precise which problem it must solve. This task consists of the following activities:

1. add a class for System to the class diagram in the Conceptual View;
2. decide which entities of the domain will be encompassed in the System; that is, if they are autonomous their activities will be realized by the System, otherwise the data that they contain will be preserved by the System; place them inside the icon of the System class;
3. decide which entities of the domain will interact with the System; connect them with the icon of the System class by a line;
4. decide if the System needs to cooperate with further external entities (not present in the domain); usually they are devices or entities offering services to support the System activity; add them as new classes to the diagram and connect them with the icon of the System class by a line;
5. decide which work cases the System will support (clearly all their participants have to be included in those considered at points 3 and 4; for each of them place the corresponding collaboration pictures over the class diagram.

After having performed the above tasks, you have got what we call "System Placement Diagram", where for simplicity we drop attributes, operations, and associatian multiplicities.

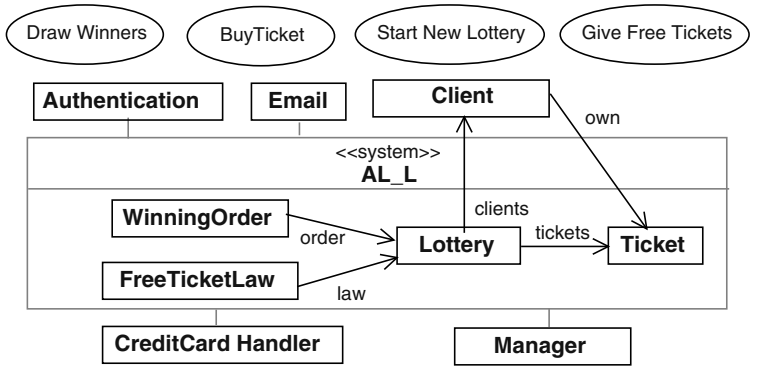
Notice that placing the System includes of course the definition of its boundary, which is recognized to be an important task almost in any development method (see [17,11]). If we consider the AL_L case study, we can see how we can place different systems in the domain described by the PDM given in Sect. 2.2. For example:

a) The System must completely automate the handling of the lottery using Internet, and taking advantage of an external authentication service and of a credit card service for the payments.

b) As for the previous case, but the System will not replace the manager deciding, e.g., when to draw the winners, and email will be used for some communications with the clients.

c) The System just helps the clerks to sell the paper tickets to the clients by showing the available tickets, printing the tickets, generating the list of the winning tickets, and printing the free tickets, which will be given by the clerks to the clients that show a paid ticket.

For what concerns the work cases all of them will be supported by the above systems. In this paper we consider case **b)**, and below we show the resulting System Placement Diagram. This diagram will be the starting point to capture and specify the requirements.



3.2 Overall Structure of a Requirement Specification

In our approach, the Requirement Specification artifacts consist of different views of the System, plus a part, Data View, needed to give a rigorous description of such views. Its structure is shown in Fig. 4 by a UML class diagram.

Context View describes the context of the System, that is which entities (*context entities*) and of which kinds may interact with the System, and in which way they can do that. Such entities are further classified into those taking advantage of the System (*service users*), and into those cooperating to accomplish the System aims (*service providers*). That explicit splitting between the System and the context entities should help avoid confusions between what exists and needs just to be precisely described (context entities) and what instead has to be developed (System) on which we have to find (capture) the requirements The further splitting between users and providers should help distinguish which context entities cannot be modified by the developer (providers), and those which may be partly tuned by the developer (users), e.g., by fixing their interface towards the System.

Use Case View, as it is now standard, shows the main ways to use the System (*use cases*), making clear which actors take parts in them. Such actors are just *roles* (*generic instances*) for some context entities depicted in the Context View.

Internal View describes abstractly the internal structure of the System, which is essentially its Abstract State. It will help precisely describe the behaviour of

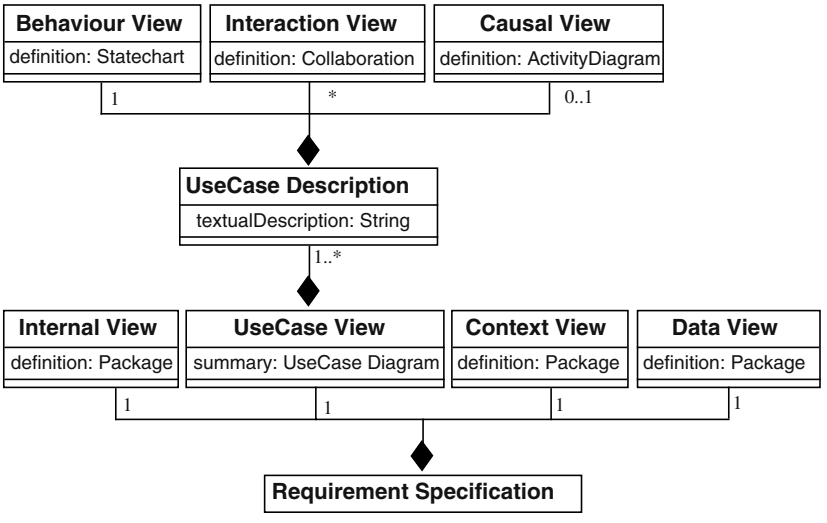


Fig. 4. Requirement Specification Structure

the use cases, by allowing to express how they read and update it. UML allows a single use case to have a proper state, but we prefer to have a unique state for all the use cases, to help model their mutual relationships.

Data View lists and makes precise all data appearing in the various views of the System to help guarantee the consistency of the concepts used in such views.

Some of the above views (e.g., Internal View and Context View) are new w.r.t. the current methods for the OO UML-based specification of requirements. In our approach, they play a fundamental role to help ensure the consistency among the various use cases and of the whole specification.

3.3 Examples from the AL_L Case Study

Here we illustrate the proposed structuring for the requirement specification artifact, showing its use in the AL_L case study. Notice that here we present the result of an activity that includes various steps and iterations. For lack of room, here we do not discuss those aspects of incremental development with feedback. We just provide a hint in Fig. 5.

Data View. The Data View for the AL_L case, see Fig. 6, is quite simple and just introduces three data types: the orders for finding the winners, the rules for finding the numbers of the tickets to be given freely, and the data needed to identify a credit card.

Context View. The Context View of the AL_L System, shown in Fig. 7, consists of a class diagram, where there is a class AL_L of stereotype ≪System≫ whose unique instance is the System, some classes of stereotype ≪SU≫ (icon ) whose instances are users of the services provided by the System (the clients and the

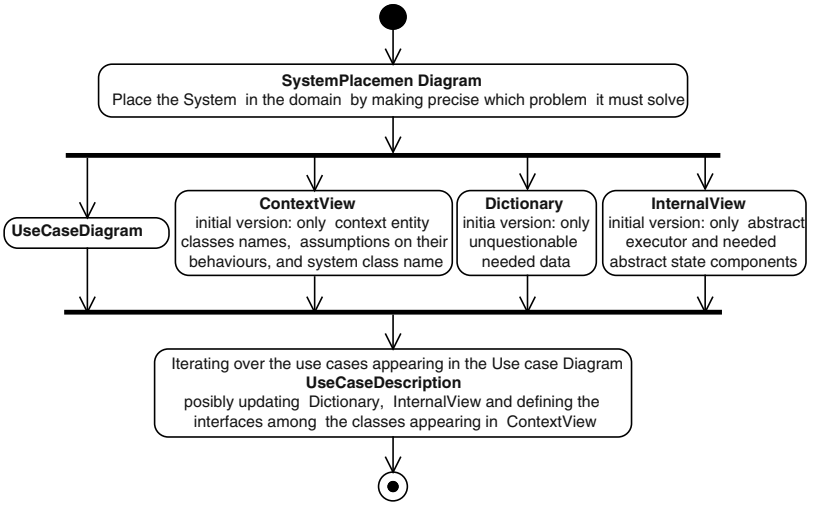
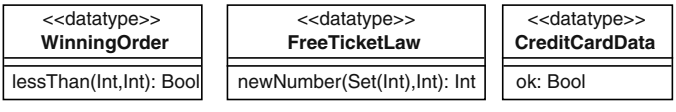
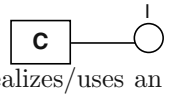
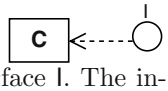


Fig. 5. Requirement Specification Tasks



context WO: WinningOrder inv: "x < y iff O.lessThan(x,y)" is a total order
context newNumber(asTks,j):
pre: {-j, ..., +j} - asTks <> {}
post: asTks->excludes(result) and -j <= result and result <= j

Fig. 6. AL_L: Data View

manager); and some classes of stereotype ≪SP≫ (icon) whose instances are providers of services used by the System (the email, the credit card service and the authentication service). In this diagram we show the mutual interfaces among these classes, that is in which way they may interact, using the standard UML *interface construct*. In Fig. 7, for example, we can see that the interface ToEmail of the **Email** context entity is really simple, just offering the possibility to receive request to send an email message.  and  visually present respectively that a class C realizes/uses an interface I. The interfaces appearing in this diagram are usually given apart (here in the bottom part of Fig. 7).

The Context View may include also some information on the behaviour of the ≪SU≫ and ≪SP≫ classes, but not of the ≪System≫ class, to model the assumptions on the behaviour of their instances on which the System relies.

Internal View. The Internal View describes at an extremely abstract level the structure (architecture) of the System. This structure consists of a unique active

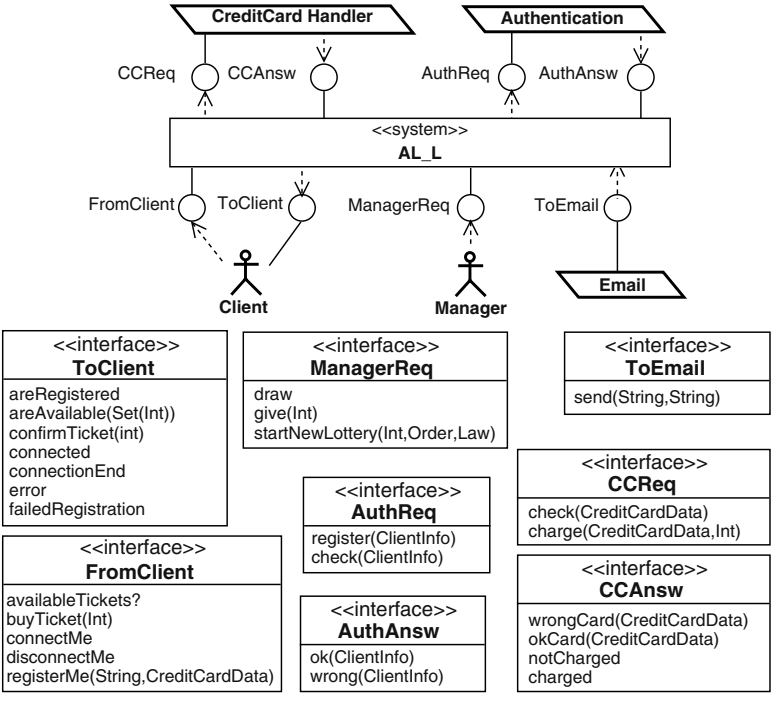


Fig. 7. AL_L Requirement Specification: Context View

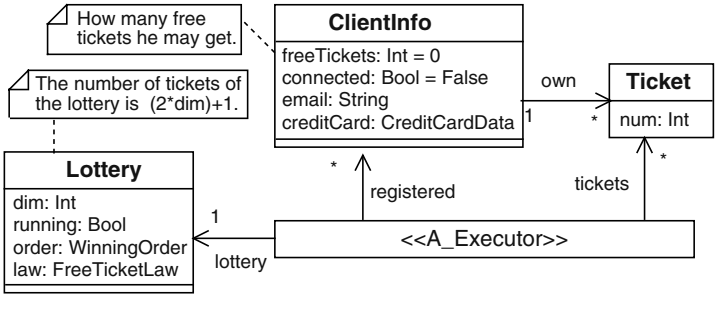


Fig. 8. AL_L Requirement Specification: Internal View

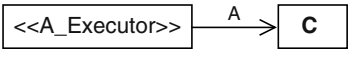
object able to perform the System activities (abstract executor) and by many passive objects describing the System Abstract State.

In Fig. 8 we show the Internal View of the AL_L case study. It consists in a class diagram containing exactly one class of the stereotype ≪A_Executor≫, and several passive classes defining the parts of the System Abstract State.

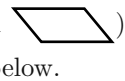
The instances of the class ClientInfo represent the information relative to the client context entities. Very frequently, the Abstract State must contain information about some context entities, and so we propose a standard way to treat

these cases We name CENTInfo the class of the information on the context entities of class CENT, and assume that its instances are in bijective correspondence with those of CENT, and thus with the context entities. Furthermore, this correspondence is supported by an operation CENT::Info: CENTInfo that returns the information element corresponding to a context entity. op(P)).

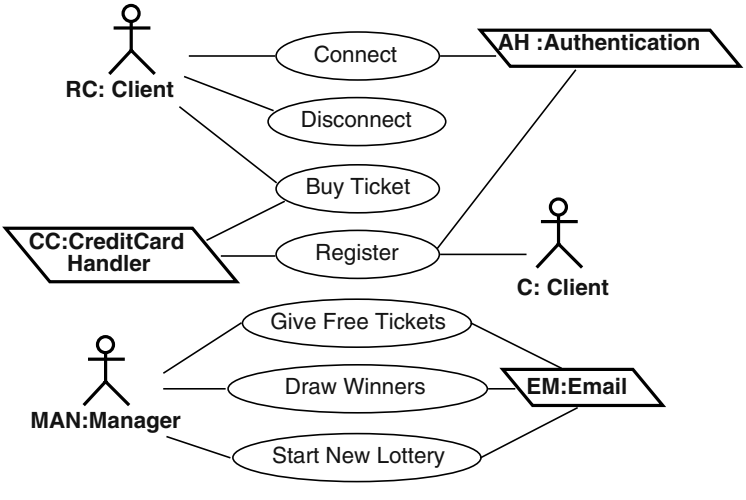
Following this approach, we avoid, on one side, models where the presence of a class named as a context entity class, say Client, requires to think about its true nature (e.g., is it a database relation ? or a kind of interface taking care of the interactions with such context entities" ? or ...), and, on the other one, precise but too much detailed models, where the association of the information to the corresponding context entities is realized, e.g., by using codes uniquely identifying the entities.

The class diagram of the Internal View describes implicitly also the "Abstract State" of the System (technically the state of the ≪System≫ class appearing in the Context View) in the following way: for each association in the diagram from the ≪A_Executor≫ class 

the ≪System≫ class has an attribute A: Bag(C).

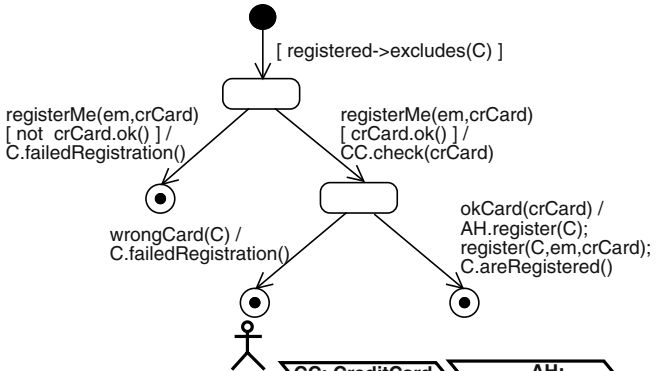
Use Case View. The Use Case View consists of a UML "Use Case Diagram" and of a Use Case Description for each use case appearing in it. But, for us the actors appearing in the Use Case diagram are possible roles for the entities outside the System interacting with it (context entities, defined in the Context View). Thus each actor will be denoted by a name, expressing the played role, and by a class, appearing in the Context View, showing which kind of context entities may play such role. Moreover, since the context entities are distinguished between users of services provided by the System and providers of services needed by the System also the actors will be distinguished in the same way. The same icons used for the context entity classes will be used for the actors ( and ).

The Use case Diagram for the AL_L case study is shown below.

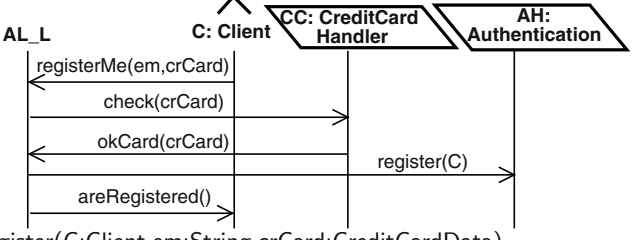


textual When a client is not registered may register himself to the lottery system by giving his email and the data of a credit card. The system check the credit card data with the credit card service, if they are ok and are validated by the credit card service, then the system registers the client with the authentication service, informs him that he has been registered, and he will be registered; otherwise the system informs him that his registration has failed.

behaviour



interaction



context AL_L::register(C:Client,em:String,crCard:CreditCardData)
post: registered->
exists(CI — CI.email = em and CI.creditCard = crCard and C.Info = CI)

Fig. 9. AL_L Requirement Specification: Use Case Register

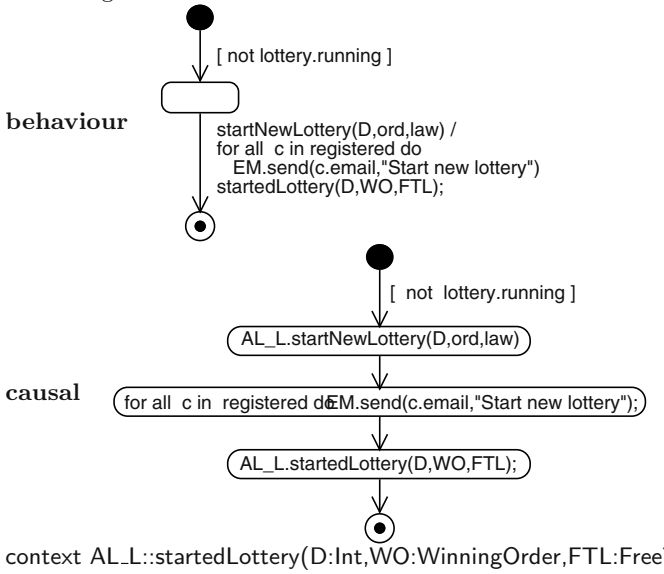
Notice how the client context entities may play two different roles, when interacting with the System, as registered client (RC) when playing with the system, and as normal client (C), when trying to register.

A Use Case Description, see those of two use cases of AL_L in Fig. 9 and 10 (the remaining use case may be found in [3]), consists of a textual presentation and of one or more views, of different kinds, of the use case.

The textual description should be expressed by sentences where the subject is either one of the actors or the System, and may start with a sentence of the form "When ..." expressing under which condition the use case may happen (pre-condition).

Any Use Case Description must include a Behaviour View, which is a statechart for the ≪System≫ class describing the complete behaviour of the System with respect to such use case. Such statechart, see, e.g., Fig. 9 and 10 has particular features. The transition from the initial state should be labelled by the "pre-condition", its events may be only call of the operations of the ≪System≫ interfaces, its conditions may test only the System Abstract State and the event

textual When no lottery is running, the manager may ask to the system to start a new one by giving its dimension (a natural greater than 1), winning order (an order on integers, which will be used to find the winners) and free ticket law (for generating the numbers of the free tickets, just a function which given a set of integers finds a new number not belonging to it). Then, a new lottery will be running having the dimension, winning order and free ticket law given by the manager.
The system will inform all the registered clients by an email message that a new lottery is running.



context AL_L::startedLottery(D:Int,WO:WinningOrder,FTL:FreeTicketLaw)
post: tickets = {}and lottery.dim = D and
lottery.Order = ord and lottery.Law = law and lottery.running = True

Fig. 10. AL_L Requirement Specification: Use Case Start New Lottery

parameters, and its actions may only be calls of the the operations of the actors, as defined by their interfaces, or actions updating the System Abstract State. To keep the behaviour views simple and quite readable we use appropriate additional operations, as previously suggested for the work cases.

The behaviour view is a "complete" description of what the System does that concerns the use case. In Fig. 9 we can see that the registration of a client requires some collaboration by the credit card service and the authentication service, that affects the System Abstract State, and that the use case has three possible cases (all of them visually presented in the diagram); whereas in Fig. 10 we see that the registered client will be informed by an email message of the new lottery, and that the use is really simple, not having any alternative way.

A Use Case Description may include any number of Interaction View, which are sequence (or collaboration) diagrams representing the interactions happening in a scenario of the use case among the context entities and the System. The Use Case Description in Fig. 9 has an Interaction View, whereas the one in Fig. 10

none. Any Interaction View must be coherent with the Behaviour View (that is, it must represent a particular execution of the complete behaviour of System described by such view).

We think that the Interaction View are really important showing visually who does what, but they are complementary to the Behaviour View because they cannot show under which conditions the various messages may be exchanged and their effects on the System Abstract State. **E**.

A Use Case Description may include also a Causal View (see for example Fig. 10), which is an activity diagram describing all the relevant facts happening during the use case and their causal relationships. The relevant facts (technically represented by action-states of the activity diagram) can be only calls of the interface operations of System by the actors, calls of the operations of the actors by System, UML actions producing side effects on the System Abstract State. In addition, the Causal View must be coherent with the Behaviour View, in the sense that the causal relationships among "facts" that it depicts may happen in the behaviour depicted by the state chart.

The various views listed above play different roles in the description of a use case and are partly complementary and partly overlapping. The choice of which of them to use depends on the nature of the considered use case. The only rule enforced by the method is that the behaviour view is mandatory, because it obliges to present all the behaviour of the use case (e.g., all possible alternative scenarios are included), even if it may be less readable than the others. However, due to the nature of the UML state chart, the behaviour view cannot be a complete description of the use case, indeed; it does not allow to express who is calling the operations to which System reacts.

4 Related Work and Conclusions

The approach that we have outlined (see [3] for an extended version with more complex case studies), here limited to the early development phases (see [4] for the design phase), is in the line of some of the best-known methods for software development, adopting a multiview and use case approach and using the UML notation. But it departs from them, at least to our knowledge, in some important respects, both from the methodological and the technical viewpoint.

First, on the method side, the overall major goal is to propose a more systematic and stringent approach, in the sense that the overall structure of our artifacts, both for the PDM and the Requirements, is constrained in order to tightly relate the components and have at hand the possibility of performing a number of consistency checks. This view contrasts with the almost total freedom given, for example in RUP [14], where the structure is just based on the use case descriptions. The same freedom, just use case diagrams and use case descriptions, is given for the Requirement Specification phase in COMET [9], in sharp contrast with the detailed structure and the many constrained guidelines and notations for Analysis and Design. That level of freedom is, on the other hand, explicitly advocated, for example in [8], on the basis that experience matters more than stringent structuring and rules. There the underlying philosophy is

admittedly the same of the Agile Methods Movement (see [18], for an interesting discussion and references). However, while we do not deny that highly skilled and experienced software developers perhaps need only loose guidelines and a liberal supporting notation, from our experience we have seen that, for less experienced people, such liberality is a source of endless discussions, contrasting choices and a proliferation of inconsistencies. Moreover, we believe that our "tight and precise" imperative and the related techniques may help from one side reduce the amount and the fuzzy verbosity of some documentation and on the other provide effective guidelines for passing to the design and then the implementation phase, though we have not yet explored all the later phases.

The approach taken in Catalysis [6], that in other details shows some similar general views to ours, is not directly comparable, being an overall transformational approach based on components that are refined from business modelling to implementation units. Definitely our way of structuring requirements is not targeted to a transformational approach; we are more interested in providing a separate step preliminary to devise in a rather structurally independent manner, a model-driven software architecture of the system. Indeed, we have already performed some experiments to pass from a requirement specification in the suggested form to a design document, for which too we have proposed a more tight and precise structuring. Our approach is totally compliant with the OMG Model Driven Architecture philosophy [12] and it is within that framework that we intend to explore the connection with the implementation phase, passing from Platform Independent Models to Platform specific Models and then to code. A second more specific methodological difference is the strict and explicit separation between the Problem Domain Model and the system, in the line for example of [10]. That distinction was and is somewhat blurred in some classical and Object Oriented approaches, though revisited with UML (see, e.g., [13,8] for very recent examples). In other approaches that distinction has been reintroduced and phrased in the distinction between Business Modelling (e.g., in [14]) and Requirements.

On the more technical side there are a number of major distinctions with the extant work, namely

- the PDM structure, encompassing conceptual modelling and business modelling;
- the System Placement activity, that encompasses the search for the system boundary;
- the use of the Context View to make explicit the distinction between the system and its environment and as a basis for defining the requirements about the interaction of the system with its context;
- the explicit use of the concept (a class) of System, both in the context diagram and in the use case descriptions, where we specify the System behavior related to a specific use case with a statechart;
- the use of a very Abstract State, instead of the many optional use case states, to allow expressing abstract requirements about the interaction of the System and the context, without providing an object-oriented structuring at a stage when such a structure is not required and can be premature.

Notice that the use of the class System is not in direct contrast with the traditional object-oriented approaches, where the presence of such a class, at the level of analysis and design, is considered a typical naive student's mistake. Still, because of the fact that those approaches also at the requirement level start with an object structure, the presence of that class is most unusual. However the danger of providing an object structure not immediately needed when defining the system requirements has been remarked by many authors (notably M. Jackson, see, e.g., [10]). Even more interesting, also in Catalysis, that claims to be completely object-oriented, a class system and a context diagram is used in the preliminary phases and it appears in the sequence diagrams explaining the role of the system (see [6, p.15, fig 1.16]). Of course the context diagram with the system initial bubble was the starting diagram in the Structured Analysis approach [20].

Finally we just mention that in our approach the choice and use of the UML constructs is guided by a careful semantic analysis (see, e.g., [15,16]), that has led us to prevent and discourage the indiscriminate use of some features that, especially in combination, may have undesirable side-effects, like interferences and ambiguities.

References

1. E. Astesiano and G. Reggio. Knowledge Structuring and Representation in Requirement Specification. In *Proc. SEKE 2002*. ACM Press, 2002. Available at `ftp://ftp.disi.unige.it/person/ReggioG/AstesianoReggio02a.pdf`.
2. E. Astesiano and G. Reggio. Consistency Issues in Multiview Modelling Techniques. In *Recent Trends in Algebraic Development Techniques, Selected Papers of the 15th International Workshop WADT'02*, LNCS. Springer Verlag, 2003. To appear. Available at `ftp://ftp.disi.unige.it/person/ReggioG/AstesianoReggio03b.pdf`.
3. E. Astesiano and G. Reggio. Tight Structuring for Precise UML-based Requirement Specifications: Complete Version. Technical Report DISI–TR–03–06, DISI, Università di Genova, Italy, 2003. Available at `ftp://ftp.disi.unige.it/person/ReggioG/AstesianoReggio03c.pdf`.
4. E. Astesiano and G. Reggio. Towards a Well-Founded UML-based Development Method. In *Proc. of SEFM Workshop*. IEEE Computer Society, Los Alamitos, CA, 2003. To appear. Available at `ftp://ftp.disi.unige.it/person/ReggioG/AstesianoReggio03g.pdf`.
5. E. Astesiano, G. Reggio, and M. Cerioli. From Formal Techniques to Well-Founded Software Development Methods. In *Proc. of The 10th Anniversary Colloquium of the United Nations University International Institute for Software Technology (UNU/IIST): Formal Methods at the Crossroads from Panacea to Foundational Support. Lisbon - Portugal, March 18-21, 2002.*, LNCS. Springer Verlag, Berlin, 2003. To appear. Available at `ftp://ftp.disi.unige.it/person/ReggioG/AstesianoEtAll03a.ps`.
6. D. F. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Object Technology Series. Addison-Wesley, 1999.
7. G. Engels, J.M. Kuester, and L. Groenewegen. Consistent Interaction of Software Components. In *Proceedings of IDPT 2002*, 2002.

8. M. Fowler and K. Scott. *UML Distilled: Second Edition*. Object Technology Series. Addison-Wesley, 2001.
9. H. Gomaa. *Designing Concurrent, Distributed and Real-Time Applications with UML*. Addison-Wesley, 2000.
10. M. Jackson. *Software Requirements & Specifications: a Lexicon of Practice, Principles and Prejudices*. Addison-Wesley, 1995.
11. Pfleeger S. L. *Software Engineering: Theory and Practice*. Prentice Hall, 2001.
12. OMG Architecture Board MDA Drafting Team. Model Driven Architecture (MDA). Available at `http://cgi.omg.org/docs/ormsc/01-07-01.pdf`, 2001.
13. Stevens P. and Pooley R. *Using UML: Software Engineering with Objects and Components*. Addison-Wesley, 2000.
14. Rational. Rational Unified Process© for System Engineering SE 1.0. Technical Report Tp 165, 8/01, 2001.
15. G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML Active Classes and Associated State Machines – A Lightweight Formal Approach. In T. Maibaum, editor, *Proc. FASE 2000*, number 1783 in LNCS. Springer Verlag, Berlin, 2000.
16. G. Reggio, M. Cerioli, and E. Astesiano. Towards a Rigorous Semantics of UML Supporting its Multiview Approach. In H. Hussmann, editor, *Proc. FASE 2001*, number 2029 in LNCS. Springer Verlag, Berlin, 2001.
17. J. Sommerville. *Software Engineering: Third Edition*. Addison-Wesley, 1989.
18. DeMarco T and Boehm B. The Agile Methods Fray. *Computer*, 2001.
19. UML Revision Task Force. *OMG UML Specification 1.3*, 2000. Available at `http://www.omg.org/docs/formal/00-03-01.pdf`.
20. E. Yourdon. *Modern Structured Analysis*. Prentice-Hall, 1989.

Integrating Performance Modeling in the Software Development Process

Simonetta Balsamo and Marta Simeoni

Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy
{balsamo,simeoni}@dsi.unive.it

Abstract. We discuss the integration of performance modeling and analysis in the software development process. Various approaches have been recently defined to integrate performance models and specification languages and models to derive or validate non-functional properties of a software system. Such integration of quantitative performance analysis should provide feedback easily understandable by the software designer and system developers. A framework that allows the combination of different performance modeling techniques and methods, defined at different levels of abstraction, should better support performance analysis and validation of complex and heterogeneous software systems during the software development process.

1 Introduction

Performance analysis of software systems is a critical issue in the software development process. Modeling is an important and useful approach for performance evaluation and system validation and it can provide prediction and comparison of design alternatives.

Software performance engineering deals with the representation and analysis of the software system dynamic based on performance models to provide feedback in the software development process. Several approaches have been defined in the last decade to integrate performance models and specification languages and models for deriving or validating non-functional properties of software systems [44,40,41].

Most of these integrated approaches are based on various formal specification models ranging form process algebra and Petri net models as well as more recent modeling languages such as UML, and they derive performance evaluation models based on different formalisms, such as queueing networks, stochastic process algebras, stochastic Petri nets and Markov processes.

An important issue of the integration of quantitative performance analysis in the software process is the ability to provide feedback that can be easily interpreted by the software designer and system developers.

Since the different performance models can be applied to represent the software system at different levels of abstraction and have different modeling constraints and solution methods, one can take advantage of the relative merit of

M. Wirsing et al. (Eds.): RISSEF 2002, LNCS 2941, pp. 35–49, 2004.

each of them. We discuss how the definition of a framework that allows the combination of different performance modeling techniques and methods should better support performance analysis and validation of complex and heterogeneous software systems, at different levels of abstraction during the software development process. Specific software system characteristics such as heterogeneity, scalability and mobility in distributed system could be considered and modeled for the software performance analysis.

2 From Software Specification to Performance Model

The Software Performance Engineering (SPE) methodology introduced by Smith in [44] has been the first comprehensive approach to the integration of performance analysis into the software development process, from the earliest stages to the end. More recent approaches focus on the derivation of a performance model right from the Software Architecture (SA) specification [2,6,45,15], thus allowing for a choice among alternative architectures on the basis of quantitative aspects [43].

In this section we briefly discuss some recent methodologies for the derivation of performance models from SA specifications. First we recall the commonly used specification languages focusing on how they are used to derive performance models, and then we introduce the main types of performance models and present some software performance approaches.

2.1 Specification Languages

The recent approaches to derive performance models from SA refer to various specification languages, ranging from the Unified Modeling Language (UML) [11,47], or other graphical languages such as Message Sequence Charts (MSC) [26], to formal specification languages like process algebras and Petri nets. The choice of UML is motivated by the fact that UML is nowadays a widely used notation for the specification of software systems. It provides various diagrams allowing the description of different system views, which capture static and behavioral aspects, interactions among system components and implementation details. On the other hand, the choice of a formal specification language based on Process Algebras or Petri nets is motivated by the possibility of integrating functional and performance aspects and analysis into a unique formalism [21].

In order to derive performance models from UML specifications, it is usually necessary to complete the UML diagrams with additional performance related information, which is associated to the diagrams by means of simple annotations or extensions based on stereotypes and tagged values. The recently defined UML Profile for Scheduling, Performance and Time [32] allows the adding of such information in a standard way.

The approaches which derive performance models from specification based on process algebras and Petri nets usually refer to their stochastic extensions, allowing action or firing duration to be expressed by random variables.

2.2 Performance Models

The main classes of performance models are queueing networks (QN), stochastic timed Petri nets (STPN) and stochastic process algebras (SPA) [28,21]. Each of these classes has its own peculiarities in terms of expressiveness (i.e., the kind of system that can be suitably modeled), level of abstraction (i.e., the degree of detail needed to describe the system), and efficiency of the solution methods. However, QN are traditionally the most commonly applied to evaluate system performance, because of their relative high accuracy in performance results and their efficiency in model analysis and evaluation.

Basic QN models represent resource sharing systems. Their extension called Extended Queuing Networks (EQN) allows also the representation of other interesting features of real systems, such as synchronization and concurrency constraints, finite capacity queues, memory constraints and simultaneous resource possession [28]. Another extension of QN models is the class of Layered Queuing Network (LQN), which models client-server communication patterns in concurrent and/or distributed software systems [42].

A performance model can be analyzed by analytical methods or by simulation, in order to evaluate a set of performance indices such as resource utilization, throughput, customer response time and others. Simulation is a widely used general technique, whose main drawback is the potential high development and computational cost to obtain accurate results. On the other hand, analytical methods require that the model satisfies a set of assumptions and constraints, and are based on a set of mathematical relationships that characterize the system behavior.

The analytical solution is often derived by considering an underlying stochastic process, usually a discrete-space continuous-time homogeneous Markov chain (MC). The solution of the associated MC is in general numerically expensive because its state space grows exponentially with the number of components of the performance model. However, some efficient analytical algorithms have been defined for performance models that satisfy some given constraints, such as product-form QN.

Besides being a possible solution technique for the introduced performance models, simulation is also used as a performance model by itself. Existing simulation tools provide a specification language for the definition of simulation models and a simulation environment to execute simulation experiments.

2.3 Some Software Performance Approaches

We discuss now some recent approaches to derive performance models from SA specifications. We consider a few main methodologies focusing on the knowledge about performance evaluation techniques required to the system designer and developer in order to apply the transformation methodology.

SPE Based Methodologies

Some of the methods proposed in the literature [46,16,37,36,20] refer to the SPE methodology [44], which is based on two models: the software execution model

and the system execution model. The former is based on execution graphs and represents the software execution behavior; the latter is based on QN models and represents the system platform, including hardware and software components. The analysis of the software model gives information concerning the resource requirements of the software system. The obtained results, together with information about the hardware devices, are the input parameters of the system execution model, which represents the model of the whole software/hardware system.

Among the approaches based on SPE we consider the one developed by Cortellessa and Mirandola in [16]. They propose the PRIMA-UML methodology, which makes a joint use of information from different UML diagrams to incrementally generate a performance model representing the specified system. SA are specified by using Deployment, Sequence and Use Case diagrams. The software execution model is derived from the Use Case and Sequence diagrams, and the system execution model from the Deployment diagram. Moreover, the Deployment diagram allows the tailoring of the software model with respect to information concerning the overhead delay due to the communication between software components. Both Use Case and Deployment diagrams are enriched with performance annotations concerning workload distribution and devices' parameters, respectively. Hence, in order to apply the PRIMA-UML methodology, the software designer has to know these data and how to specify them. As an example of a SPE based approach, we illustrate a simple application of the PRIMA-UML methodology.

Example 1. Consider the architecture of a multiphase compiler shown in Figure 1, where the components are the Lexer (lexical analyzer), the Parser (syntactic analyzer), the Checker (semantic analyzer), the Optimizer and the code Generator. The Optimizer is an optional component, in the sense that the user can select a simple or optimized compilation.

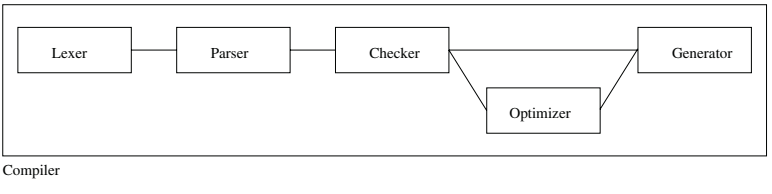


Fig. 1. Static description of the multiphase compiler.

The PRIMA-UML methodology applied to evaluate the performance of the compiler system starts with the UML specification. The Use-Case diagram shown in Figure 2.(a) considers one type of user and two different use cases, that are the simple and the optimized compilation. Note that the Use Case diagram has been enriched with the data p and $1 - p$ representing the probabilities that the user asks for a simple or optimized compilation, respectively. We assume that the compiler is allocated on a single machine, as shown in the Deployment diagram

Fig. 2. Use Case diagram (a) and Deployment diagram (b) of the compiler system.

in Figure 2.(b). This diagram should be enriched with information concerning the specific hardware platform.

The behavior of the optimized compilation is illustrated by the Sequence diagram of Figure 3.(a). The Sequence diagram of the simple compilation can be immediately obtained by removing the optimization phase performed by the Optimizer component.

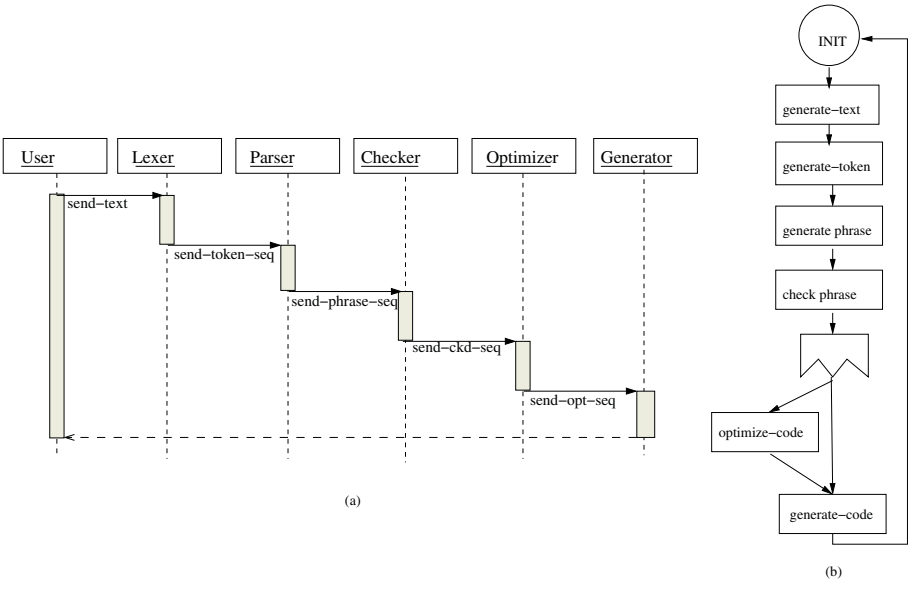


Fig. 3. (a) Sequence diagram for the compiler system with code optimization; (b) *Meta* execution graph of the sequential multiphase compiler.

The PRIMA-UML methodology derives the system execution model (i.e., the QN model) from the Deployment diagram, and the software execution model from the Use Case and Sequence diagrams.

Since the compiler is entirely allocated on a single machine, the corresponding QN model is simply composed of a single service center. The obtained system is completely sequential. However, one can use different allocation strategies to distribute the compiler components into a set of connected machines. Then,

the corresponding QN model has one service center for each machine, and the compilation phases allocated on different machines proceed in parallel.

Concerning the software execution model, the methodology generates one *meta* execution graph for each Sequence diagram and then merges the obtained graphs together into a unique model, which is shown in Figure 3.(b). The execution graph refers to a single compiler execution and contains two types of nodes. Basic nodes correspond to the various software components and are labeled with the name of the performed actions. The case node represents the alternative execution of the simple or optimized compilation. The arrows connecting the nodes describe the component interactions and the flow of control.

The term *meta* execution graph indicates that the activities of the various components are just named at this level. The final execution graph, called execution graph *instance*, can be obtained by substituting the action name of each node with the corresponding numerical parameters, that define the component resource demands. Note that while the methodology automatically generates the *meta* execution graph, the numerical parameters of the final model have to be defined by the designer. They are identified on the basis of information derived from the system execution model and the knowledge about the resource capabilities. The final performance model is the QN model parametrized by the analysis of the software model and is evaluated to compute the performance indices of the software system.

Petriu and co-authors present in [37,36,20] other approaches based on the SPE methodology. They propose three conceptually similar approaches where SA are described by architectural patterns, such as pipe and filters, client/server, broker, layers, critical section and master-slave, whose structure is specified by UML-Collaborations and whose behavior is described by Sequence or Activity diagrams. The three methods use graph transformation techniques to build LQN models out of complex SA based on combinations of the considered patterns. The approaches in [20,36] add performance related information to UML diagrams according with the UML Performance Profile [32].

Trace Based Methodologies

Besides the SPE based approaches, other proposals [1,2,45,27,35] still derive QN based performance models from SA specification. In particular two approaches described in [2,45] are based on the generation and analysis of traces (sequence of events/actions) gathered from scenarios describing the behavior of the software system under consideration.

The first approach [1,2] proposes an algorithm that automatically generates QN models from SA specifications described by Message Sequence Charts (MSC) [1] or by Labeled Transition Systems (LTS) [2]. The approach analyzes the SA dynamic specification in terms of the execution traces (sequences of events) it defines, in order to single out the maximum degree of parallelism among components and their dynamic dependencies. A key point of this approach is the assumption that the SA dynamics, described by MSC or LTS, is the only

available system knowledge. This allows the methodology to be applied when information concerning the system implementation or deployment are not yet available. An example of application of this methodology to the compiler system of Example 1 is given in [2], where three different SA are compared. We refer to [2] for the complete description of the example and for the derivation of the corresponding QN models.

The second trace-based approach has been developed by Woodside et al. [45], and propose the automatic derivation of a LQN model from a commercial software design environment called ObjecTime Developer [33], by means of an intermediate prototype tool called PAMB (Performance Analysis Model Builder). ObjecTime Developer allows the designer to describe a set of communicating actor processes, each controlled by a state machine, plus data objects and protocols for communications. It is possible to "execute" the design over a scenario by inserting events, stepping through the state machines, and executing the defined actions. Moreover, the tool can generate code from the system design, that can be used as prototype or as a first version of the product. The approach in [45] takes advantage of such code generating and executing scenarios capabilities for model-building: the prototype tool PAMB, integrated with ObjecTime Developer, keeps track of the execution traces, and captures the resource demands obtained by executing the generated code in different execution platforms. The trace analysis allows the building of the various LQN submodels, one for each scenario, which are then merged into a global model, while the resource demands data provides the model parameters. After solving the model through an associated model solver, the PAMB environment reports the performance results through performance annotated MSC and graphs of predictions. Note that this approach requires less knowledge about performance aspects to the software designer, since the performance parameters are automatically evaluated by the PAMB tool.

Methodologies Based on Formal Specification Languages

Some authors propose the translation of UML models into stochastic Petri net models or stochastic process algebra specifications [38,29,10] in order to perform quantitative evaluation of software systems. However, the more mature approaches based on formal specification languages such as STPN and SPA do not combine UML with SPA or STPN. These methods allows integrating functional and non-functional aspects into a unique reference framework and model for both SA specification and performance analysis. Formal specification languages make SA specification and its functional analysis more rigorous and well-founded.

However, from the performance evaluation viewpoint, the analysis of such models usually refers to the numerical solution of the underlying Markov chain which can easily lead to numerical problems due to the state space explosion. Another disadvantage of these approaches is the knowledge required to the software designer to specify the software system with process algebras or Petri nets, and to define the appropriate parameters for action or firing duration.

In particular several stochastic extensions of process algebras have been recently proposed, such as TIPP (TIme Processes and Performability evaluation) [18], EMPA (Extended Markovian Process Algebra) [9,8] and PEPA (Performance Evaluation Process Algebra) [24]. The main differences between these SPA concern the expressivity of their languages. They associate exponentially distributed random variables to actions and provide the generation of a Markov chain out of the semantic model (a Labeled Transition System enriched with time information) of a specified system. Furthermore, they are supported by appropriate tools, the TIPP tool, Two Towers and PEPA Workbench, respectively.

In this setting, Balsamo et alt. introduced in [6] a SPA based Architectural Description Language (ADL) called Æmilia, whose semantics is given in terms of EMPA specifications. The introduction of an ADL aims at easing the software designer in identifying the system components and their interconnections. Æmilia provides a formal framework for the compositional, graphical, and hierarchical modeling of software systems and is equipped with some functional checks for the detection of possible architectural mismatches. Moreover the authors propose a systematic approach to the translation of Æmilia specifications into basic QN models, with the aim of taking advantage of the orthogonal strengths of the two formalisms: formal techniques for the verification of functional properties for Æmilia (SPA in general), and efficient performance analysis for QN.

The example described in Example 1 is applied for this methodology in [6], where its Æmilia specification and its translation into the QN model is presented.

Simulation Based Methodologies

Another class of integrated software performance approaches are based on simulation models. The method proposed by De Miguel et alt. in [17] uses simulation packages in order to define a simulation model, whose structure and input parameters are derived from UML diagrams. The approach focuses on real-time systems and proposes extensions of UML diagrams (based on the use of stereotypes, tagged values and stereotyped constraints) to express temporal requirements and resource usage. The SA is specified using the extended UML diagrams without restrictions on the type of diagrams to be used. Like some previous approaches, this technique requires the software designer to know how to specify the timing and performance parameters in order to use the whole framework. The resulting diagrams are used as input for the automatic generation of the scheduling and simulation models via two components called Analysis Model Generator (AMG) and Simulation Model Generator (SMG), respectively. In particular, SMG generates OPNET models [34], by first generating one submodel for each application element and then combining the obtained submodels into a unique simulation model. The approach provides also a feedback mechanism by including the simulation results in the tagged values of the original UML diagrams.

To conclude, we point out that most of the existing methodologies do not yet provide a complete framework for the automatic derivation of performance models from SA specification. Most of them do not yet integrate the various steps

concerning software specification, model transformation, performance evaluation and feedback to the software designer into a unique environment properly supported by tools. In particular just a few approaches address the issue of reporting or interpreting the performance results at the SA level. However, this is an important feature which has to be considered in order to make the performance evaluation of SA really complete and useful in the software process.

3 Open Problems and Perspectives

Performance modeling and analysis in the software development process is based on the generation of appropriate performance models that allow high integration of functional and behavioral models, as discussed in the previous section.

For each class of performance models, from queueing networks to stochastic process algebras, stochastic Petri net to Markov processes, we can identify specific advantages, constraints and peculiarities in terms of expressiveness, efficiency of the solution methods and level of abstraction. A detailed review of software performance approaches can be found in [5]. In this section we discuss some open problems and perspectives in software performance modeling.

An Integrated Framework for Software Performance

We can take advantage of the characteristics of the modeling formalism to define a combined or integrated framework for software performance analysis, by combining different types of models. Such framework should better support performance analysis and validation of complex and heterogeneous software systems at different levels of abstraction during the software development process. By referring to different levels of abstraction in the development process, one can identify and select the appropriate performance model that can be applied to evaluate the software system. This corresponds to model the appropriate features and details of the software system that allow the description its dynamic behavior.

This performance modeling approach is illustrated in Figure 4. Starting from an appropriate software specification, from the description of its dynamic behavior one can choose and derive a performance model that allows representing the relevant characteristics at the selected level of abstraction. By applying the corresponding transformation algorithm one defines a performance model, selecting from analytical models, such as QN Markov chains, STPN and SPA, simulation models and possibly other types of models such as hybrid models.

Once the appropriate performance model has been identified, one can apply the corresponding performance evaluation methods. When an analytical performance model is selected, solution methods can be symbolic for some particular classes, such as some simple Markov chains and the class of product-form QN [28], so allowing an easier parametric analysis and result interpretation at the software level. More generally, analytical models can be analyzed through approximate numerical algorithms, such as for EQN and LQN [42], and for more

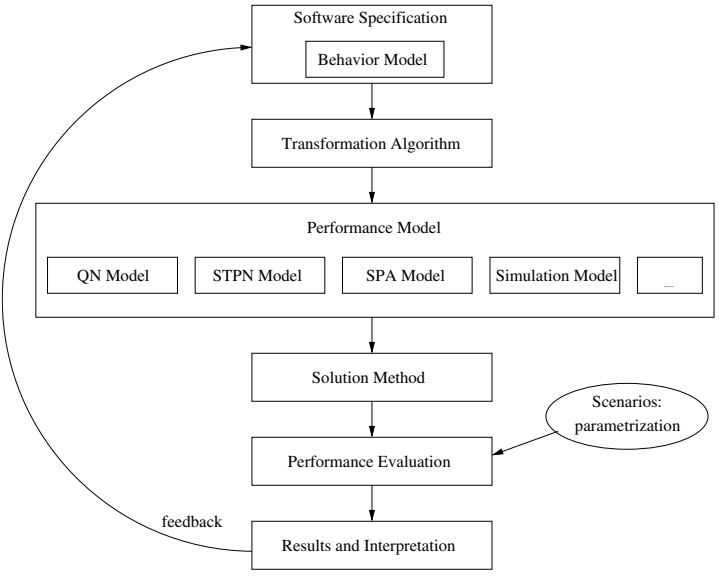


Fig. 4. Performance modeling of software specification.

complex Markov chains. STPN and SPA are usually analyzed by defining and numerically solving the underlying Markov chain. Simulation models are analyzed by discrete simulation techniques. The analysis and solution of the performance model requires the instantiation of system parameters, whose type and number depend on the abstraction level of the model. Parameter instantiation can be determined by a scenario driven approach.

Hence, the selection of the performance solution method depends on the selected performance models and parameters and can provide different degrees of accuracy in evaluating the performance figures of merit. The level of abstraction of the model also determines the number of parameters to be derived. Hierarchical modeling and a top-down approach can be applied to increase the level of details in successive steps of the modeling process, as discussed in the next subsection.

An important feature of this performance modeling framework, based on various software performance models, is to provide the designer a feedback that has to be easily interpreted by the software designer without specific knowledge of performance modeling techniques. This feature strongly depends on the selected performance modeling approach.

Therefore the selection of the appropriate model should be provided by the performance environment and driven by several issues, as discussed in the previous section. These include the software specification model or language, the relevant characteristics of the software systems, the accuracy of the performance analysis required in the software development step and the ability to provide feedback to the software designer.

Specific software system characteristics such as heterogeneity, scalability and mobility in distributed systems could be considered and represented by the appropriate model for the software performance analysis. For example a few recent approaches have been proposed for performance analysis of mobile software systems, e.g. [15,19].

Another issue in selecting the appropriate software performance methodology is the availability of supporting tools. Although several methodologies and approaches have been proposed, so far only few of them have been completely automated. Some prototypes have been recently presented, as discussed in the previous section, such as the PAMB tool for real-time interactive software based on LQN [45], UCM2LQN tool [35] based on Use Case Maps and various tools for Stochastic Process Algebras, e.g., TIPP [18], EMPA [9,8] and PEPA [24].

Hierarchical Modeling

An integrated software performance environment should allow successive refinements of performance modeling and tools based on various models, corresponding to different levels of abstraction, to be applied at different phases of the software development process, driven by the specified requirements. Then the scheme illustrated in Figure 4 can be iterated by considering different performance models to obtain more complete performance results. Additional information should be possibly integrated in successive steps to define either different or more refined performance models that allow further software system evaluation.

Considering a top-down approach, at the high levels of abstraction in the software development process we should select simpler high-level performance models with a limited number of parameters so that they can be easily derived by a possible incomplete software specification and some hypotheses on the possible scenarios of interest. Models like Use Case Maps and simple product-form QN appear to be possible appropriate choices at this step, and simple solution techniques, such as symbolic solution methods or simple bound analysis allow the evaluation of performance results also from incomplete descriptions of the software specification. At more detailed levels of abstraction in the software development process a more complete and detailed software specification can drive us to apply software performance methodologies based on more sophisticated performance models. Depending on the considered specification language and the goal of the analysis we can consider integrated methodologies based on SPA, EQN, LQN and STPN. In particular by applying SPA or STPN one can take advantage of the integration of the performance and specification formalism. Such more detailed models usually require a more complete set of system parameters to be instantiated and possibly a high computational complexity in the evaluation of performance indices.

In this framework it is relevant to investigate the application of methodologies to derive structured performance models at different levels of abstraction, where the solution of a model at a given level can be obtained extending the solution of a related model at the previous level [12,14]. These methodologies are based on the decomposition and aggregation principle and can be applied to various

classes of performance models, from Markov chain to QN, to STPN and SPA [28,25,3,30].

In line with hierarchical modeling in engineering context, describing a complex system with a hierarchical performance model means applying a top-down decomposition technique. Starting from an abstract model of the system, each step defines a more refined model of the same system, which is composed of interacting submodels that can be further refined in successive steps. Performance analysis of hierarchical models usually starts from the most detailed model and requires the application of a bottom-up aggregation technique [14]. Roughly speaking, the model of each level is analyzed first by solving each submodel in isolation and then by aggregating all the submodels into the model of the higher level. An aggregation technique defines a simpler and smaller model that is equivalent to the original one with respect to some performance indices. For example, flow-equivalent aggregation for product-form QN defines an aggregated QN where each subnetwork is substituted by a single service center. Such smaller QN is equivalent to the original one in terms of average performance indices and queue length distribution. Similarly, lumping of Markov processes and aggregation of SPA and STPN define new reduced models that are equivalent to the original ones with respect to the steady state distribution [28,25,30]. Performance analysis of hierarchical models can be also carried out through a top-down approach by disaggregation techniques. These methods define a more detailed performance model starting from a high-level model, by specifying parameters constraints so that the new disaggregated model is equivalent to the original one with respect to a set of performance indices. For example, synthesis of product-form QN defines a disaggregated queueing network from a smaller network, where a service center is substituted by a larger subnetwork, keeping the same average performance indices and queue length distribution [7].

In this framework of hierarchical performance modeling one can also apply a hybrid approach to analyze the different submodels: *hybrid models* apply mixed analytical and simulation techniques to evaluate the solution of various submodels and to combine their results in order to evaluate the performance of the whole system. Hybrid methods exploit the specific features of each submodel to take advantage of the relative merit of the different solution technique, i.e., generality of simulation and efficiency of analytical methods.

Another relevant issue in this framework of software performance is to integrate various models into the same framework, to identify and take advantage of the specific characteristics of the various classes of performance models. Starting from the software performance methodologies that derive a class of performance models from the software specification, some examples of integrated or combined approaches have been recently presented. The comparison of LQN and SPA and their application to performance validation is described in [22]. The combination of SPA and basic QN models for SA analysis have been introduced in [6] to take advantage of formal techniques to verify functional properties for the former model and efficient solution algorithms for the latter. The combination of QN and generalized STPN for the solution of complex models of system behavior

is described in [4]. An automated software design performance environment has been presented in [45], and a performance approach that considers software in a distributed mobile environment has been recently introduced in [15].

Further research has to investigate a more complete integration of different models and solution techniques, including simulation and hybrid methods, into the software performance process, exploiting the identification of appropriate and representative models and the problem of how to efficiently feedback the performance results at the software specification level.

Acknowledgments

We would like to thank Antinisca Di Marco and Paola Inverardi for useful discussions, comments and suggestions about the translation methodologies described in this paper. We would like to thank Raffaela Mirandola for useful discussions about the PRIMA-UML methodology.

References

1. F. Andolfi, F. Aquilani, S. Balsamo, P. Inverardi "Deriving Performance Models of Software Architectures from Message Sequence Charts". In [40], pp. 47–57, 2000.
2. F. Aquilani, S. Balsamo, P. Inverardi "Performance Analysis at the Software Architectural Design Level". *Perfomance Evaluation* Vol.45, Issue 2-3, pp. 147–178, 2001.
3. F. Baccelli, G. Balbo, R.J. Boucherie, J. Campos, G. Chiola "Annotated bibliography on stochastic Petri nets". In *Performance Evaluation of Parallel and Distributed Systems — Solution Methods*, CWI Tract, 105, pp. 1–24, Amsterdam, 1994.
4. G. Balbo, S. Bruell, S. Ghanta, "Combining queueing networks and generalized stochastic Petri nets for the solution of complex models of system behaviour". *IEEE Transactions on Computers*, Vol. 37, pp. 1251–1268, 1988.
5. S. Balsamo, A. Di Marco, P.Inverardi, M. Simeoni "Software Architectures: State of the art and perspectives". Rapporto Tecnico CS-2003-01, Università *Ca' Foscari* di Venezia, 2003.
6. S. Balsamo, M. Bernardo, M. Simeoni "Combining Stochastic Process Algebras and Queueing Networks for Software Architecture Analysis". In [41], 2002.
7. S.Balsamo, G.Iazeolla "Product-Form Synthesis of Queueing Networks", *IEEE Transactions on Software Engineering*, vol.11, n.1, February 1985.
8. M. Bernardo, P. Ciancarini, L. Donatiello "ÆMPA: A Process Algebraic Description Language for the Performance Analysis of Software Architectures". In [40], pp. 1–11, 2000.
9. M. Bernardo, R. Gorrieri "A Tutorial on EMPA: A Theory of Concurrent Processes with Nondeterminism, Priorities, Probabilities and Time". *Theoretical Computer Science* Vol. 202, pp. 1–54, 1998.
10. S. Bernardi, S. Donatelli, J. Meseguer "From UML Sequence Diagrams and Statecharts to analysable Petri Net models". In [41], 2002.
11. G. Booch, J. Rumbaugh, I. Jacobson "The Unified Modeling Language User Guide". Addison Wesley, New York, 1999.

12. P. Buchholz "A framework for the hierarchical analysis of discrete event dynamic systems". Phd Thesis, University of Dortmund, 1996.
13. R.J.A. Buhr, R.S.Casselman "Use CASE Maps for Object-Oriented Systems", *Prentice Hall*, 1996.
14. P.S. Coe, F.W. Howell, R.N. Ibbett, L.M. Williams, "Technical Note: A Hierarchical Computer Architecture Design and Simulation Environment". *ACM Transactions on Modelling and Computer Simulation*, Vol. 8, No. 4, pp. 431–446, 1998.
15. V. Cortellessa, V. Grassi "A performance based methodology to early evaluate the effectiveness of mobile software architectures". *Journal of Logic and Algebraic Programming*, 2002.
16. V. Cortellessa, R. Mirandola "Deriving a Queueing Network based Performance Model from UML Diagrams". In [40], pp. 58–70, 2000.
17. M. De Miguel, T. Lambolais, M. Hannouz, S. Betgé-Brezetz, S. Piekarec "UML Extensions for the Specification and Evaluation of Latency Constraints in Architectural Models". In [40], pp. 83–88, 2000.
18. N. Götz, U. Herzog, M. Rettelback "TIPP – a language for timed processes and performance evaluation". *Technical Report* 4/92, IMMD7, University of Erlangen-Nürnberg, Germany, 1992.
19. V. Grassi, R. Mirandola "PRIMAmob-UML: A Methodology for Performance Analysis of Mobile Software Architectures". In [41], 2002.
20. G. Gu, Dorina C. Petriu "XSLT transformation from UML models to LQN performance models". In [41], 2002.
21. H. Hermanns, U. Herzog, J-P. Katoen "Process algebra for performance evaluation" Theoretical Computer Science, Vol. 274, pp. 43-87, 2002.
22. U. Herzog, J. Rolia "Performance Validation tools for software/hardware systems". *Perfomance Evaluation*, Vol. 45, Issue 2-3, pp. 125–146, 2001.
23. J. Hillston "A Compositional Approach to Performance Modelling". Cambridge University Press, 1996.
24. J. Hillston "A Compositional Approach to Performance Modelling". Cambridge University Press, Distinguished Dissertation Series, 1996.
25. J. Hillston, R. Pooley. "Stochastic Process Algebras and their application to Performance Modelling". Proceedings Tutorial, TOOLS'98, Palma de Mallorca, Spain 1998.
26. ITU - Telecommunication Standardization Sector, "Message Sequence Charts, ITU-T Recommentation Z.120(11/99)", 1999.
27. P. Kähkipuro "UML-based Performance Modeling Framework for Component-Based Distributed Systems". *Proc. of Performance Engineering*, Springer LNCS 2047, pp. 167–184, 2001.
28. K. Kant "Introduction to Computer System Performance Evaluation". *McGraw-Hill*, 1992.
29. P. King, R. Pooley "Derivation of Petri Net Performance Models from UML Specifications of Communication Software". *Proc. of the 11th International Conference on Tools and Techniques for Computer Performance Evaluation*, pp. 262–276, (2000).
30. J.G. Kemeny, J.L. Snell, "Finite Markov Chains". *Springer*, New York, 1976.
31. D. Menascè, H. Gooma "A Method for design and Performance Modeling of Client/Server Systems". *IEEE Trans. on Software Engineering*, Vol. 26, n. 11, pp. 1066–1085, 2000.
32. Object Management Group, "UML Profile, for Schedulability, Performance, and Time", OMG document ptc/2002-03-02 at `http://www.omg.org/cgi-bin/doc?ptc/2002-03-02`.

33. ObjecTime Ltd., Developer 5.1 Reference Manual, ObjecTime Ltd., Ottawa, Canada 1998.
34. OPNET Manuals, Mil 3, Inc., 1999.
35. Dorin C. Petriu, M. Woodside "Software Performance Models from System Scenarios in Use Case Maps". *Proc. of TOOLS02, Springer Verlag LNCS 2324*, pp. 141–1158, 2002.
36. Dorina C. Petriu, H. Shen "Applying UML Performance Profile: Graph Grammar-Based Derivation of LQN Models from UML Specifications". *Proc. of TOOLS02, Springer Verlag LNCS 2324*, pp. 159–177, 2002.
37. Dorina C. Petriu, X. Wang "From UML descriptions of High-Level Software Architectures to LQN Performance Models". *Proc. of AGTIVE'99, Springer Verlag LNCS 1779*, pp. 47–62, 1999.
38. R. Pooley "Using UML to Derive Stochastic Process Algebra Models". *Proc. of XV UK Performance Engineering Workshop*, 1999.
39. R. Pooley, P. King "The Unified Modeling Language and Performance Engineering". In *Proc. IEEE Software*, 1999.
40. Proceedings of the 2nd Int. Workshop on Software and Performance (WOSP 2000), ACM Press, Ottawa, Canada, 2000.
41. Proceedings of the 3rd Int. Workshop on Software and Performance (WOSP 2002), ACM Press, Rome, Italy, 2002.
42. J.A. Rolia and K.C. Sevcik "The Method of Layers". *IEEE Transaction on Software Engineering*, Vol. 21, n.8, pp. 682–688, 1995.
43. M. Shaw, D. Garlan "Software Architecture: Perspectives on an Emerging Discipline". Prentice Hall, 1996.
44. C. Smith "Performance Engineering of Software Systems". Addison-Wesley, Reading, MA, 1990.
45. M. Woodside, C. Hrischuk, B. Selic, S. Bayarov "Automated performance modeling of software generated by a design environment". *Performance Evaluation*, Vol. 45, Issue 2-3, pp. 107–123, 2001.
46. L.G. Williams, C.U. Smith "Performance Evaluation of Software Architectures". *Proc. of WOSP'98, Santa Fe, New Mexico, USA*, pp. 164–177, 1998.
47. "Unified Modeling Language (UML), version 1.4". *OMG Documentation*. At `http://www.omg.org/technology/documents/formal/uml.htm`

The Inevitable Pain of Software Development: Why There Is No Silver Bullet

Daniel M. Berry

School of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada
dberry@uwaterloo.ca
http://se.uwaterloo.ca/∼dberry

Abstract. A variety of programming accidents, i.e., models, methods, artifacts, and tools, are examined to determine that each has a step that programmers find very painful. Consequently, they habitually avoid or postpone the step. This pain is generally where the programming accident meets requirements, the essence of software, and their relentless volatility. Hence, there is no silver bullet.

1 Introduction

The call for papers (http://www.dsi.unive.it/∼mont2002/topics.html) for the 2002 Monterey Workshop on *Radical Innovations of Software and Systems Engineering in the Future* says of the object-oriented programming methods introduced in the last decade, "There is no proof and no evidence that software productivity has increased with the new methods"[1]. The call for papers argues that as a consequence of this and other reasons, there is an urgent need for new "post object-oriented" software engineering and programming techniques. However, there is no proof, evidence, or guarantee that any such new technique will increase productivity any more than object-oriented techniques have. Indeed, the past failures of any method to significantly increase productivity is the strongest predictor that any new technique will fare no better. In other words, what makes you, who designs new methods, think you can do better?

This paper tries to get to the root of why any given new programming technique has not improved productivity very much. This paper is, as the call for papers requires, an attempt "to analyze why some ideas were or were not successful" with the choice of "were not".

This paper is about building computer-based systems (CBS). Since the most flexible component of a CBS is its software, we often talk about developing its software, when in fact we are really developing the whole CBS. In this paper, "software" and "CBS" are used interchangeably.

This paper is based on personal observation. Sometimes, I describe an idea based solely on my own observations over the years. Such ideas carry no citation and have no

[1] Actually, this claim is a bit too strong. There is a feeling that object orientation has improved programming a bit, and there are even some data [33], but it is clearly not the silver bullet that it was hoped, and even hyped, to be.

M. Wirsing et al. (Eds.): RISSEF 2002, LNCS 2941, pp. 50–74, 2004.

formal or experimental basis. If your observations disagree, then please write your own rebuttal.

Sometimes, I give as a reason for doing or not doing something that should not be or should be done what amounts to a belief or feeling. This belief or feeling may be incorrect in the sense that it is not supported by the data. Whenever possible, I alert the reader of this mode by giving the belief-or-feeling-based sentence in italics.

2 Programming Then and Now

I learned to program in 1965. I can remember the first large program I wrote in 1966 outside the class room for a real-life problem. It was a program that implemented the external functionality of Operation Match, a computer-based dating and matchmaking service set up in the mid 1960s. I wrote it for my synagogue youth group in order that it could have a dance in which each person's date for the dance was that picked by a variation of the Operation Match software. The dance and the software were called "Operation Shadchan"[2]. I got a hold of the questionnaire for Operation Match, which was being used to match a new client of one gender with previously registered clients of the opposite gender. Each client filled out the form twice, once about him or herself and the second time about his or her ideal mate. For each new client, the data for the client would be entered. Then the software would find all sufficiently good matches with the new clients from among the previously registered clients. Presumably, the matches had to be both good and balanced; that is, the total number of questions for which each answered the way the other wanted had to be greater than a threshold and the difference between the number of matches in each direction had to be smaller than another threshold. I adapted this questionnaire for high school purposes. For example, I changed "Do you believe in sex on the first date?" to "Do you believe in kissing on the first date?"[3].

I then proceeded to write a program. I remember doing requirements analysis at the same time as I was doing the programming in the typical seat-of-the-pants build-it-and-fix-it-until-it-works method of those days:

- discover some requirements,
- code a little,
- discover more requirements,
- code a little more,
- etc, until the coding was done;
- test the whole thing,
- discover bugs or new requirements,
- code some more, etc.

The first requirements were fairly straightforward to identify. Since this matching was for a dance, unlike with Operation Match, each person would be matched with one and only one person of the opposite gender[4]. Obviously, we had to make sure that in the input set, the number of boys was equal to the number of girls, so that no one would

[2] "Shadchan" is Yiddish for "Matchmaker". The "ch" in "shadchan" is pronounced as the "X" in "TEX". One person attending the dance thought the name of the dance was "Operation Shotgun".

[3] Remember, this was during the mid 1960s!

[4] It was assumed that each person wanted someone of the opposite gender.

have the stigma of being unmatched by the software. The next requirements were not so easy to identify. Each boy and each girl should be matched to his or her best match. So, I wrote a loop that cycled through each person and for each, cycled through each other person of the opposite gender to find the best match. But whoa! what is a match? Ah, it cannot be just one way. It must be a mutually balanced match. But double whoa! I have to remove from the list of those that can be cycled through in either loop those that have been matched before. But triple whoa! Suppose the best mutual match for someone is among those not considered because that best mutual match has already been matched to someone else, and that earlier match is not as good. Worse than that, suppose that what is left to match with someone are absolute disasters for the someone. This simple algorithm is not so hot.

In those days and at that age, couples in which the girl was taller than the boy was a disaster, especially if the difference was big. Also, it was not considered so good if the girl of a couple were older than the boy. Therefore, to avoid being painted into a disastrous-match corner, I decided to search in a particular order, from hardest-to-find-non-disastrous matches to easiest. That is, I searched for matches for the tallest girls and shortest boys first and the shortest girls and tallest boys last. Presumably the tallest boys get assigned fairly early to the tallest girls and the shortest girls would get assigned fairly early to the shortest boys. I randomly chose the gender of the first person to be matched and alternated the gender in each iteration of the outer loop. To help avoid disastrous matches, I gave extra weight to the height and age questions in calculating the goodness of any potential match. Each "whoa" above represents a scrapping of previously written code in favor of new code based on the newly discovered requirements. Thus, I was discovering requirement flaws and correcting them during coding as a result of what I learned during coding.

The biggest problem I had was remembering all the requirements. It seemed that each thought brought about the discovery of more requirements, and these were piling up faster than I could modify the code to meet the requirements. I tried to write down requirements as I thought of them, but in the excitement of coding and tracking down the implications of a new requirement, which often included more requirements, I neglected to or forgot to write them all down, only to have to discover them again or to forget them entirely.

Basically, programming felt like skiing down a narrow downhill valley with an avalanche following me down the hill and gaining on me.

Nowadays, we follow more systematic methods. However, the basic feelings have not changed. Since then, I have maintained and enhanced a curve fitting application for chemists[5]. I built a payroll system. I have participated in the writing of a collection of text formatting software. I have watched my graduate students develop tools for requirements engineering. I watched my ex-wife build a simulation system. I have watched my ex-wife and former students and friends involved in startups building large CBSs. To me and, I am sure, the others, programming still feels like skiing with an avalanche following closely behind. I see all the others undergoing similar feelings and being as empathic as

[5] This was my second system, and it suffered Brooks's second system syndrome [18], as I tried to build a super-duper, all-inclusive, fancy whiz-bang general curve fitting application with all sorts of fantastic options.

I am, I get the same skiing feeling. No matter how much we try to be systematic and to document what we are doing, we forget to write things down, we overlook some things, and the discoveries seem to grow faster than the code.

The real problem of software engineering is dealing with ever-changing requirements. It appears that no model, method, artifact, or tool offered to date has succeeded to put a serious dent into this problem. I am not the first to say so. Fred Brooks and Michael Jackson, among others, have said the same for years. Let us examine their arguments.

3 The Search for a Silver Bullet

Some time ago, Fred Brooks, in saying that there is no software engineering silver bullet [17] [6], classified software issues into the essence and the accidents. The essence is what the software does and the accidents are the technology by which the software does the essence or by which the software is developed. That is, the requirements are the essence, while the language, tools, and methods used are the accidents. He went on to say, "The hardest single part of building a software system is deciding precisely what to build.... No other part of the work so cripples the resulting system if it is done wrong. No other part is more difficult to rectify later." This quotation captures the essential difficulty with software that must be addressed by any method that purports to alter fundamentally the way we program, that purports to make programming an order of magnitude easier, that purports to be the silver programming bullet we have all been looking for. Heretofore, no single method has put a dent into this essential problem, although all the discovered methods have combined to improve programming by at least an order of magnitude since 1968, the year the term "software engineering" was invented [59]. Bob Glass reviews the data showing the modest improvements of each of a variety of techniques, methods, and models [33].

Moreover, Brooks, based on his experience, predicted that "There is no single development, in either technology nor management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity." He made this prediction in 1986 when he first published "No Silver Bullet" in *Information Processing '86*. Since we are now well past a decade after 1986, and no such single technology or management technique has appeared, he has been proven correct. He added a slightly stronger, but still conditional, prediction with which I agree. "*I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation.* We still make syntax errors, to be sure; but they are fuzz compared to conceptual errors in most systems. If this is true, building software will always be hard. There is inherently no silver bullet." Because we will always be building systems at the new frontiers opened by the latest advances in the accidents, I believe that the antecedent of this conditional statement, that conceptual errors are the hardest kind to find, will always be true. Therefore, *I* believe that the conclusion will always be true and that there is inherently no silver bullet.

[6] This paper, published in *IEEE Computer in 1987* is a reprint of an earlier, less accessible publication in the 1986 *IFIP* proceedings [16], and is in turn reprinted in the currently more accessible 20th Annniversary Edition of *The Mythical Man-Month* [18].

Make no bones about it. Software productivity has improved since the publication of "No Silver Bullet" in 1986 because of the accumulative effect of all the accidental advances. Ironically, each such advance makes additional advances more and more difficult, because as each accidental difficulty is solved, what is left is more and more purely of essence.

The obvious question is "Why is there no silver bullet, and why can there not be a silver bullet?" The contention of this paper is that every time a new method that is intended to be a silver bullet is introduced, it does make many parts of the accidents easier. However, as soon as the method needs to deal with the essence or something affecting or affected by the essence, suddenly one part of the method becomes painful, distasteful, and difficult, so much so that this part of the method gets postponed, avoided, and skipped. Consequently, the method ends up being only slightly better than no method at all in dealing with essence-borne difficulties.

But, what *is* so difficult about understanding requirements? I mean, it should be possible to sit down with the customer and users, ask a few questions, understand the answers, and then synthesize a complete requirements specification. However, it never works out that way. Michael Jackson, Paul Clements, David Parnas, Meir Lehman, Bennet Lientz, Burton Swanson, and Laszlo Belady explain why.

4 Requirements Change

Michael Jackson, in his Keynote address at the 1994 International Conference on Requirements Engineering [42] said that two things are known about requirements:

1. They will change.
2. They will be misunderstood.

The first implies that a CBS will always have to be modified, to accommodate the changed requirements. Even more strongly, *there ain't no way that requirements are not gonna change*, and there is as much chance of stopping requirements change as there is stopping the growth of a fully functioning and heartily eating teenager. The second implies that a CBS will always have to be modified, to accommodate the changes necessitated by better understanding, as the misunderstandings are discovered. Clements and Parnas describe how difficult it is to understand everything that might be relevant [62].

Meir Lehman [50] classifies a system that solves a problem or implements an application in some real world domain as an E-type system. He points out that once installed, an E-type system becomes inextricably part of the application domain so that it ends up altering its own requirements.

Certainly, not all changes to a CBS are due to requirement changes, but the data show that a large majority of them are. Bennett Lientz and Burton Swanson found that of all maintenance of application software, 20% deal with correcting errors, and 80% deal with changing requirements. Of the requirement changes, 31% are to adapt the software to new platforms, and 69% are for perfecting the software, to improve its performance or to enhance its functionality [53].

Laszlo Belady and Meir Lehman observed the phenomenon of eventual unbounded growth of errors in legacy programs that were continually modified in an attempt to fix errors and enhance functionality [7,8]. That is, as programs undergo continual change their structure decays to the point that it is very hard to add something new or change

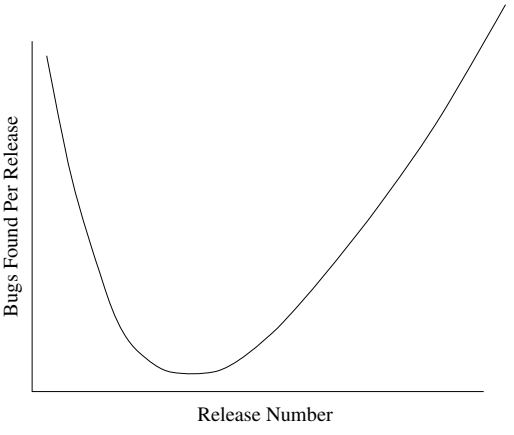


Fig. 1. Belady-Lehman Graph.

something already there without affecting seemingly unrelated parts of the program in a way that causes errors. It can be difficult even to find all the places that need to be modified. The programmers make poor guesses and the program, if it even runs, behaves in strange and unpredicted ways. They modeled the phenomenon mathematically and derived a graph like that of Fig. 1, showing the expected number of errors in a program as a function of time, as measured by the ordinal numbers of releases during which modifications are made. In practice, the curve is not as smooth as in the figure, and it is sometimes necessary to get very far into the upswing before being sure where the minimum point is. The minimum point represents the software at its most bug-free release. After this point, during what will be called the *Belady-Lehman (B-L) upswing*, the software's structure has so decayed that it is very difficult to change anything without adding more errors than have been fixed by the change.

The alternative to continuing on the B-L upswing for a CBS is to roll back to the best version, that is, the version that existed at the minimum point. Of course, rolling back assumes that all versions have been saved. All the bugs in this best version are declared to be features, and no changes are ever made in the CBS from then on. Usually not changing a CBS means that the CBS is dead, that no one is demanding changes because no one is using the software any more. However, some old faithful, mature, and reliable programs e.g, cat and other basic UNIX applications, vi, and ditroff[7], have gone this all-bugs-are-features route. The user community has grown to accept, and even, require that they will never change. If the remaining bugs of the best version are not acceptable features or the lack of certain new features begins to kill usage of the CBS, then a new CBS has to be developed from scratch to meet all old and new requirements, to eliminate bugs, and to restore a good structure to make future modifications possible. Another alternative that works in some special cases is to use the best version as a feature server for what it can do well and to build a new CBS that implements only the new and corrected features and has the feature server do the features of the best version of the old CBS.

[7] Well, at least *I* think so! One great thing about these programs that have not been modified since the 1980s is that their speed doubles every 18 months!

The time at which the minimum point comes and the slopes of the curves before and after the minimum point vary from development to development. The more complex the CBS is, the steeper the curve tends to be. Moreover, most of the time, for a carefully developed CBS, the minimum point tends to come in later releases of that CBS. However, occasionally, the minimum point is passed during the development of the first release, as a result of extensive requirements creep during that initial development. The requirements have changed so much since the initial commitment to architecture that the architecture has had to be changed so much that it is brittle. It has become very hard to accommodate new or changed requirements without breaking the CBS. Sometimes, the minimum point is passed during the initial development as a result of code being slapped together into the CBS with no sense of structure at all. The slightest requirements change breaks the CBS.

5 Purpose of Methods

One view of software development methods is that each method has as its underlying purpose to tame the B-L graph for the CBS developments to which it is applied. That is, each method tries to delay the beginning of the B-L upswing or to lower the slope of that B-L upswing or both. For example, Information Hiding [61] attempts to structure a system into modules such that each implementation change results in changing only the code, and not the interface, of only one module. If such a modularization can be found, then all the code affecting and affected by any implementation change is confined to one module. Thus, it is easier to do the modifications correctly and without adversely affecting other parts of the system. Hence, arrival at the minimum point is delayed and the slope of the upswing is reduced.

In this sense, each method, if followed religiously, works. Each method provides the programmer a way to manage complexity and change so as to delay and moderate the B-L upswing. However, each method has a catch, a fatal flaw, at least one step that is a real pain to do, that people put off. People put off this painful step in their haste to get the software done and shipped out or to do more interesting things, like write more new code. Consequently, the software tends to decay no matter what. The B-L upswing is inevitable.

What is the pain in Information Hiding? Its success in making future changes easy depends on having identified a right decomposition. If a new requirement comes along that causes changes that bridge several modules, these changes might very well be harder than if the code were more monolithic, simply because it is generally easier on tools to search within one, even big, module than in several, even small, modules [25,36]. Moreover, future changes, especially those interacting with the new requirement, will likely cross cut [45] module boundaries[8]. Consequently, it is really necessary to restructure the code into a different set of modules. This restructuring is a major pain, as it means moving code around, writing new code, and possibly throwing out old code for no exter-

[8] For an example that requires minimum explanation, consider two independently developed modules, for implementing unbounded precision *integer* and *real* arithmetic. Each can have its implementation change independently of the other. However, if the requirement of inter-type conversion is added, suddenly the two modules have to be programmed together with representations that allow conversion.

nally observable change in functionality. It is something that gets put off, causing more modifications to be made on an inappropriate structure.

The major irony is that the reason that the painful modifications are necessary is that the module structure no longer hides all information that should be hidden. Because of changes in requirements, there is some information scattered over several modules and exposed from each of these modules. Note that these changes are requirements changes rather than implementation changes, which continue to be effectively hidden. The painful modifications being avoided are those necessary to restore implementation information hiding, so that future implementation changes would be easier. Without the painful changes, all changes, both implementation and requirements-directed, will be painful. This same pain applies to any method based on Information Hiding, such as Object-Oriented Programming.

In the subsequent sections, each of a number of models, methods, and tools is examined to determine what its painful step is. In some cases, the whole model, method, or tool is the pain; in others, one particular step is the pain. No matter what, when people use these models, methods, and tools, they tend to get stopped by a painful step.

Despite the rhetoric, please understand that I am not opposed to any of the methods I am describing below. I have used some of them successfully. I am merely trying to show where each method becomes painful to use. A mature software engineer would continue to use them even when they are painful.

6 Development Models and Global Methods

This section considers several development models and general programming methods to identify their painful parts. The set chosen is only a sampling. Space limitations and reader boredom preclude covering more. It is hoped that after reading these, the reader is able to identify the inevitable pain in his or her own favorite model or method. In fact, for many models and methods, the pain lies in precisely the same or corresponding activities, all in response to requirements change. The models covered are the Waterfall Model and Requirements Engineering. The general methods covered are Structured Programming, Extreme Programming, Program Generation, Rapid Prototyping, and Formal Methods. A fuller set of models and programming models are covered in a technical report of the same title available at:
http://se.uwaterloo.ca/∼dberry/FTP_SITE/tech.reports/painpaper.pdf

6.1 Waterfall Model

The waterfall model [67] is an attempt to put discipline into the software development process by forcing understanding and documentation of the requirements before going on to design, by forcing understanding and documentation of design before going on to coding, by forcing thorough testing of the code while coding each module, etc. The model would work if the programmers could understand a stage thoroughly and document it fully before going on to the next stage [62]. However, understanding is difficult and elusive, and in particular, documentation is a pain. The typical programmer would prefer to get on to coding before documenting. Consequently, some programmers consider the whole model to be a pain, because it tries to force a disciplined way of working that

obstructs programmers from getting on with the coding in favor of providing seemingly endless, useless documentation. However, even for the programmers who do believe in discipline, the waterfall becomes a pain in any circumstance in which the waterfall cannot be followed, e.g., when the full requirements are learned only after significant implementation has been carried out or when there is significant repeated backtracking, when new requirements are continually discovered.

6.2 Structured Programming

I can recall the first systematic programming method I learned, used, and taught, from 1973 until it fell out of fashion in the mid 1980s, namely, Structured Programming or Stepwise Refinement [75,23]. After having done build-it-and-fix-it programming for years, Structured Programming appeared to be the silver bullet, at last, a way to put some order into the chaotic jumble of thoughts that characterized my programming, at last, a way to get way ahead of or to the side of the avalanche that was coming after me all the time.

In fact, I found that Structured Programming did work as promised for the development of the first version of any CBS. If I used the clues provided by the nouns that appeared in more than one high-level statement [9,11], Structured Programming did help me keep track of the effects of one part of the program on another. It did help me divide my concerns and conquer them one-by-one, without fear of forgetting a decision I had made, because these decisions were encoded in the lengthy, well-chosen names of the abstract, high-level statements that were yet to be refined. Best of all, the structured development itself was an ideal documentation of the structure of the program.

However, God help me if I needed to change something in a program developed by stepwise refinement, particularly if the change was due to an overlooked or changed requirement. I was faced with two choices:

1. Patch the change into the code in the traditional way after a careful analysis of ripple effects; observe that the documented structured development assists in finding the portions of the code affected by and affecting the change.
2. Redo the entire structured development from the top downward, taking into account the new requirement and the old requirements, all in the right places in the refinement.

The problems with the first choice are that:

1. no matter how careful I was, always some ripple effects were overlooked, and
2. patching destroys the relation between the structured development and the code, so that the former is no longer accurate documentation of the abstractions leading to the code. This discrepancy grows with each change, thus hastening the onset of the B-L upswing[9]. Thus, the new code contradicts the nice structure imposed by the structured development. Moreover, the names of the high level abstractions that get refined into code no longer imply their refinements.

[9] It is clear that there is a disrepancy, because if the abstractions had derived the new code, then the new code would have been there before.

Therefore, the correct alternative was the second, to start from scratch on a new structured development that recognizes the modified full set of requirements. However, then the likelihood is that the new code does not match the old code. Most structured programmers do this redevelopment with an eye to reusing as much as possible. That is, whenever one encounters a high-level statement with identical name and semantics as before, what it derived before is put there. However, the programmer must be careful to use the same high-level statements as refinements whenever possible, and he or she must be careful that in fact the same semantics is wanted as before and that he or she is not self deluding to the point of falsely believing a high-level statement has the same semantics as before.

This second alternative turned out to be so painful that I generally opted to the first alternative, patching the code and thus abandoned the protection, clarity, and documentation offered by Structured Programming.

About that time, I learned the value of faking it. Do the modification by patching up the code, and then go back and modify the original structured development to make it look like the new program was derived by Structured Programming. However, faking it was also painful, and soon, unless I was required to do so for external reasons, e.g., preparing a paper for publication or a contract, I quickly stopped even faking it. Adding to the pain of faking it was the certainty of not doing the faking perfectly and being caught. It was only later that David Parnas and Paul Clements legitimized faking it [62] and relieved my guilt.

6.3 Requirements Engineering

The basic premise of requirements engineering is spend sufficient time up front, before designing and coding, to anticipate all possible requirements and contingencies, so that design and coding consider all requirements and contingencies from the beginning [13,60]. Consequently, fewer changes are required, and the B-L upswing is both delayed and moderated. Data show that errors found during requirements analysis cost one order of magnitude less to fix than errors found during coding and two orders of magnitude less to fix than errors found during operation [15]. These economics stem from the very factors that cause the B-L upswing, namely, the fact that in a full running program, there is a lot more code affecting and affected by the changes necessary to fix an error than in its requirements specification. There are data and experiences that show that the more time spent in requirements engineering, the smoother the implementation steps are [31,24], not only in software engineering, but also in house building [14,73]. When the implementation goes smoother, it takes less time, it is more predictable, and there are fewer bugs.

However, for reasons that are not entirely clear to me[10], a confirmed requirements engineer, people seem to find haggling over requirements a royal pain. They would much rather move on to the coding, and they feel restless, bored, or even guilty when forced to spend more time on the requirements. A boss with his or her eyes focused on an unrealistically short deadline does not help in this respect. I would bet that Arnis Daugulis [24], his colleagues, and bosses at Latvenergo felt the pain of not being able

[10] Then again, I always read the manual for an appliance or piece of hardware or software completely before using the appliance or piece. I return the appliance or piece for a refund if there is *any* disagreement between what the manual says and the appliance or piece does, even if it is in only format of the screen during the set up procedure.

to move on to implementation, even though in the end, they were happy that they did not move on to implement the first two requirements specifications, which were, in retrospect, wrong.

The pain is exacerbated, and is felt even by those willing to haggle the requirements, because the requirements engineer must make people discover requirements by clairvoyance rather than by prototyping. The pain is increased even more as the backtracking of the waterfall model sets in, as new requirements continue to be discovered, even after it was thought that all requirements had been found.

There appear to be at least two ways of carrying out RE in advance of CBS design and implementation, what are called in the agile software development community [1] "Big Modeling Up Front (BMUF)" [3] and "Initial Requirements Up Front (IRUF)" [2]. They differ in the ways they treat the inevitable requirements changes.

In BMUF, the CBS developers try to create comprehensive requirement models for the whole system up front, and they try to specify the CBS completely before beginning its implementation. They try to get these models and specifications fully reviewed, validated, agreed to, and signed off by the customer and users. Basically, the developers try to get the requirements so well understood that they can be frozen, no matter how painful it is. However, as demonstrated in Section 4, the requirements continue to change as more and more is learned during implementation. To stem the tide of requirements change, the powers with vested interest in the freezing of the requirements create disincentives for changes, ranging from contractual penalties against the customer who demands changes to heavy bureaucracy, in the form of a change review board (CRB). For each proposed change, the CRB investigates the change's impact, economic and structural. For each proposed change, the CRB decides whether to reject or accept the change, and if it accepts the change, what penalties to exact. Of course, the CRB requires full documentation of all requirements models and specifications and up-to-date traceability among all these models and specifications[11].

Agile developers, on the other hand, expect to be gathering requirements throughout the entire CBS development. Thus, they say that we should "embrace change" [2]. We should explore the effects of a proposed change against the organizations business goals. If the change is worth it, do it, carrying out the required modifications, including restructuring. If the change isn't worth it, don't do it [40]. It's that simple.

The objective of agile processes is to carry out a CBS "development project that

1. focuses and delivers the essential system only, since anything more is extra cost and maintenance,
2. takes into account that the content of the essential system may change during the course of the project because of changes in business conditions,
3. allows the customer to frequently view working functionality, recommend changes, and have changes incorporated into the system as it is built, and
4. delivers business value at the price and cost defined as appropriate by the customer." [69]

[11] The agile development community says that traceability is a waste of resources. The cost of keeping trace data up to date must be balanced against the cost of calculating the trace data when they are needed to track down the ripple effects of a proposed change. The community believes that updating is so much more frequent than tracing that the total resources spent in continually updating outstrips the resources spent in occasional tracing.

Accordingly, agile developers get the IRUF, with all the stakeholders, i.e., customers, users, and developers, participating actively at all times, in which as many requirements as possible are gathered up front from all and only stakeholders. The goals of getting the IRUF are [2]

1. to identify the scope of the CBS being built,
2. to define high-level requirements for the CBS, and
3. to build consensus among stakeholders as to what the requirements imply.

The IRUF session ideally is as short as a few hours, but it can stretch into days and even weeks in less than ideal circumstances, such as not all stakeholders being in one place, or particularly tricky requirements. Then come several days of modeling sessions to produce a full set of models of the CBS as conceived by the IRUF. These models include use cases, which are eminently suitable for discussions between users and developers.

Following this modeling, the requirements are ranked by priority by all stakeholders. Business goals are taken into account during this ranking process, which typically requires about a day. Detailed modeling of any requirement takes place only during the beginning of the iteration during which it is decided to implement that requirement.

Notice that BMUF and IRUF differ in their approaches to dealing with the relentless, inevitable requirements changes. BMUF tries to anticipate all of them. IRUF does not; it just lets them come. BMUF is considered as not having succeeded totally if it fails to find a major requirement. The pain of dealing with the change with BMUF is felt during the RE process. IRUF practitioners embrace the change and decide on the basis of business value whether or not to do the change. The pain of dealing with the change with IRUF is felt in the re-implementation necessitated by the change, e.g., in the refactoring that can be necessary. See Section 6.4 for details about refactoring pain. Ultimately, neither approach can *prevent* a new requirement from appearing.

6.4 Extreme Programming

Extreme Programming (XP) [6] argues that the preplanning that is the cornerstone of each the various disciplined programming methods, such as the Waterfall model, documentation, requirements engineering, is a waste of time. This preplanning is a waste of time, because most likely, its results will be thrown out as new requirements are discovered. XP consists in simply building to the requirements that are understood at the time that programming commences. However, the requirements are given, not with a specification but with executable test cases. Unfortunately, these test cases are not always written because of the pain of writing test cases in general and in particular before it known what the code is supposed to do [58]. During this programming, a number of proven, minimum pain, and lightweight methods are applied to insure that the code that is produced meets the requirements and does so reliably. The methods include continual inspection, continual testing, and pair programming. Thus, the simplest architecture that is sufficient to deal with all the known requirements is used without too much consideration of possible changes in the future and making the architecture flexible enough to handle these changes. It is felt that too much consideration of the future is a waste, because the future requirements for which it plans for may never materialize and an

architecture based on these future requirements may be wrong for the requirements that do come up in the future.

What happens when a requirement comes along that does not fit in the existing architecture? XP says that the software should be refactored. Refactoring consists in stepping back, considering the new current full set of requirements and finding a new simplest architecture that fits the entire set. The code that has been written should be restructured to fit the new architecture, as if it were written with the new architecture from scratch. Doing so may require throwing code out and writing new code to do things that were already implemented. XP's rules say that refactoring should be done often. Because the code's structure is continually restructured to match its current needs, one avoids having to insert code that does not match the architecture. One avoids having to search widely for the code that affects and is affected by the changes. Thus, at all times, one is using a well-structured modularization that hides information well and that is suited to be modified without damaging the architecture. Thus, the B-L upswing is delayed and moderated.

However, refactoring, itself, is painful [28]. It means stopping the development process long enough to consider the full set of requirements and to design a new architecture. Furthermore, it may mean *throwing out perfectly good code whose only fault is that it no longer matches the architecture*, something that is very painful to the authors[12] of the code that is changed. Consequently, in the rush to get the next release out on time or early, refactoring is postponed and postponed, frequently to the point that it gets harder and harder. Also, the programmers realize that the new architecture obtained in any refactoring will prove to be wrong for some future new requirements. Ironically, the rationale for doing XP and not another method, is used as an excuse for not following a key step of extreme programming.

Indeed, Elssamadisy and Schalliol report on an attempt to apply a modified XP approach to a software development project that was larger than any previous project to which they had applied XP, with great success. They use the term "bad smells" to describe symptoms of poor practice that can derail XP from its usual swift track [28]. Interestingly, each of these bad smells has the look, feel, and odor of an inevitable pain.

1. In XP, to ensure that all customer requirements are met, a customer representative has to be present or at least available at all times and he or she must participate in test case construction. The test cases, of course, are written to test that the software meets the requirements. Elssamadisy and Schalliol report that over time, the customer representatives began to "refrain from that 'toil"' and began to rely on the test cases written by the developers. Consequently, the test cases ceased to be an accurate indication of the customer's requirements, and in the end, the product failed to meet the customer's requirements even though it was running the test cases perfectly. It appears to me that participation in the frequent test case construction was a pain to the customer that was caused by the relentless march of new requirements.
2. XP insists on the writing of test cases first and on delivery of running programs at the ends of frequent development iterations in an attempt to avoid the well-known horror of hard-and-slow-to-fix bug-ridden code that greets many software development teams at delivery time. However, as a deadline for an iteration approaches, and

[12] Remember that it's *pair* programming.

it is apparent that programming speed is not where it should be, managers begin to excuse developers from having to write test cases; developers refrain from just barely necessary refactorings; testers cut back on the thoroughness of test cases; and developers fail to follow GUI standards and to write documentation, all in the name of sticking to the iteration schedule. Speed is maintained, but at the cost of buggy code. It appears to me that all curtailed activities were painful in XP, just as they are in other methods.

3. Elssamadisy and Schalliol report that there was a routine of doing a particular function, called **u**nbooking and **r**ebooking of a **l**ease (URL). The first version of the routine did URL in the only way that was required by the first story. As a new story required a different specific case of URL, the required code was patched in to the routine as a special case. With each such new specific case of URL, the code became patchier and patchier; it became more and more apparent that a more and more painful refactoring was needed so that each specific case of URL is a specialization of a more generic, abstract URL. Elssamadisy and Schalliol

> were the ones who got stuck with fixing the problem. Also, because one of them is a big whiner, he kept complaining that "we knew this all along—something really stank for iterations and now I'm stuck with fixing it." The sad truth is that he was right. Every developer after the initial implementation knew that the code needed to be refactored, but for one reason or another ..., they never made the refactoring.
>
> This type of smell has no easy solution. The large refactoring had to be made because the inertia of the bad design was getting too high (footnote: Look ahead design would have been useful here also.) By taking the easy road from the beginning and completely ignoring the signs, we coded ourselves into a corner. So the moral of the story is this: when you find yourself making a large refactoring, stop and realize that it is probably because that [sic] you have skipped many smaller refactorings [28].

Is this ever a description of pain? It comes in a method which has been carefully designed to avoid many painful activities that programmers dislike.

The claim by agile developers is that the requirements to be addressed in the first iterations are those with the biggest business values. Any other requirements to be considered later are less important, and are even less likely to survive as originally conceived or even at all, due to the relentless change of requirements. Thus, refactoring becomes less and less important with each iteration.

However, this claim presupposes that the initial set of requirements, the IRUF, is so comprehensive that no new important requirements, forcing a major, painful refactoring, will ever be discovered. It's hard for me to imagine, and it runs counter to my experience that, any requirements modeling that is not the result of BMUF be so successful so as to effectively make refactoring completely unnecessary. Certainly, even if BMUF is done, it's hard not to ever need refactoring. Thus in the end, the potential for pain is there.

6.5 Program Generation and Domain-Specific Approaches

One approach to simplifying programming is to use a program generator to write a complete program from a declarative specification of the program. For example, one can use

a compiler-compiler to write most of a compiler from some syntax and semantic specifications for the language being compiled. These program generators do significantly simplify the process of writing a complete program. However, program generators are generally for application domains that are well enough understood that major requirement surprises that force refactoring of the system's archtecture, are unlikely. In more understood domains, programs are really more manufactured than they are programmed. In less understood domains, domain-specific approaches provide enough structure, a stable architecture that the domain has become a true engineering discipline, like aircraft, automobile, and bridge engineering. In these domains, each new instance is a perturbation of all previous instance. There are surprises during construction, and significant ones too, but these surprises are still rare enough to force a wholesale change the architecture of the system.

However, even program generators are not without pain. I recall that my third major software development project, for a summer job, was to develop a set of report programs operating on a single employee database. All the programs in the set were to be generated by RPG, the old Report Program Generator. To produce any program in the set, I specified the input data it needed in the database and the format and formulae for the output data, called the report. RPG would read this specification and write a program in assembly language which then had to be assembled. I was glad that I did not have to modify the generated program, which looked nothing like what a human being would write. To change a program, all I had to do was change its specification and to submit the new specification to RPG, which generated the new program. What could be simpler?

Nevertheless, after a while, after the number of related programs grew to a number, n, about 10, each change of data format, each new field, each deleted field, etc. became a nightmare as the ripple effects reverberated to all n programs. I felt like I was skiing ahead of an avalanche even though it was a summer job in an area noted for its 90s (90°F and 90% humidity) weather[13]. If I forgot to make a change or made the wrong change in one or more of the specifications, the next payroll would be a mess.

Basically, a program generator, and for that matter all schemes for automatic programming [5], just move programming back to earlier in the lifecyle to a specification that is at a higher level than the normal program. However, even a high-level specification, because it must be internally consistent, gives pain when it must be modified.

6.6 Rapid Prototyping

Rapid prototyping [4,21] is offered as a means to identify requirements, to try out various ideas, to do usability testing, etc. To the extent that it succeeds, it delays and moderates the B-L upswing. The pain is to throw the prototype out and to start anew when implementation starts. Often, in the face of an impending implementation deadline, instead, the prototype is used as a basis for the first implementation, thus ratifying poor design decisions that were made for expedience in putting together the prototype rapidly. The need to throw out the prototype and start all over with a systematic development is at least the need to refactor if one is following XP. When the prototype's architecture is used

[13] On the other hand, in those days, computer rooms were so fiercely airconditioned that I had to wear my skiing sweater in the machine room.

for the production software, the very first production version is brittle, hard to modify, and prone to breaking whenever a change is made.

6.7 Formal Methods

There are a number of formal program development methods, each of which starts with a formal specification of the CBS to be built [74,22,34,37,44]. Once a formal specification has been written it can be subjected to verification that it meets higher-level formally stated requirements, such as security criteria [43]. Any code written for the implementation can be verified as correct with respect to the specification [41,27]. Sometimes implementing code can be generated directly from the specification [5].

Some consider the mere writing of a formal specification a pain. Some consider writing such a specification to be fun, but consider all the verification that needs to be done to be a pain. Some consider this verification to be fun. However, my experience is that everyone considers the work dealing with changed requirements to be a pain. The specification has to be changed with the same difficulties as changing code, that is, of finding all other parts of the specification affecting or affected by the change at hand.

The worst of all is the necessary reverification. Since requirements are inherently global, formal statements about them and proofs about the formal statements are inherently global as well. Potentially every proof needs to be redone, because even though a theorem clearly still holds, the old proof may use changed formulae. Some have tried to build tools to automatically redo proofs in the face of changing specifications [56,12]. The pain is enough to drive one to adopt a lightweight method in which only the specification is written and compiler-level checks are done and no verification is done [10].

7 Specific Methods and Techniques

Besides models and methods for the overall process of software engineering, there are a large variety of specific methods and techniques each of which is for attacking one specific problem of software engineering. Even these small methods have their pains, exactly where they brush up against changes. Again, only a small sampling of the available methods are covered in the hope that they are a convincing, representative sample.

7.1 Inspection

The effectiveness of inspection [29] as a technique for finding errors has been documented widely [32]. When applied to early documents, such as requirements or design documents, inspection helps delay and moderate the B-L upswing. However, inspection is a double pain. First, the documents to be inspected must be produced, and we know that documentation itself is a pain. Second, inspection is one of these unpopular activities that are the first to be scrubbed when the deadlines are looming [32]. Roger Pressman quotes Machiavelli in dealing with the unpopularity of inspection.

> "*... some maladies, as doctors say, at the beginning are easy to cure but difficult to recognize ... but in the course of time ... become easy to recognize but difficult to cure.*" Indeed! But we just don't listen when it comes to software. Every shred of evidence indicates that formal technical reviews (for example, inspections) result in fewer production bugs, lower maintenance costs, and higher software success rates. Yet we're unwilling to plan the effort required to recognize bugs in their early stage, even though bugs found in the field cost as much as 200 times more to correct [64].

7.2 The Daily Build

In a situation in which lots of programmers are continually modifying relatively but not completely independent code modules in a single CBS to fix bugs, improve performance, add new features, etc., a common practice is the process called the *daily build* [55]. It works best when there is tool support for version control, including commands for checking a module out, checking a module in, and identifying potential conflicts between what is checked in and what is already in.

Each programmer gets a problem, perhaps a bug to fix, performance to improve, or a feature to add, and he or she plans his or her attack. When ready, he or she downloads the latest versions of each module that needs to be changed. He or she makes the changes, he or she tests and regression tests the changed modules, he or she tests and regression tests the whole changed system, and when he or she is satisfied that the changes will work, he or she checks the modules back in.

The fun comes when each day, whatever complete collection of modules is there is compiled into a running version of the system and is tested and regression tested. If the system passes all the test, everything is fine. If not, whoever made the last change has to fix the problems, perhaps in consultation with other programmers, particularly of the modules affected by his or her changes. Consequently, the incentive is for each programmer to work quickly and to verify that the version of each module that is there just before a check in is what was checked out. If not, he or she should certainly feel compelled to re-implement and test his or her changes on that latest version. If the changes made by others are truly independent, this reimplementation is straightforward, and consists of folding in the independent changes identified with the help of diff. Obviously, the faster he or she works after checking out the modules to be changed, the less likely he or she will find them changed by others at checkin. Note that two programmers can work at cross purposes. However, if they notice that they are working on the same modules, they are encouraged to work together to ensure that their changes do not conflict.

The testing, regression testing, the checking in, and possible rework are all real pains. Most people would just as soon dispense with them to get on to the real work, programming. However, in this case, the social and work costs of failing to do these activities is very high. Basically, the one whose checkin caused the build not to work correctly is the turkey of the day to be hung by his or her thumbs, and he or she has to spend considerable time to fix a version that has multiple programmers' changes.

7.3 Open Sourcing

Open sourcing [65] is a world-wide application of the daily build, compounded from daily to continually. The main advantage is that the software has a world-wide legion of

merciless inspectors and bug fixers. Whoever sees the source of a problem is encouraged by the promise of fame to fix it. "Given enough eyeballs, all bugs are shallow." The pain for each programmer is integrating his or her update in the face of almost continual change. The biggest pain is for the manager of the whole project, e.g., Linus Torvalds for Linux. Can you imagine what he has to go through to reconcile different fixes of a problem, to pick the one that he thinks fits best in a moving target. Compound this problem with the fact that he must deal with several problems at once. Goodness of fit is measured not only by how well it solves the problem but also by how consistent it is with all other changes that are being done. If the manager is not careful, the source could slide into B-L upswing oblivion very quickly.

8 Documentation

Almost all methods, including the general waterfall model, require programmers to document their decisions so that all decisions are visible to all persons working on the same CBS development, and even to the persons who make the decisions and later forget them. For example, Information Hiding requires that the interfaces and behaviors of the procedures and containing modules be documented so that users know what the modules are supposed to do without having to see the hidden code. When the documented information is available, the methods work as intended and the B-L upswing is both delayed and moderated. The kinds of documentation included are:

1. in the waterfall model, the requirements specification, the project plan, the design, the test plan, the comments, etc.,
2. in Structured Programming, the structured development itself,
3. in Information Hiding, besides the interfaces and behaviors, for each module, the secret that it hides; this secret is the encapsulated design information of the module [39],
4. in traceability schemes, the links between related units of the artifacts, which can be any other kind of documentation and the code itself [35], and
5. in Requirements Engineering, the requirements specification, the lexicon giving the problem vocabulary [52], the scenarios describing how the CBS is supposed to be used [71,20], and sometimes a network of issues, positions, and arguments for capturing rationale [19].

A number of authors, including S. D. Hester, D. L. Parnas, and D. F. Utter [39], have gone so far as to advocate using the documentation that one should write anyway as a design medium.

However, documentation itself is painful. It requires that people stop the analysis, design, or coding work that they are doing and to write their ideas down on the spot. If they do not stop to document, they forget the details, and if and when there is time later to document, a lot of the information that should have gone into the documentation has been forgotten. Often, *there is no time to document.* The shipping deadline is looming. So, the documentation never happens. If any of it happens, it quickly gets out of date as the program continues to change, and people continue to postpone recording decisions. Later, the programmers who must make changes have no information or poor, out-of-date

information about the requirements of, the architecture of, the design of, the rationale behind, the test plans for, and the commentary about the code. Since they do not know what documentation to trust, they trust none of it [26]. Therefore, they have a hard time finding the code and tests affecting and affected by the changes. The B-L upswing sets in earlier and is steeper.

Again, the irony is that the very process that the documentation helps is the process that undermines the validity of the documentation.

9 Tools and Environments

There are a number of environments of tools that help manage CBS development by keeping all artifacts on-line in an editable form, by keeping relationships between the artifacts, by performing a variety of consistency checks and analyses, and by doing other mundane tasks such as running regression tests.

The general pain of all these tools and environments is remembering to use them and keep their information up to date in the pressure of an impending deadline breathing down the developers' necks. For example, configuration management systems [30,70,66] depend on programmers' checking modules out for working on them and on programmers' checking modules back in when the programmers are finished with working on them. They depend on programmers' meeting when the system suggests that they have to meet. When a programmer fails to check a module out, check a module back in, or to meet with another, the system loses control and the completeness and consistency it promises cannot be guaranteed any more. These requirements on the programmers are really not much of a pain, but programmers do occasionally forget to do one step, again in the rush to beat the deadline. A more complete discussion of tools and environments is in a technical report of the same title available at:
http://se.uwaterloo.ca/∼dberry/FTP_SITE/tech.reports/painpaper.pdf

10 Conclusions

Thus, it appears that there is no software engineering silver bullet[14]. All software engineering bullets, even those that contain some silver, are made mostly of lead. It is too hard to purify the painful lead out of the real-life software engineering bullet to leave a pure painless silver software engineering bullet.

The situation with software engineering methods is not unlike that stubborn chest of drawers in the old slapstick movies; a shlimazel[15] pushes in one drawer and out pops another one, usually right smack dab on the poor shlimazel's knees or shins. If you find a new method that eliminates an old method's pain, the new method will be found to have its own source of pain.

[14] Some have called the feeling that there is no silver bullet pessimistic. However, I view the feeling as optimistic, because it says that software engineering can *never* be automated, that it will always require thinking, creative, human beings. Therefore, we programmers are always assured of jobs!

[15] "Shlimazel" is Yiddish for "one who always suffers bad luck".

There cannot be any significant change in programming until someone figures out how to deal, with a lot less pain, with the relentless change of requirements and all of its ripple effects. Perhaps, we have to accept that CBS development is an art and that no amount of systematization will make it less so.

Software engineering is an art, no less than painting. Indeed, the first part of the titles of Don Knuth's books in his series on algorithms for computer programming is *The Art of Computer Programming* [46,47,48]. The fact that software engineering is an art does not minimize its technical roots. A programmer must know the languages, the hardware platforms, the domain areas of his or her software, etc. However, what he or she does with them is very much a matter of art.

Even traditional arts, e.g., painting, have technical aspects. No matter how talented a painter is, if he or she does not know how to work with paints, his or her paintings will not be good. Furthermore, as has been demonstrated by mediocre artists, knowledge of the techniques does not insure good quality art.

Actually software engineering is an art just like mathematics. Both creativity and knowledge are required. Would mathematicians get upset if they were told that it was impossible to turn mathematics into a systematic engineering discipline? Would mathematicians even try?

If we know a domain well enough that architecture-changing requirement surprises are significantly tamed, as for compiler production these days, we can go the engineering route for that domain, to make building software for it as systematic as building a bridge or a building. However, for any new problem, where the excitement of major innovation is, there is no hope of avoiding relentless change as we learn about the domain, the need for artistry, and the pain.

The key concept in the Capability Maturity Model (CMM) [63] is *maturity*. Getting the capability to do the recommended practices is not a real problem. The real problem is getting the maturity to stick to the practices despite the real pain.

As some, including Alexander Egyed and Dino Mandrioli in private conversation, have observed, the psychology of computer programming [72] is coming to play here. Psychologically, we programmers tend to think coding as the real work and we feel disturbed by, pained by, and as wasting our time in, doing anything else, no matter how necessary the anything else is to producing quality software.

As Les Belandy and Jack Goldberg have observed in private communication, designers and builders of large, complex systems in other engineering disciplines, e.g., of aircraft, bridges, buildings, nuclear power plans, etc., do not complain that there is no magic silver bullet for their discipline. What is the difference? Well, the engineers in the other disciplines know that there is no silver bullet. So why do we software engineers search for a silver bullet?

The search for a software engineering silver bullet has been motivated by a belief that there *has* to be or at least there *ought* to be a software engineering silver bullet. After all, why bother looking for something that does not exist? The search is based on the assumption that programming can be made easy or at least systematic.

However, my own experience has been that programming is extremely difficult, of the order of difficulty of theorem proving[16], and requires much creativity as well as good ol' fashioned sweat of the brow! While certain aspects of programming, namely the production of code snippets that meet very well-understood requirements, i.e., the body of a procedure or loop not more than a few pages long, coming to grips with any other aspect of code, i.e., requirements, architecture, changes to requirements or architecture, that is, the essence, requires much creativity, blood, sweat, and tears. I am certainly not the only one to say so. Also Fred Brooks [17], Richard Gabriel [38], Don Knuth [49], and John Knight, in private communication, have said so. In particular, Knuth said,

> What were the lessons I learned from so many years of intensive work on the practical problem of setting type by computer? One of the most important lessons, perhaps, is the fact that SOFTWARE IS HARD.... From now on I shall have significantly greater respect for every successful software tool that I encounter. During the past decade I was surprised to learn that the writing of programs for TEX and for *METAFONT* proved to be much more difficult than all the other things I had done (like proving theorems or writing books). The creation of good software demands a significantly higher standard of accuracy than those other things do, and it requires a longer attention span than other intellectual tasks.

One referee suggests that the focus of this paper is on the interface between the essence and the accidents of software. Heretofore, software engineering's focus has been on one or the other. It is necessary to consider also the interface between them and of the feedback loop caused by that interface [51]. It is my belief that the interface between the essence and the accidents is of essential difficulty.

However, a more perplexing question is how we, in computer science and software engineering ever go into this search for a silver bullet business, which continues to this very day, as indicated by the continuing claims of how method *X* will improve your programming productivity by 100% or some such large number? How did we ever get into this believe that programming is easy or at least systematizable? Perhaps our fields' origins in Mathematics is responsible for this delusion in a perversely negative way. Some mathematicians, or least some that I have known, tend to look down at programming as not being as intellectually substantial as doing mathematics, as proving theorems, which is regarded as the ultimate in creativity. This belief has been the basis for denials of tenure to people who do only software, no matter how substantial, innovative, and creative it may be. This belief has been the basis for disowning of nascent software engineering programs. Many of us in software engineering have had to fight this shortsightedness. The irony is that while fighting this belief in others, we possibly have adopted it, as the basis for our belief that programming ought to be easy or at least systematizable.

Perhaps now it is clear why I no longer get excited over any new language, development model, method, tool, or environment that is supposed to improve programming[17].

[16] I am reminded of a paper by Manna and Waldinger showing that programming is equivalent to a constructive proof of the existence of code that is partially correct with respect to input-output assertions [54].

[17] One reviewer offered an alternative explanation, saying "It's because you are getting old!"

It is clear also why I think that the most important work is that addressing requirements, changes, and the psychology and sociology of programming.

Acknowledgements

I am grateful to Matt Armstrong, Les Belady, Barry Boehm, Fred Brooks, Alexander Egyed, Bob Glass, Jack Goldberg, John Knight, Andrew Malton, Dino Mandrioli, Daniel Morales Germán, John Mylopoulos, Torsten Nelson, Peter Neumann, David Parnas, James Robertson, Steve Schach, Davor Svetinović and the anonymous referees of previous versions of this paper for useful discussions about specific methods and other issues and careful and thoughtful comments about the paper, both favorable and not so favorable. I was supported in part by NSERC grant NSERC-RGPIN227055-00.

References

1. Principles: The Agile Alliance. The Agile Alliance (2001) http://www.agilealliance.org/
2. Ambler, S.W.: Agile Requirements Modeling. The Official Agile Modeling (AM) Site (2001) http://www.agilemodeling.com/essays/agileRequirementsModeling.htm
3. Ambler, S.W.: Agile Software Development. The Official Agile Modeling (AM) Site (2001) http://www.agilemodeling.com/essays/agileSoftwareDevelopment.htm
4. Andriole, S.J.: Fast Cheap Requirements: Prototype or Else!. IEEE Software **14**:2 (March 1994) 85–87
5. Balzer, R.M.: Transformational Implementation: An Example. IEEE Transactions on Software Engineering **SE-7**:1 (January 1981) 3–14
6. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley, Reading, MA (1999)
7. Belady, L.A.; Lehman, M.M.: A Model of Large Program Development. IBM Systems Journal **15**:3 (1976) 225–252
8. Belady, L.A.; Lehman, M.M.: Program System Dynamics or the Metadynamics of Systems in Maintenance and Growth. Lehman, M.M.; Belady, L.A. (eds.): Program Evolution, Academic Press, London, UK (1985)
9. Berry, D.M.: An Example of Structured Programming. UCLA Computer Science Department Quarterly **2**:3 (July 1974)
10. Berry, D.M.: The Application of the Formal Development Methodology to Data Base Design and Integrity Verification. UCLA Computer Science Department Quarterly **9**:4 (Fall 1981)
11. Berry, D.M.: Program Proofs Produced Practically. Tutorial Notes of Fifth International Conference on Software Engineering, San Diego, CA (March 1981)
12. Berry, D.M.: An Ina Jo Proof Manager for the Formal Development Method. Proceedings of Verkshop III, Software Engineering Notes (August 1985)
13. Berry, D.M.; Lawrence, B.: Requirements Engineering. IEEE Software **15**:2 (March 1998) 26–29
14. Berry, D.M.: Software and House Requirements Engineering: Lessons Learned in Combating Requirements Creep. Requirements Engineering Journal **3**:3 & 4 (1998) 242–244
15. Boehm, B.W.: Software Engineering Economics. Prentice-Hall, Englewood Cliffs, NJ (1981)
16. Brooks, F.P. Jr.: No Silver Bullet. Information Processing '86, Kugler, H.-J. (ed.) Elsevier Science Publishers B.V. (North-Holland), IFIP, (1986)
17. Brooks, F.P. Jr.: No Silver Bullet. Computer **20**:4 (April 1987) 10–19

18. Brooks, F.P. Jr.: The Mythical Man-Month: Essays on Software Engineering. Second Edition, Addison Wesley, Reading, MA (1995)
19. Burgess-Yakemovic, K.C.; Conklin, J.: Report on Development Project Use of an Issue-Based Information System. Proceedings of the ACM Conference on CSCW, Los Angeles, CA (October 1990)
20. Carroll, J.; Rosson, M.B.; Chin, G.; Koenemann, J.: Requirements Development in Scenario-Based Design. IEEE Transactions on Software Engineering **SE-24**:12 (December 1998) 1156–1170
21. CECOM: Requirements Engineering and Rapid Prototyping Workshop Proceedings. Eatontown, NJ, U.S. Army CECOM (1989)
22. Chan, W.; Anderson, R.J.; Beame, P.; Burns, S.; Modugno, F.; Notkin, D.; Reese, J.D.: Model Checking Large Software Specifications. IEEE Transactions on Software Engineering **SE-24**:7 (July 1998) 498–520
23. Dahl, O.-J.; Dijkstra, E.W.; Hoare, C.A.R.: Structured Programming. Academic Press, London, UK (1972)
24. Daugulis, A.: Time Aspects in Requirements Engineering: or 'Every Cloud has a Silver Lining'. Requirements Engineering **5**:3 (2000) 137–143
25. Dunsmore, A.; Roger, M.; Wood, M.: Object-Oriented Inspection in the Face of Delocalization. Proceedings of the Twenty-Second International Conference on Software Engineering (ICSE2000), Limerick, Ireland (June 2000) 467–476
26. Egyed, A.: A Scenario-Driven Approach to Traceability. Proceedings of the Twenty-Third International Conference on Software Engineering (ICSE2001), Toronto, ON, Canada (June 2001) 132–132
27. Elspas, B.; Levitt, K.N.; Waldinger, R.J.; Waksman, A.: An Assessment of Techniques for Proving Program Correctness. Computing Surveys **4**:2 (June 1972) 81–96
28. Elssamadisy, A.; Schalliol, G.: Recognizing and Responding to "Bad Smells" in Extreme Programming. Proceedings of the Twenty-Fourth International Conference on Software Engineering (ICSE2002), Orlando, FL (May 2001)
29. Fagan, M.E.: Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal **15**:3 (1976) 182–211
30. Feldman, S.I.: Make—A Program for Maintaining Computer Programs. Software—Practice and Experience **9**:4 (April 1979) 224–265
31. Forsberg, K.; Mooz, H.: System Engineering Overview. Software Requirements Engineering, Second Edition, Thayer, R.H.; Dorfman, M. (eds.): IEEE Computer Society, Washington (1997)
32. Gilb, T.; Graham, D.: Software Inspection. Addison Wesley, Wokingham, UK (1993)
33. Glass, R.L.: The Realities of Software Technology Payoffs. Communications of the ACM **42**:2 (February 1999) 74–79
34. Goguen, J.A.; Tardo, J.: An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications. Proceedings Conference on Specifications of Reliable Software, Boston (1979)
35. Gotel, O.C.Z.; Finkelstein, A.C.W.: An Analysis of the Requirements Traceability Problem. Proceedings of the First International Conference on Requirements Engineering, IEEE Computer Society, Colorado Springs, CO (1994) 94–101
36. Griswold, W.G.; Yuan, J.J.; Kato, Y.: Exploiting the Map Metaphor in a Tool for Software Evolution. Proceedings of the Twenty-Third International Conference on Software Engineering (ICSE2001), Toronto, ON, Canada (June 2001) 265–274
37. Hall, A.: Using Formal Methods to Develop an ATC Information System. IEEE Software **13**:2 (March 1996) 66–76
38. Heiss, J.J.: The Poetry of Programming. An Interview of Richard Gabriel, java.sun.com Report (3 December 2002) http://java.sun.com/features/2002/11/gabriel_qa.html

39. Hester, S.D.; Parnas, D.L.; Utter, D.F.: Using Documentation as a Software Design Medium. The Bell System Technical Journal **69**:8 (October 1981)
40. Highsmith, J.; Cockburn, A.: Agile Software Development: The Business of Innovation. IEEE Computer **34**:9 (September 2001) 120–122
41. Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. Communications of the ACM **12**:10 (October 1969) 576–580,585
42. Jackson, M.A.: The Role of Architecture in Requirements Engineering. Proceedings of the First International Conference on Requirements Engineering, IEEE Computer Society, Colorado Springs, CO (April 18–22 1994) 241
43. Kemmerer, R.A.: Formal Verification of the UCLA Security Kernel: Abstract Model, Mapping Functions, Theorem Generation, and Proofs. Ph.D. Dissertation, Computer Science Department, UCLA (1979)
44. Kemmerer, R.A.: Testing Formal Specifications to Detect Design Errors. IEEE Transactions on Software Engineering **SE-11**:1 (January 1985) 32–43
45. Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.V.; Loingtier, J.-M.; Irwin, H.: Aspect-Oriented Programming. Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Springer, Finland (1997)
46. Knuth, D.E.: The Art of Computer Programming: Fundamental Algorithms. Addison Wesley, Reading, MA (1969)
47. Knuth, D.E.: The Art of Computer Programming: Seminumerical Algorithms. Addison Wesley, Reading, MA (1971)
48. Knuth, D.E.: The Art of Computer Programming: Sorting and Searching. Addison Wesley, Reading, MA (1973)
49. Knuth, D.E.: Theory and Practice. Theoretical Computer Science **90**:1 (1991) 1–15
50. Lehman, M.M.: Programs, Life Cycles, and Laws of Software Evolution. Proceedings of the IEEE **68**:9 (September 1980) 1060–1076
51. Lehman, M.M.; Ramil, J.F: The Impact of Feedback in the Global Software Process Journal of Systems and Software **46**:2–3 (1999) 123–134
52. Leite, J.C.S.P.; Franco, A.P.M.: A Strategy for Conceptual Model Acquisition. Proceedings of the First IEEE International Symposium on Requirements Engineerin g, San Diego, CA (January 1993) 243–246
53. Lientz, B.P.; Swanson, E.B.: Characteristics of Application Software Maintenance. Communications of the ACM **21**:6 (June 1978) 466–481
54. Manna, Z.; Waldinger, R.J.: Toward Automatic Program Synthesis. Communications of the ACM **14**:3 (May 1971) 151–165
55. McConnell, S.: Daily Build and Smoke Test. IEEE Software **13**:4 (July/August 1996) 144–143
56. Moriconi, M.S.: A Designer/Verifier's Assistant. IEEE Transactions on Software Engineering **SE-5**:4 (July 1979) 387–401
57. Myers, G.J.: The Art of Software Testing. Wiley-Interscience, New York, NY (1979)
58. Müller, M.M.; Tichy, W.F.: Case Study: Extreme Programming in a University Environment. Proceedings of the Twenty-Third International Conference on Software Engineering (ICSE2001), Toronto, ON, Canada (June 2001) 537–544
59. Naur, P.; Randell, B.: Software Engineering: Report on a Conference Sponsored by the NATO Science Commission. Garmisch, Germany, 7–11 October 1968, Scientific Affairs Division, NATO, Brussels, Belgium (January 1969)
60. Nuseibeh, B.; Easterbrook, S.: Requirements Engineering: A Roadmap. Finkelstein, A. (ed.): The Future of Software Engineering 2000, ACM, Limerick, Ireland (June 2000)
61. Parnas, D.L.: On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM **15**:2 (December 1972) 1053–1058
62. Parnas, D.L.; Clements, P.C.: A Rational Design Process: How and Why to Fake It. IEEE Transactions on Software Engineering **SE-12**:2 (February 1986) 196–257

63. Paulk, M.C.; Curtis, B.; Chrissis, M.B.; Weber, C.V.: Key Practices of the Capability Maturity Model. Technical Report, CMU/SEI-93-TR-25, Software Engineering Institute (February 1993)
64. Pressman, R.: Software According to Niccolò Machiavelli. IEEE Software **12**:1 (January 1995) 101–102
65. Raymond, E.: Linux and Open-Source Success. IEEE Software **26**:1 (January/February 1999) 85–89
66. Rochkind, M.J.: The Source Code Control System. IEEE Transactions on Software Engineering **SE-1**:4 (December 1975) 364–370
67. Royce, W.W.: Managing the Development of Large Software Systems: Concepts and Techniques. Proceedings of WesCon (August 1970)
68. Schach, S.R.: Object-Oriented and Classical Software Engineering. Fifth Edition, McGraw-Hill, New York, NY (2002)
69. Schwaber, K.: Agile Processes — Emergence of Essential Systems. The Agile Alliance (2002) http://www.agilealliance.org/articles/
70. Tichy, W.: RCS—A System for Version Control. Software—Practice and Experience **15**:7 (July 1985) 637–654
71. Weidenhaupt, K.; Pohl, K.; Jarke, M.; Haumer, P.: Scenarios in System Development: Current Practice. IEEE Software **15**:2 (1998) 34–45
72. Weinberg, G.M.: The Psychology of Computer Programming.. van Nostrand Reinhold, New York, NY (1971)
73. Wieringa, R.: Software Requirements Engineering: the Need for Systems Engineering and Literacy. Requirements Engineering Journal **6**:2 (2001) 132–134
74. Wing, J.M.: A Specifier's Introduction to Formal Methods. IEEE Computer **23**:9 (September 1990) 8–24
75. Wirth, N.: Program Development by Stepwise Refinement. Communications of the ACM **14**:4 (April 1971) 221–227

Toward Component-Oriented Formal Software Development: An Algebraic Approach*

Michel Bidoit[1], Donald Sannella[2], and Andrzej Tarlecki[3]

[1] Laboratoire Spécification et Vérification, CNRS & ENS de Cachan, France
[2] Laboratory for Foundations of Computer Science, University of Edinburgh, UK
[3] Institute of Informatics, Warsaw University and Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland

Abstract. Component based design and development of software is one of the most challenging issues in software engineering. In this paper, we adopt a somewhat simplified view of software components and discuss how they can be conveniently modelled in a framework that provides a modular approach to formal software development by means of stepwise refinement. In particular we take into account an observational interpretation of requirements specifications and study its impact on the definition of the semantics of specifications of (parametrized) components. Our study is carried out in the context of CASL architectural specifications.

1 Introduction

Nowadays component based design is perceived as a key technology for developing systems in a cost- and time effective manner. However, there is still no clear understanding of what is a component, and in particular of how to provide a formal semantics of components. Similarly, formal software development has received relatively little attention in the context of component based approaches.

We focus here on a rather restrictive view of components, namely *software components* (understood as pieces of code) in contrast with *system components* (understood as self-contained computers with their own hardware and software interacting with each other and the environment by exchanging messages across linking interfaces). However, our view of (software) components is consistent with the best accepted definition in the software industry, see [Szy98]: a (software) component is a unit of composition with contractually specified interfaces and fully explicit context dependencies that can be deployed independently.

There has been a great deal of work in the algebraic specification tradition on formalizing the intuitive idea of software development by stepwise refinement, including [EKMP82,GM82,Gan83,Sch87,ST88b,ST89,Sch90]; the general ideas go back at least to [Hoa72]. For a recent survey, see [EK99]. There are many issues that make this a difficult problem, and some of them are rather subtle,

* This work has been partially supported by KBN grant 7T11C 002 21 and European AGILE project IST-2001-32747 (AT), CNRS–PAS Research Cooperation Programme (MB, AT), and British–Polish Research Partnership Programme (DS, AT).

M. Wirsing et al. (Eds.): RISSEF 2002, LNCS 2941, pp. 75–90, 2004.

one example being the relationship between specification structure and software structure. An overview that covers most of our own contributions is [ST97], with some more recent work addressing the problem of how to prove correctness of refinement steps [BH98], and the design of a convenient formalism for writing specifications [ABK+02,BST02a].

In this paper we discuss how software components can be modelled in an algebraic framework providing a modular approach to formal software development by means of stepwise refinement. In particular we take into account an observational interpretation of requirements specifications and study its impact on the definition of the semantics of specifications of (parametrized) components. Our study is carried out in the context of CASL architectural specifications. Architectural specifications, for describing the modular structure of software systems, are probably the most novel feature of CASL. We view them here as a means of making complex refinement steps, by defining well-structured constructions to be used to build a software system from implementations of individual components (these also include parametrized components, acting as constructions providing local construction steps to be used in a more global context).

We begin with a brief introduction to CASL basic and structured specifications in Sect. 2. Then we present our basic view of formal software development by means of stepwise refinement in Sect. 3, motivating CASL architectural specifications introduced in Sect. 4. In Sect. 5 we motivate and recall the observational interpretation of specifications, and we study in Sect. 6 the impact of this observational interpretation on the semantics of specifications of parametrized components. An example is given in Sect. 7 to illustrate a few of the main points. Further work is discussed in Sect. 8. The present paper is a high-level overview that concentrates on presenting and justifying the concepts without dwelling on the technicalities, which are presented in [BST02b] and elsewhere.

2 CASL Essentials

A basic assumption underpinning algebraic specification and derived approaches to software specification and development is that programs are modelled as algebras (of some kind) with their "types" captured by algebraic signatures (again, adapted as appropriate). Then specifications include axioms describing their required properties. This leads to quite a flexible framework, which can be tuned as desired to cope with various programming features of interest by selecting the appropriate variation of algebra, signature and axiom. This flexibility has been formalized via the notion of *institution* [GB92] and related work on the theory of specifications and formal program development [ST88a,ST97,BH93].

CASL is an algebraic specification language to describe CASL *models*: many-sorted algebras with subsorts, partial and total operations, and predicates. CASL models are classified by CASL *signatures*, which give *sort* names (with their subsorting relation), partial and total *operation* names, and *predicate* names, together with *profiles* of operations and predicates. In CASL models, subsorts and supersorts are linked by implicit *subsort embeddings* required to compose

with each other and to be compatible with operations and predicates with the same names. For each signature Σ, the class of all Σ-models is denoted $Mod(\Sigma)$.

The basic level of CASL includes *declarations* to introduce components of signatures and *axioms* to give properties that characterize *models* of a specification. The logic used to write the axioms is essentially first-order logic (so, with quantification and the usual logical connectives) built over *atomic formulae* which include strong and existential equalities, definedness formulae and predicate applications, with generation constraints added as special, non-first-order sentences. A basic CASL specification SP amounts to a definition of a signature Σ and a set of axioms Φ. It denotes the class $[\![SP]\!] \subseteq Mod(\Sigma)$ of SP-models, which are those Σ-models that *satisfy* all the axioms in Φ: $[\![SP]\!] = \{A \in Mod(\Sigma) \mid A \models \Phi\}$.

Apart from basic specifications as above, CASL provides ways of building complex specifications out of simpler ones by means of various *structuring constructs*. These include translation, hiding, union, and both free and loose forms of extension. *Generic specifications* and their *instantiations* with pushout-style semantics [BG80,EM85] are also provided. Structured specifications built using these constructs can be given a compositional semantics where each specification SP determines a signature $Sig[SP]$ and a class $[\![SP]\!] \subseteq Mod(Sig[SP])$ of models.

3 Program Development and Refinement

The intended use of CASL, as of any such specification formalism, is to specify programs. Each CASL specification should determine a class of programs that correctly realize the specified requirements. To fit this into the formal view of CASL specifications, programs must be written in a programming language having a semantics which assigns[1] to each program its *denotation* as a CASL model. Then each program P determines a signature $Sig[P]$ and a model $[\![P]\!] \in Mod(Sig[P])$. The denotation $[\![SP]\!]$ of a specification SP is a description of its admissible realizations: a program P is a *(correct) realization* of SP if $Sig[P] = Sig[SP]$ and $[\![P]\!] \in [\![SP]\!]$.

In an idealized view of program development, we start with an initial loose requirements specification SP_0 and refine it step by step until we have a specification SP_{last} that records all the intended design decisions:

$$SP_0 \rightsquigarrow SP_1 \rightsquigarrow \cdots \rightsquigarrow SP_{last}$$

Stepwise refinement only makes sense if the above chain of refinements guarantees that any correct realization of SP_{last} is also a correct realization of SP_0: for any P, if $[\![P]\!] \in [\![SP_{last}]\!]$ then $[\![P]\!] \in [\![SP_0]\!]$. This is ensured by the definition of refinement. For any SP and SP' with the same signature, we define:

$$SP \rightsquigarrow SP' \iff [\![SP']\!] \subseteq [\![SP]\!]$$

[1] This may be rather indirect, and in general involves a non-trivial abstraction step. It has not yet been attempted for any real programming language, but see [SM02] for an outline of how this could be done for Haskell. See also the pre-CASL work on Extended ML [KST97].

The construction of a program to realize SP_{last} is a separate task which strongly depends on the target programming language. If SP_{last} is relatively small and sufficiently detailed, then achieving this task tends to be straightforward for clean functional programming languages and problems that are appropriately coded in such languages, see for instance the example in Sect. 7. If the target programming language is C or a similar low-level language then of course there is a larger gap between specification and program, largely because a lot of work must be devoted to mundane and complex matters like management of heap space that are handled automatically by more modern languages. This step is outside the scope of CASL, which provides means for building specifications only; in this sense it is not a "wide-spectrum" language [BW82]. See [AS02] for a more detailed discussion. Furthermore, there is no construct in CASL to explicitly express refinement between specifications. All this is a part of the meta-level, though firmly based on the formal semantics of CASL specifications.

A more satisfactory model of refinement allows for modular decomposition of a given development task into several tasks by refining a specification to a sequence of specifications, each to be further refined independently. (Of course, a development may branch more than once, giving a tree structure.)

$$SP \rightsquigarrow BR \begin{cases} SP_1 \rightsquigarrow \cdots \rightsquigarrow SP_{1,last} \\ \vdots \\ SP_n \rightsquigarrow \cdots \rightsquigarrow SP_{n,last} \end{cases}$$

Given realizations $P_1, \ldots, P_n$ of the specifications $SP_{1,last}, \ldots, SP_{n,last}$, we should be able to put them together with no extra effort to obtain a realization of SP. So for each such branching point we need an operation to combine arbitrary realizations of $SP_1, \ldots, SP_n$ into a realization of SP. This may be thought of as a linking procedure $LINK_{BR}$ attached to the branching point BR, where for any $P_1, \ldots, P_n$ realizing $SP_1, \ldots, SP_n$, $LINK_{BR}(P_1, \ldots, P_n)$ realizes SP:

$$\text{if } [\![P_1]\!] \in [\![SP_1]\!], \ldots, [\![P_n]\!] \in [\![SP_n]\!] \text{ then } [\![LINK_{BR}(P_1, \ldots, P_n)]\!] \in [\![SP]\!]$$

Crucially, this means that whenever we want to replace a realization P_i of a component specification SP_i with a new realization P_i' (still of SP_i), all we need to do is to "re-link" it with realizations of the other component specifications, with no need to modify them in any way. $LINK_{BR}(P_1, \ldots, P_i', \ldots, P_n)$ is guaranteed to be a correct realization of SP, just as $LINK_{BR}(P_1, \ldots, P_i, \ldots, P_n)$ was. In other words, the only interaction between the components happens via $LINK_{BR}$, so the components may be developed entirely independently from each other.

The nature of $LINK_{BR}$ depends on the nature of the programs considered. They could be for instance just "program texts" in some programming language like Pascal (in which case $LINK_{BR}$ may be a simple textual operation, say, regrouping the declarations and definitions provided by the component programs) or actual pieces of compiled code (in which case $LINK_{BR}$ would really be linking in the usual sense of the word). Our preferred view is that the programming

language in use has reasonably powerful and flexible modularization facilities, such as those in Standard ML [Pau96] or Ada [Ada94]. Then $P_1, \ldots, P_n$ are program modules (structures in Standard ML, packages in Ada) and $LINK_{BR}$ is a module expression or a *generic module* with formal parameters for which the actual modules $P_1, \ldots, P_n$ may be substituted. Note that if we later replace a module P_i by P_i' as above, "recompilation" of $LINK_{BR}(P_1, \ldots, P_i', \ldots, P_n)$ might be required but in no case will it be necessary to modify the other modules.

One might expect that BR above is just a *specification-building operation* OP (or a specification construct expressible in CASL), and branching could be viewed as "ordinary" refinement $SP \rightsquigarrow OP(SP_1, \ldots, SP_n)$. Further refinement of $OP(SP_1, \ldots, SP_n)$ might then consist of separate refinements for $SP_1, \ldots, SP_n$ as above. This requires at least that OP is "monotonic" with respect to inclusion of model classes[2]. Then the following "refinement rule" is sound:

$$\frac{SP_1 \rightsquigarrow SP_1' \qquad \cdots \qquad SP_n \rightsquigarrow SP_n'}{OP(SP_1, \ldots, SP_n) \rightsquigarrow OP(SP_1', \ldots, SP_n')}$$

This view is indeed possible provided that the specification-building operation OP is *constructive* in the following sense: for any realizations $P_1, \ldots, P_n$ of $SP_1, \ldots, SP_n$, we must be able to construct a realization $LINK_{OP}(P_1, \ldots, P_n)$ of $OP(SP_1, \ldots, SP_n)$. In that case, $OP(SP_1, \ldots, SP_n)$ will be consistent whenever $SP_1, \ldots, SP_n$ are. However, simple examples show that some standard specification-building operations (like the union of specifications) do not have this property. It follows that refining SP to $OP(SP_1, \ldots, SP_n)$, where OP is an arbitrary specification-building operation, does not ensure that we can provide a realization of SP even when given realizations of $SP_1, \ldots, SP_n$. (See [HN94] for a different approach to this problem.)

Another problem with the refinement step $SP \rightsquigarrow OP(SP_1, \ldots, SP_n)$ is that it does not explicitly indicate that subsequent refinement is to proceed by independently refining each of $SP_1, \ldots, SP_n$, so preserving the structure imposed by the operation OP. The structure of the specification $OP(SP_1, \ldots, SP_n)$ in no way prescribes the structure of the final program. And this is necessarily so: *while preserving the structure of a specification throughout the ensuing development is convenient when it is natural to do so, refinements that break this structure must also be allowed.* Otherwise, at very early stages of the development process we would have to fix the final structure of the resulting program: any decision about structuring a specification would amount to a decision about the structure of the final program. This is hardly practical, as the aims of structuring specifications in the early development phases (and at the requirements engineering phase) are quite distinct from those of structuring final programs.

On the other hand, at certain stages of program development we need to fix the structure of the system under development: the design of the modular

[2] Most of the specification constructs of CASL and other specification languages are indeed monotonic. The few exceptions — like imposing the requirement of freeness — can be viewed as operations which add "constraints" to specifications rather than as fully-fledged specification-building operations, cf. data constraints in Clear [BG80].

structure of the system is often among the most important design decisions in the development process. In CASL, this is the role of *architectural specifications*, introduced in the next section.

4 Architectural Specifications

In CASL, an architectural specification *prescribes* a decomposition of the task of implementing a requirements specification into a number of subtasks to implement specifications of "modular components" (called *units*) of the system under development. The units may be parametrized or not. Another essential part of an architectural specification is a prescription of how the units, once developed, are to be put together using a few simple operators. Thus, an architectural specification may be thought of as a definition of a construction that implements a requirements specification in terms of a number of specified units to be developed subsequently.

For the sake of readability, we present here only a very simple version of CASL architectural specifications, with a limited (but representative) number of constructs, shaped after a somewhat less simplified fragment used in [SMT$^+$01].

Architectural specifications: $ASP ::=$ **arch spec** Dcl^* **result** T
An architectural specification consists of a list of unit declarations followed by a unit result term.

Unit declarations: $Dcl ::= U\colon SP \mid U\colon SP_1 \xrightarrow{\iota} SP_2$
A unit declaration introduces a unit name with its type, which is either a specification or a specification of a parametrized unit, determined by a specification of its parameter and its result, which extends the parameter via a signature morphism ι.

Unit terms: $T ::= U \mid U[T$ **fit** $\sigma] \mid T_1$ **and** T_2
A unit term is either a (non-parametrized) unit name, or a (parametrized) unit application with an argument that fits via a signature morphism σ, or an amalgamation of (non-parametrized) units.

The semantics of this CASL fragment can be defined following the same lines as for full CASL, see [CoFI03,SMT$^+$01,BST02b]. Let us just discuss here the semantics of specifications of parametrized units. Consider for instance the following simple architectural specification:

arch spec AS
  units $U_1 : SP_1$;
    $F\colon SP_1 \xrightarrow{\iota} SP_2$;
  result $F[U_1]$

To be well-formed, the specification $SP_1 \xrightarrow{\iota} SP_2$ of the parametrized unit F should be such that $[\![SP_2]\!]|_\iota \subseteq [\![SP_1]\!]$ [3]. Let Σ_1 and Σ_2 be the respective sig-

[3] Given a signature morphism $\iota\colon \Sigma_1 \to \Sigma_2$ and a Σ_2-model $M_2 \in Mod(\Sigma_2)$, $M_2|_\iota \in Mod(\Sigma_1)$ is the *reduct* of M_2 w.r.t. ι to a Σ_1-model defined in the usual way; this obviously extends further to classes of Σ_2-models.

natures of SP_1 and SP_2. To realize the specification $SP_1 \xrightarrow{\iota} SP_2$, we should provide a "program fragment" ΔP (i.e., a parametrized program, see [Gog84]) that extends any realization P_1 of SP_1 to a realization P_2 of SP_2, which we will write as $\Delta P(P_1)$. The basic semantic property required is that for all programs P_1 such that $[\![P_1]\!] \in [\![SP_1]\!]$, $\Delta P(P_1)$ is a program that extends P_1 and realizes SP_2 (semantically: $[\![\Delta P(P_1)]\!]\big|_\iota = [\![P_1]\!]$ and $[\![\Delta P(P_1)]\!] \in [\![SP_2]\!]$). This amounts to requiring ΔP to determine a partial function[4] $[\![\Delta P]\!]\colon Mod(\Sigma_1) \to ? Mod(\Sigma_2)$ that "preserves" its argument whenever it is defined, is defined on (at least) all models in $[\![SP_1]\!]$,[5] and yields a result in $[\![SP_2]\!]$ when applied to a model in $[\![SP_1]\!]$. This leads to the following definitions.

Definition 1. *Given a signature morphism $\iota\colon \Sigma_1 \to \Sigma_2$, a* local construction along ι *is a persistent partial function $F\colon Mod(\Sigma_1) \to ? Mod(\Sigma_2)$ (for each $A_1 \in dom(F)$, $F(A_1)\big|_\iota = A_1$). We write $Mod(\Sigma_1 \xrightarrow{\iota} \Sigma_2)$ for the class of all local constructions along ι.*

Definition 2. *A local construction F along $\iota\colon Sig(SP_1) \to Sig(SP_2)$ is* strictly correct w.r.t. SP_1 and SP_2 *if for all models $A_1 \in [\![SP_1]\!]$, $A_1 \in dom(F)$ and $F(A_1) \in [\![SP_2]\!]$. We write $[\![SP_1 \xrightarrow{\iota} SP_2]\!]$ for the class of all local constructions along ι that are strictly correct w.r.t. SP_1 and SP_2.*

The class $[\![SP_1 \xrightarrow{\iota} SP_2]\!]$ is empty if there is some model of SP_1 that cannot be extended to a model of SP_2; then we say that $SP_1 \xrightarrow{\iota} SP_2$ is *inconsistent*.

The crucial idea here is that while programs (closed software components) are represented as CASL models, parametrized programs (generic software components) are represented as persistent partial functions mapping models to models.

Instantiation of such generic modules is then simply function application. This may be further complicated in CASL by cutting the argument out of a larger "global" context via some fitting morphism, and putting the result back into that context. (Hence the terminology "*local* construction".) The latter involves the amalgamation construct, which puts together models (non-parametrized units) sharing common parts, as discussed in detail in [SMT+01].

Note that in the architectural specification AS above, although F is generic it is only instantiated once. Genericity is used here merely to separate the task of implementing SP_1 from the task of implementing the rest of SP_2. This may be stressed by making the generic unit anonymous, and treating U_1 as an *import* for the unit that implements SP_2. In CASL, this is written as follows:

arch spec AS′
  units $U_1 : SP_1$;
    $U_2 : SP_2$ **given** U_1;
  result U_2

Semantically, this is essentially equivalent to AS except that the generic unit remains anonymous and there is an explicit name for the result.

[4] As in CASL, $X \to ? Y$ denotes the set of partial functions from X to Y.

[5] Intuitively, $\Delta P(P_1)$ is "statically" well-formed as soon as P_1 has the right signature, but needs to be defined only for arguments that realize SP_1.

5 Observational Equivalence

So far, we have followed the usual interpretation for basic specifications given as sets of axioms over some signature, which is to require models of such a basic specification to satisfy all its axioms. However, in many practical examples this turns out to be overly restrictive. The point is that only a subset of the sorts in the signature of a specification are typically intended to be directly observable — the others are treated as internal, with properties of their elements made visible only via *observations*: terms producing a result of an observable sort, and predicates. Often there are models that do not satisfy the axioms "literally" but in which all observations nevertheless deliver the required results. This calls for a relaxation of the interpretation of specifications, as advocated in numerous "observational" or "behavioural" approaches, going back at least to [GGM76,Rei81]. Two approaches are possible:

- introduce an "internal" *observational indistinguishability* relation between elements in the carrier of each model, and re-interpret equality in the axioms as indistinguishability; or
- introduce an "external" *observational equivalence* relation on models over each signature, and re-interpret specifications by closing their class of models under such equivalence.

It turns out that under some acceptable technical conditions, the two approaches are closely related and coincide for most basic specifications [BHW95,BT96]. We follow the second approach here.

From now on, for the sake of simplicity, we will assume that the set of observable sorts is empty and so predicates are the only observations. Note that this is not really a restriction, since one can always treat a sort as observable by introducing an "equality predicate" on it.

Definition 3. *Consider a signature Σ. A* correspondence *between two Σ-models A, B, written $\rho: A \bowtie B$, is a relation $\rho \subseteq |A| \times |B|$ that is closed under the operations*[6] *and preserves and reflects the predicates*[7]*. Two models $A, B \in Mod(\Sigma)$ are* observationally equivalent*, written $A \equiv B$, if there exists a correspondence between them.*

This formulation is due to [Sch87] (cf. "simulations" in [Mil71] and "weak homomorphisms" in [Gin68]) and is equivalent to other standard ways of defining observational equivalence between algebras, where a special role is played by *observable equalities*, i.e., equalities between terms of observable sorts.

[6] That is, for $f: s_1 \times \ldots \times s_n \to s$ in Σ, $a_1 \in |A|_{s_1}, \ldots, a_n \in |A|_{s_n}$ and $b_1 \in |B|_{s_1}, \ldots, b_n \in |B|_{s_n}$, if $(a_1, b_1) \in \rho_{s_1}, \ldots, (a_n, b_n) \in \rho_{s_n}$ then $f_A(a_1, \ldots, a_n)$ is defined iff $f_B(b_1, \ldots, b_n) \in \rho_s$ is defined, and then $(f_A(a_1, \ldots, a_n), f_B(b_1, \ldots, b_n)) \in \rho_s$.

[7] That is, for $p: s_1 \times \ldots \times s_n$ in Σ, $a_1 \in |A|_{s_1}, \ldots, a_n \in |A|_{s_n}$ and $b_1 \in |B|_{s_1}, \ldots, b_n \in |B|_{s_n}$, if $(a_1, b_1) \in \rho_{s_1}, \ldots, (a_n, b_n) \in \rho_{s_n}$ then $p_A(a_1, \ldots, a_n) \iff p_B(b_1, \ldots, b_n)$.

For any specification SP with $Sig(SP) = \Sigma$, we define its *observational interpretation* by abstracting from the standard interpretation as follows:

$$Abs_{\equiv}(SP) = \{A \in Mod(\Sigma) \mid A \equiv B \text{ for some } B \in [\![SP]\!]\}.$$

6 Observational Interpretation of Architectural Specifications

The observational interpretation of specifications sketched in the previous section leads to a more liberal notion of refinement of specifications. Given two specifications SP and SP' with the same signature, we define:

$$SP \overset{\equiv}{\rightsquigarrow} SP' \iff [\![SP']\!] \subseteq Abs_{\equiv}(SP)$$

This observational refinement concept means that now we consider that a program P is a correct realization of a specification SP if it determines a $Sig(SP)$-model which is observationally equivalent to an SP-model, thus relaxing the requirements spelled out in Sect. 3.

The crucial issue is now to understand how to re-interpret the semantics of architectural specifications to take account of the observational interpretation of specifications. Surprisingly enough, there is not much to change in the semantics of architectural specifications, the essential modifications to be made concerning only the semantics of specifications of parametrized units.

The key insight is that we should require local constructions to satisfy a "stability" property, see [Sch87].

Definition 4. *A local construction F along $\iota\colon \Sigma_1 \to \Sigma_2$ is* stable *if it preserves observational equivalence of models, i.e., for any Σ_1-models A, B such that $A \equiv B$, if $A \in dom(F)$ then $B \in dom(F)$ and $F(A) \equiv F(B)$.*

While stability seems to be an obvious requirement to be imposed on local constructions, it turns out that it is not quite strong enough for our purposes. The reason is that when applying a local construction, the argument may be "cut out" of a larger global context where more observations are available. To restrict attention to conditions that will be both strong enough and essentially local to the local constructions involved, we define *local stability* as follows.

Definition 5. *A local construction F along $\iota\colon \Sigma_1 \to \Sigma_2$ is* locally stable *if for any Σ_1-models A, B and correspondence $\rho_1\colon A \bowtie B$, $A \in dom(F)$ if and only if $B \in dom(F)$ and moreover, if this is the case then there exists a correspondence $\rho_2\colon F(A) \bowtie F(B)$ that extends ρ_1 (i.e., $\rho_2\big|_{\iota} = \rho_1$).*

Obviously, local stability implies stability. However, it also implies that using the local construction in a global context as suggested above yields a stable construction at the global level as well.

We now have the necessary ingredients to provide a re-interpretation of specifications of parametrized units.

Definition 6. *A local construction F along $\iota: Sig(SP_1) \to Sig(SP_2)$ is* observationally correct w.r.t. SP_1 and SP_2 *if for every model $A_1 \in [\![SP_1]\!]$, $A_1 \in dom(F)$ and there exists a model $A_2 \in [\![SP_2]\!]$ and correspondence $\rho_2: A_2 \bowtie F(A_1)$ such that $\rho_2|_\iota$ is the identity. We write $Mod_{lc}(SP_1 \xrightarrow{\iota} SP_2)$ for the class of all locally stable constructions along ι that are observationally correct w.r.t. SP_1 and SP_2.*

This implies that $A_2 \equiv F(A_1)$ and $A_2|_\iota = A_1$, which is in general stronger than $F(A_1) \in Abs_{\equiv}(SP_2)$. It follows that if $F \in Mod_{lc}(SP_1 \xrightarrow{\iota} SP_2)$ then there is some $F' \in [\![SP_1 \xrightarrow{\iota} SP_2]\!]$ such that $dom(F') = dom(F)$ and for each $A_1 \in [\![SP_1]\!]$, $F'(A_1) \equiv F(A_1)$. But in general $[\![SP_1 \xrightarrow{\iota} SP_2]\!] \not\subseteq Mod_{lc}(SP_1 \xrightarrow{\iota} SP_2)$, as strictly correct local constructions need not be stable. Moreover, it may happen that there are no stable observationally correct constructions, even if there are strictly correct ones: that is, we may have $Mod_{lc}(SP_1 \xrightarrow{\iota} SP_2) = \emptyset$ even if $[\![SP_1 \xrightarrow{\iota} SP_2]\!] \neq \emptyset$. For a more detailed treatment of stability and of observational interpretation of architectural specifications, see [BST02b].

The conclusion here is that now parametrized program modules, i.e., generic software components, are modelled as *persistent locally stable partial functions*. It is important to understand that stable constructions are those that respect modularity in the software construction process. That is, such constructions can use the ingredients provided by their parameters, but they cannot take advantage of their particular internal properties. Thus, (local) stability is a directive for language design, rather than a condition to be checked on a case-by-case basis: in a language with good modularization facilities, all constructions that one can code should be locally stable.

7 Example

The following example illustrates some of the points in the previous sections. The notation of CASL is hopefully understandable without further explanation; otherwise see [ABK+02].

We start with a simple specification of sets of strings.

spec STRING = **sort** *String* ...
spec STRINGSET = STRING
  then sort *Set*
    ops *empty* : *Set*;
      add : *String* × *Set* → *Set*
    pred *present* : *String* × *Set*
    ∀ s, s' : *String*, t : *Set*
      • $\neg present(s, empty)$
      • $present(s, add(s, t))$
      • $s \neq s' \Rightarrow (present(s, put(s', t)) \iff present(s, t))$

We now refine this specification to introduce the idea of using a hash table implementation of sets.

spec INT = **sort** *Int* ...
spec ELEM = **sort** *Elem*
spec ARRAY[ELEM] = ELEM **and** INT
  then sort *Array*[*Elem*]
    ops $empty : Array[Elem]$;
      $put : Int \times Elem \times Array[Elem] \to Array[Elem]$;
      $get : Int \times Array[Elem] \ \to ?\ Elem$
    pred $used : Int \times Array[Elem]$
    $\forall\ i, j : Int; e, e' : Elem; a : Array[Elem]$
      • $i \neq j \implies put(i, e', put(j, e, a)) = put(j, e, put(i, e', a))$
      • $put(i, e', put(i, e, a)) = put(i, e', a)$
      • $\neg used(i, empty)$
      • $used(i, put(i, e, a))$
      • $i \neq j \implies (\ used(i, put(j, e, a)) \iff used(i, a)\ )$
      • $get(i, put(i, e, a)) = e$
spec ELEM_KEY = ELEM **and** INT
  then op $hash : Elem \to Int$
spec HASHTABLE[ELEM_KEY] = ELEM_KEY **and** ARRAY[ELEM]
  then ops $add : Elem \times Array[Elem] \to Array[Elem]$;
    $putnear : Int \times Elem \times Array[Elem] \to Array[Elem]$
    preds $present : Elem \times Array[Elem]$
      $isnear : Int \times Elem \times Array[Elem]$
    $\forall\ i : Int; e : Elem; a : Array[Elem]$
      • $add(e, a) = putnear(hash(e), e, a)$
      • $\neg used(i, a) \implies putnear(i, e, a) = put(i, e, a)$
      • $used(i, a) \wedge get(i, a) = e \implies putnear(i, e, a) = a$
      • $used(i, a) \wedge get(i, a) \neq e \implies$
        $putnear(i, e, a) = putnear(succ(i), e, a)$
      • $present(e, a) \iff isnear(hash(e), e, a)$
      • $\neg used(i, a) \implies \neg isnear(i, e, a)$
      • $used(i, a) \wedge get(i, a) = e \implies isnear(i, e, a)$
      • $used(i, a) \wedge get(i, a) \neq e \implies$
        $(isnear(i, e, a) \iff isnear(succ(i), e, a))$
spec STRINGKEY = STRING **and** INT
  then op $hash : String \to Int$
spec STRINGHASHTABLE =
  HASHTABLE[STRINGKEY] **with** $Array[String] \mapsto Set$
    reveal *String*, *Set*, *empty*, *add*, *present*

It is easy to check that STRINGHASHTABLE is a refinement of STRINGSET. This is a fairly natural structure, building on a specification of arrays that is presumably already available, and including a generic specification of hash tables that may be reused in future.

However, the structure of this specification does not prescribe the structure of the final implementation! For example, we may decide to adopt the structure given by the following architectural specification:

arch spec StringHashTableDesign =
  units N : Int;
    S : String;
    SK : StringKey **given** S, N;
    A : Elem $\to$ Array[Elem] **given** N;
    HT : StringHashTable
      given $\{A[SK]$ **with** $Array[String] \mapsto Set\}$
  result HT **reveal** $String, Set, empty, add, present$

The structure here differs in an essential way from the earlier one since we have chosen to forego genericity of hash tables, implementing them for the special case of strings.

Further development might lead to a final implementation in Standard ML, including the following modules. The task of extracting Standard ML signatures (`ARRAY_SIG` etc.) from the corresponding Casl signatures of the specifications given above is left for the reader.

```
functor A(E : ELEM_SIG) : ARRAY_SIG =
  struct
    open E
    type array = int -> elem
    exception unused
    fun empty(i) = raise unused
    fun put(i,e,a)(j) = if i=j then e else a(j)
    fun get(i,a) = a(i)
    fun used(i,a) = (a(i); true) handle unused => false
  end

structure HT : STRING_HASH_TABLE_SIG =
  struct
    open SK
    structure ASK = A(struct type elem=string end); open ASK
    type set = array
    fun putnear(i,s,t) =
       if used(i,t)
       then if get(i,t)=s then t else putnear(i+1,s,t)
       else put(i,s,t)
    fun add(s,t) = putnear(hash(s),s,t)
    fun isnear(i,s,t) =
       used(i,t) andalso (get(i,t)=s orelse isnear(i+1,s,t))
    fun present(s,t) = isnear(hash(s),s,t)
  end
```

The functor `A` is strictly correct with respect to Elem and Array[Elem], and the structure `HT` satisfies the axioms of HashTable[StringKey] literally (at least on the reachable part, and assuming the use of extensional equality on

functions). The former would not hold if, for instance, we implemented arrays so as to store the history of updates:

```
functor A'(E : ELEM_SIG) : ARRAY_SIG =
  struct
    open E
    type array = int -> elem list
    fun empty(i) = nil
    fun put(i,e,a)(j) = if i=j then e::a(j) else a(j)
    fun get(i,a) = let val e::_=a(i) in e end
    fun used(i,a) = not(null a(i))
  end
```

Then `A'` is not strictly correct with respect to ELEM and ARRAY[ELEM] (it violates the axiom $put(i, e', put(i, e, a)) = put(i, e', a)$) but it is observationally correct. Similarly we might change the code for `HT` to count the number of insertions of each string. This would violate the axiom $used(i, a) \wedge get(i, a) = s \Longrightarrow putnear(i, s, a) = a$, but again would be correct under an observational interpretation.

The Standard ML functors above are locally stable: they respect encapsulation since they do not use any properties of their arguments other than what is spelled out in their parameter signatures. Indeed, it is impossible to express non-stable functors in Standard ML.

Now let us try to instead directly implement the structure expressed by the specification STRINGHASHTABLE. That structure may be captured by the following architectural specification:

arch spec STRINGHASHTABLEDESIGN′ =
  units N : INT;
    A : ELEM → ARRAY[ELEM] **given** N;
    HT' : ELEM_KEY × ARRAY[ELEM] → HASHTABLE[ELEM_KEY];
    S : STRING;
    SK : STRINGKEY **given** S, N;
  result $HT'[SK][A[S]]$ **reveal** *String*, *Set*, *empty*, *add*, *present*

Then we might try

```
functor HT'
    (structure EK : ELEM_KEY_SIG and A : ARRAY_ELEM_KEY_SIG
         sharing type EK.elem=A.elem) : HASH_TABLE_ELEM_KEY_SIG =
  struct
    open EK A
    fun putnear(i,e,a) =
       if used(i,a)
       then if get(i,a)=e then a else putnear(i+1,e,a)
       else put(i,e,a)
    fun add(e,a) = putnear(hash(e),e,a)
```

```
    fun isnear(i,e,a) =
       used(i,a) andalso (get(i,a)=e orelse isnear(i+1,e,a))
    fun present(e,a) = isnear(hash(e),e,a)
  end
```

However, this does not define a locally stable functor, and is in fact not correct code in Standard ML, since it requires equality on `elem` (in `get(i,a)=e`) which is not required by `ELEM_KEY_SIG`. There is no locally stable functor satisfying the required specification. So, what is a reasonable structure for the requirements specification, as expressed in StringHashTable, turned out to be inappropriate as a modular design.

There is of course more than one way of changing the structure to make it appropriate; one was provided above in StringHashTableDesign. Another would be to require equality on *Elem* in Elem_Key, by introducing an equality predicate — this corresponds to making `elem` an "eqtype" in `ELEM_KEY_SIG`. One point of architectural specifications is that such change of structure is an important design decision that deserves to be recorded explicitly.

8 Conclusions and Further Work

In this paper, we have recalled how the standard and quite general view of formal software development by stepwise refinement can be refined to take into account some notion of components, leading to what is called architectural specifications in Casl. An important outcome of this view is the interpretation of software components by means of local constructions.

In a second step, we have taken into account the fact that a program need not be a strictly correct realization of the given requirements specification: it need only be observationally correct. Then we have pointed out how observational interpretation of specifications leads to the key — and quite natural — stability requirement on local constructions, and how this leads to a re-interpretation of software components by means of persistent locally stable partial functions.

A challenging issue is now to understand how far the concepts developed for our somewhat simplified view of software components can be inspiring for a more general view of components, in particular for the case of system components implemented as communicating processes. While this is clearly beyond the scope of this paper, we nevertheless imagine that a promising direction of future research would be to look for an adequate counterpart of (local) stability in this more general setting.

References

[Ada94] *Ada Reference Manual: Language and Standard Libraries*, version 6.0. International standard ISO/IEC 8652:1995(E). `http://www.adahome.com/rm95/` (1994).

[AS02] D. Aspinall and D. Sannella. From specifications to code in CASL. *Proc. 9th Intl. Conf. on Algebraic Methodology and Software Technology, AMAST'02*. Springer LNCS 2422, 1–14 (2002).

[ABK+02] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P.D. Mosses, D. Sannella and A. Tarlecki. CASL: The Common Algebraic Specification Language. *Theoretical Computer Science* 286:153–196 (2002).

[AKBK99] E. Astesiano, B. Krieg-Brückner and H.-J. Kreowski, eds. *Algebraic Foundations of Systems Specification*. Springer (1999).

[BW82] F. Bauer and H. Wössner. *Algorithmic Language and Program Development*. Springer (1982).

[BH93] M. Bidoit and R. Hennicker. A general framework for modular implementations of modular systems. *Proc. 4th Int. Conf. on Theory and Practice of Software Development TAPSOFT'93*, Springer LNCS 668, 199–214 (1993).

[BH98] M. Bidoit and R. Hennicker. Modular correctness proofs of behavioural implementations. *Acta Informatica* 35(11):951–1005 (1998).

[BHW95] M. Bidoit, R. Hennicker and M. Wirsing. Behavioural and abstractor specifications. *Science of Computer Programming* 25:149–186 (1995).

[BST02a] M. Bidoit, D. Sannella and A. Tarlecki. Architectural specifications in CASL. *Formal Aspects of Computing*, 13:252–273 (2002).

[BST02b] M. Bidoit, D. Sannella and A. Tarlecki. Global development via local observational construction steps. *Proc. 27th Intl. Symp. on Mathematical Foundations of Computer Science, MFCS'02*. Springer LNCS 2420, 1–24 (2002).

[BT96] M. Bidoit and A. Tarlecki. Behavioural satisfaction and equivalence in concrete model categories. *Proc. 20th Coll. on Trees in Algebra and Computing CAAP'96*, Linköping, Springer LNCS 1059, 241–256 (1996).

[BG80] R. Burstall and J. Goguen. The semantics of Clear, a specification language. *Proc. Advanced Course on Abstract Software Specifications*, Copenhagen. Springer LNCS 86, 292–332 (1980).

[CoFI03] The CoFI Task Group on Semantics. Semantics of the Common Algebraic Specification Language CASL. Available from `http://www.cofi.info/` (2003).

[EK99] H. Ehrig and H.-J. Kreowski. Refinement and implementation. In: [AKBK99], 201–242.

[EKMP82] H. Ehrig, H.-J. Kreowski, B. Mahr and P. Padawitz. Algebraic implementation of abstract data types. *Theoretical Computer Science* 20:209–263 (1982).

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I: Equations and Initial Semantics*. Springer (1985).

[Gan83] H. Ganzinger. Parameterized specifications: parameter passing and implementation with respect to observability. *ACM Transactions on Programming Languages and Systems* 5:318–354 (1983).

[GGM76] V. Giarratana, F. Gimona and U. Montanari. Observability concepts in abstract data type specifications. *Proc. 5th Intl. Symp. on Mathematical Foundations of Computer Science*, Springer LNCS 45, 576–587 (1976).

[Gin68] A. Ginzburg. *Algebraic Theory of Automata*. Academic Press (1968).

[Gog84] J. Goguen. Parameterized programming. *IEEE Trans. on Software Engineering* SE-10(5):528–543 (1984).

[GB92] J. Goguen and R. Burstall. Institutions: abstract model theory for specification and programming. *Journal of the ACM* 39:95–146 (1992).

[GM82] J. Goguen and J. Meseguer. Universal realization, persistent interconnection and implementation of abstract modules. *Proc. 9th Intl. Coll. on Automata, Languages and Programming*. Springer LNCS 140, 265–281 (1982).

[HN94] R. Hennicker and F. Nickl. A behavioural algebraic framework for modular system design and reuse. *Selected Papers from the 9th Workshop on Specification of Abstract Data Types*, Caldes de Malavella. Springer LNCS 785, 220–234 (1994).

[Hoa72] C.A.R. Hoare. Proofs of correctness of data representations. *Acta Informatica* 1:271–281 (1972).

[KST97] S. Kahrs, D. Sannella and A. Tarlecki. The definition of Extended ML: a gentle introduction. *Theoretical Comp. Sci.* 173:445–484 (1997).

[Mil71] R. Milner. An algebraic definition of simulation between programs. *Proc. 2nd Intl. Joint Conf. on Artificial Intelligence*, London, 481–489 (1971).

[Pau96] L. Paulson. *ML for the Working Programmer*, 2nd edition. Cambridge Univ. Press (1996).

[Rei81] H. Reichel. Behavioural equivalence — a unifying concept for initial and final specification methods. *Proc. 3rd Hungarian Comp. Sci. Conference*, 27–39 (1981).

[ST88a] D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Computation* 76:165–210 (1988).

[ST88b] D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: implementations revisited. *Acta Informatica* 25:233–281 (1988).

[ST89] D. Sannella and A. Tarlecki. Toward formal development of ML programs: foundations and methodology. *Proc. Colloq. on Current Issues in Programming Languages, Intl. Joint Conf. on Theory and Practice of Software Development TAPSOFT'89*, Barcelona. Springer LNCS 352, 375–389 (1989).

[ST97] D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing* 9:229–269 (1997).

[Sch87] O. Schoett. *Data Abstraction and the Correctness of Modular Programming*. Ph.D. thesis, report CST-42-87, Dept. of Computer Science, Univ. of Edinburgh (1987).

[Sch90] O. Schoett. Behavioural correctness of data representations. *Science of Computer Programming* 14:43–57 (1990).

[SM02] L. Schröder and T. Mossakowski. HasCasl: Towards integrated specification and development of functional programs. *Proc. 9th Intl. Conf. on Algebraic Methodology and Software Technology, AMAST'02*. Springer LNCS 2422, 99–116 (2002).

[SMT+01] L. Schröder, T. Mossakowski, A. Tarlecki, P. Hoffman and B. Klin. Semantics of architectural specifications in Casl. *Proc. 4th Intl. Conf. Fundamental Approaches to Software Engineering FASE'01*, Genova. Springer LNCS 2029, 253–268 (2001).

[Szy98] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New-York, N.Y. (1998).

Higher Order Applicative XML Documents

Peter T. Breuer, Carlos Delgado Kloos,
Vicente Luque Centeno, and Luis Fernández Sánchez

Depto. de Ingeniería Telemática, Universidad Carlos III de Madrid
Avda. Universidad, 30, E-28911 Leganés, Madrid, Spain

Abstract. XML is the extensible mark-up language designed to describe data on the world wide web. The syntax and semantics of a higher order applicative extension of XML is described here. In HOAX, the tag constructions <*f*>*a*</*f*> of standard XML documents denote the application of a function *f* to its arguments *a*.

HOAX provides a natural "higher order XSLT", that is, a higher order functional language which manipulates XML documents.

The objective of HOAX is to make transformation and automatic generation of XML documents more like declarative programming, thereby extending into the programming domain the same clarity and power of expression that has distinguished XML as a datatype language.

1 Introduction

Data modelling is always the first step in program development. The signature of the abstract data types is designed prior to the definition of their semantics or the operational behaviour of the program. Data modelling thus sets the basis for program structure. Several notations have been used for it, including UML and other well known formalisms. XML was defined a few years ago in order to facilitate precisely that task, namely the definition of deep data structures, together with their syntactic representation as flat text. It is of great importance on the Internet, where it forms the basis of the next generation approach beyond HTML. Browsers now interpret XML as well as the traditional HTML. Indeed XML-capable browsers interpret HTML because HTML can be defined as an XML document type, one of an unbounded number of such types that may be constructed in XML.

At present, we are performing a number of experiments to integrate into XML a behavioural component. In this paper, we present one experiment in this direction: attaching a functional semantics to XML tags allows us to give two interpretations to XML documents: the first, the "initial algebra" interpretation as a description of free data types and their instances, and a second, which represents the result of a manipulation done to the document.

M. Wirsing et al. (Eds.): RISSEF 2002, LNCS 2941, pp. 91–107, 2004.

The observation taken as the starting point for this article is that

1. Standard XML ***documents can be read as the applications of functions to their arguments***

where an XML tag is read as a function and the XML elements between the tags are read as the arguments. That is to say,

$$<f>\, a \,</f>$$

can be read as the application of the function f to the arguments a.

This principle says something about the kind of functions that must be allowed. Some must take (sequences of) XML document elements as arguments and produce an XML document element or elements as result. There is also a function associated with an ordinary XML tag, and it is a data constructor – it constructs an XML tree from its XML branches. Other kinds of functions will construct different data from their arguments. In this article the question of how to add in higher order functions within the general framework of XML will be approached. Adding it gives rise to a "**H**igher **O**rder **A**pplicative **X**ML", HOAX. XML documents can then be subjected to any required analysis either in-line or by referencing them as elements within function tags from a HOAX document.

XML and Functional Programming have often been associated in the literature. Most of the related work tends to add XML support to some well-known FP languages, such as the HaXML [4] compiler and tools written in Haskell, the XMLambda [3] XML-based FP language, and the XML support [5] in Erlang. However, nothing is seemingly being done in the other direction – adding functional programming capabilities to XML. An exception, perhaps, may be XSLT itself, which recent papers (see for example [1]) claim can be considered as a full functional programming language, instead of not quite being part of the family of FP languages because of its lack of support for higher order functions.

Is there really much point to extending XML to a full higher order functional language, rather than embedding XML in an existing FP language? The result will not be very human-readable, but it will be very easy for machines to manipulate. Such an extension to XML can improve the current usage in a number of ways: commonly, for example, applications are designed around a certain DTD or XML schema that defines the format for interchange and storage of documents in a given application domain. The application then provides the semantics. The documentation syntax alone cannot define all the constraints that should be imposed on the data or the use of it made in practice. But using HOAX functions to create "views" of an underlying XML document has exactly the effect of adding algebraic constraints to the otherwise free datatypes of XML. HOAX allows semantic constraints to be documented along with the syntactic constraints.

A second area that HOAX lends support to is the conversion of XML documents from one format to another, replacing the use of XSLT stylesheets for a given XML application. HOAX certainly binds the programming functionality closer to the programming data specification than in XSLT, but is also more

```
Doc ::= Abstract fn
      | Atomic src
      | Apply tag docs

where fn :: Doc* → Doc*,  src :: String,  tag :: Name,  docs :: Doc*
```

Fig. 1. HOAX documents are interpreted either as functions (abstractions, templates), or basic (XML) types, or are constructed from simpler elements by the application of a tag surround to its (document) parameters, as in XML.

declarative in style, and importantly, HOAX is strongly typed, where XSLT is not.

In this article, the semantics and the capabilities of HOAX will be the focus of attention. The functional semantics set out here is well adjusted to XML, because its data structures are XML documents and its data types are parsers of XML documents. To say that an XML document is of a certain type is to say that it may be parsed correctly by the parser associated with the type.

A second idea of importance in this article is that

2. Functions are documents that take parameters

Document templates, in other words. This principle delimits the kind of object that is considered to be a function. A function is not something that rings the telephone in the middle of the night, for example.

It also says that there are at least two types of documents: the everyday sort which can be read and written just as they are, and those which need to have extra information fed into them first. Their parameters are other documents too, since, in the context of HOAX, a third basic idea applies:

3. Everything is a document

In XML, it is usual to distinguish clearly between documents and the elements within documents. They lie in different syntactic categories in XML, but in HOAX that is not the case. This has no implications for XML itself; HOAX simply allows certain declarative constructs, which XML restricts to the distinguished root document, to be bound with interior document elements too. So there is no distinguished root.

The idea may be viewed as a first-class citizenship charter for documents; there is no discrimination about where a document may or may not be found in HOAX. If, as a result, HOAX allows a combination of XML elements which is not allowed in XML, the combination is simply "not XML". HOAX extends XML, and XML documents are HOAX documents, but not necessarily vice-versa. That is fine, because HOAX documents eventually evaluate to XML documents when used in practice, just as complex numbers multiply together to give real numbers when used as part of a practical calculation.

All HOAX documents are interpreted in one of the semantic categories in the abstract data type shown in Figure 1. This sets out the *semantic domain* of HOAX. There are three kinds of semantic entities listed there: atomic, abstract, and synthetic documents, respectively. The table below illustrates the three kinds

in the three rows, and gives in the first column their concrete syntax and in the second column gives their interpreted semantics, an abstract syntax. The square brackets ("[...]") that appear in the second column are list delimiters, and the operator "$\ldots\frown\ldots$" denotes concatenation of lists. "x" is a variable of list type:

document	example text	interpretation
atomic	`"a"`	`Atomic "a"`
abstract	`<var name="x"> "a" x </var>`	`Abstract`$(\lambda x.([$`Atomic "a"`$]\frown x))$
synthetic	`<t> "a" "b" </t>`	`Apply t [Atomic "a", Atomic "b"]`

The first example shows a simple document made by writing a single string. Its interpretation is as the atomic object `Atomic "a"`, a record of kind `Atomic`, with a single field, a string.

The second example shows a declaration of pending context surrounding two elements, one of them being the (possibly plural) context `x`. This is a document template, or function. The interpretation is as a record of kind `Abstract`, with a single field, a function $\lambda x.([$`Atomic "a"`$]\frown x)$. The function is that which prefixes the interpretation of "a" (the first example) to a generic list x supplied as an argument to the function.

The third example shows a standard XML tag written around two strings. This is a synthetic document. The interpretation is as a record of kind `Apply`, with two fields. The first field is the XML tag and the second field is the list of (interpretations of the) elements within the tag.

One of the technical difficulties in designing the formal semantics for an applicative language based on XML is that many elements of the language have to be interpreted as lists even though they do not "look" like lists. That is because there is no clear distinction in XML between the way many types of list may be used, and the way a singleton of that type may be used. The simplest technical approach to this equivalence is to always promote elements to lists in the interpretation. That is why the second example above treats "a" as being interpreted as the singleton list of the interpretation for it given in the first example.

On the technical side, this paper also defines a formal semantics for HOAX. It does so by defining types for the language which are also its parsers, or interpreters. Then the interpretation of a text is obtained by applying the type of the text to the text. This is a novel approach that succeeds quickly in proving that the interpretation is unambiguous (because the parse can be shown to be unique) and also satisfies certain correctness criteria, importantly that the interpretation of a HOAX document is obtained by substituting its variable references by the documents they in turn reference. This means that the meaning of documents is independent of the context of their use, an important consideration for a language based around the World Wide Web.

The HOAX syntax illustrated in the first column of the table is an *extension* of XML. That is, XML texts are also HOAX texts. Therefore HOAX texts look just like XML texts that have been "marked up" with additional "meta-annotations" –, the HOAX primitives.

```
<schema>                                                          } types
 <element name="example" type="anyType"/>                         }
 <element name="double" type="anyType">                           }
  <result type="anyType"/>                                        }
 </element>                                                       }
</schema>

<example>

  <def name="double" var="x">                                     } defns
    <val> x x </val>                                              }
    <double> <double> "goodbye" </double> </double>               } text
  </def>

</example>
```

Fig. 2. A simple HOAX document, showing types, definition, and text "sections".

The layout of this article is as follows. Section 2 defines the syntax of HOAX documents. Section 3 goes on to describe HOAX types and semantics. It is provable that there is only one parse of a document, i.e. that the HOAX interpretation is well-defined. It is also provable that the HOAX interpretation of functional application is substitution of the argument document in the places where it is referenced. This is a correctness result and it guarantees that documents always mean what they are supposed to, despite the flexible nature of the XML paradigm in which they are expressed, and independently of their locale, which is important in terms of the world wide web. Section 4 provides further examples and discussions related to practical improvements in HOAX.

2 Document Layouts

A simple HOAX document consists of the following sections:

1. a header section stating the types of the HOAX function tags defined in the rest of the document;
2. a section giving the applicative code associated with those function tags;
3. a section of XML-like text in which functions are applied to their arguments via tags and document references, resulting notionally in a transformed text.

This picture is not complete, because both type definitions and function definitions as used in HOAX may have local scope instead of global, but it is a useful first approximation. In this section, the concrete grammar of HOAX grammatical elements is discussed.

Figure 2 shows an example of a complete HOAX document. It consists of a set of type headers, followed by a function definition, followed by a text area in which the function is applied. The XML type representing the document (the "root" element) is `example`. It is constructed by `<example>` tags surrounding a sequence of arbitrary XML elements. Its type declaration is found within the XML `<schema>` tags just above the point of use.

In the example, the whole document is read by generating the text contents within the `<example>` tags by the application of the `<double>` function tags twice to the string `"goodbye"` (XML type `string`).

Note that the type of the `double` function is declared via a HOAX schema pattern, which is an extension of the XML type declaration schema. It differs from an ordinary XML schema in having one extra sub-element, `<result>`, binding a type (and optionally a maximum repeat number). It is the type returned by the function. The types of the arguments to the function are still given by the ordinary XML "type" attribute of the HOAX element.

The body of the function shows that it takes the argument sequence and duplicates it. The HOAX document shown applies the function twice. Therefore it is equivalent to the simple XML document below that mentions the word `"goodbye"` $2^2 = 4$ times:

```
<schema>
  <element name="example" type="anyType"/>
</schema>
<example>
    "goodbye" "goodbye" "goodbye" "goodbye"
</example>
```

This valid XML content is generated on the fly by HOAX interpreters.

In the rest of this section, the language in each of the "type", "definition", and "text" sections of a HOAX document is explained.

2.1 Type Declaration Section

Type declarations for function tags (which appear later in the document) are implemented via DTD- or XML schema-like constructs. The type of a function tag like `double` must be declared before its first use, using a format that extends standard XML DTD or schema declarations. If the schema format is used, an extra `<result>` sub-element declares the return type of the function.

```
<element name="double" type="anyType">
  <result type="anyType"/>
</element>
```

Note that:

1. arbitrary valid HOAX types are allowed, instead of just XML types;
2. there is an extra field in the element for the return type of a function, allowing function types to be expressed;
3. type declarations may precede any tag construct, and the scope of the declaration is the sequence of following text until the end of the current scope;
4. type declarations may also declare non-functions.

```
<schema>
  <element name=f type=τ1 resultType=τ2/>
  <element name=x type=τ1/>
  ...
</schema>
<t>
  <!--
     scope in which <f> ... </f> is mentionable
     and in which x is taken to have type τ1
    -->
</t>

where t, f::name, τ1, τ2::type
```

Fig. 3. Localised schema elements are used for type declarations.

In the case of the `double` example, `double` is a global and the type declaration for it covers the whole file. The general pattern of type declarations is shown in Figure 3, and they may be applied before any XML element. An abbreviated form may be used in which the attribute `resultType` replaces the `result` sub-element and its `type` attribute:

```
<element name="double" type="anyType" resultType="anyType"/>
```

The following is the declaration of a non-function:

```
<element name="x" type="anyType"/>
```

It declares that the symbol x is to be taken as having type `anyType`*.

The same symbol may be used as a tag, and may also be used "stand alone". It should always be declared, however.

Note that complex types are not allowed to be written in-line in element declarations in XML – only simple strings or numbers are allowed in the fields within tags. Instead, complex types must be written within a type construction in the body of the element declaration. We allow in-line forms as a convenience in HOAX. Thus:

```
<element name="double" type="anyType">
    <result> anyType </result>
</element>
```

is a strictly conforming declaration for the `double` function. The verbosity of the strict form makes it difficult to use in good conscience in the examples here.

2.2 Definition and Text Sections

HOAX documents generally reduce to XML documents. That is to say, the objective in writing them as HOAX documents is to achieve a different way of

obtaining an XML document. HOAX documents express transformation rules that may be applied to texts, and then apply them to a given text. The relationship of HOAX to XML is like the relation of complex numbers to real numbers. A real number (XML text) may be defined as the product of two complex numbers (HOAX definitions applied to HOAX text). Likewise, an XML text may be the result of applying HOAX transformations to HOAX texts.

HOAX syntax can be defined via a standard XML schema (see Figure 8). HOAX works like mark-up on XML documents. It introduces four special tags and associated grammatical constructs over and above the ordinary XML constructs, the first of which is the <*def*> tag:

1. <*def*> for binding a name to a value;
2. <*var*> for expressing abstraction;
3. <*eval*> for expressing the application of a function to its list of arguments;
4. <*ref*> for dereferencing a variable.

plus the ordinary XML tag mechanisms. Each of the above four constructs will be treated in turn. The basic idea is that definition tags bind a name to a value in a given expression. The other three tag constructs are supporting mechanisms.

Definitions. The special tag <*def*> is used inside a document to define a new tag with functional semantics. It binds a name to a value in an expression, e.g.

```
<def name="f" val="double"> <f> "a" "b" </f> </def>
```

declares the name `f` to have scope from the <*def* `name="f" val="double"`> tag to the corresponding close tag, </*def*>. And the value assigned to `f` in the example is that of the already defined function `double`, which writes its single argument out twice. The above is the functional programming construct:

```
let f = double in f ["a","b"]
```

Normally the scope is limited to a particular element of the HOAX document. A special case is a definition with global scope found at the head of a document.

Multiple simultaneous mutually recursive definitions are also allowed:

```
<def name1=sum name2=map name3=count>
  <val1> ...  </val1>
  <val2> ...  </val2>
  <val3> ...  </val3>
...
</def>
```

and the names and the bodies must correspond in numerical count. The bodies may reference the names being bound.

XML lore says that an attribute $a = b$ inside a tag t is equivalent to an embedded tag <a> b </a>. When the value of the field is not an atomic type (a string or a number, etc.), then a tag form should be used. The generic equivalence is shown in Figure 4. The construction in the paragraph above may also be written <*def* `name1=sum val1=...name2=map val2=...`> if the values are simple.

$$\frac{\texttt{<}a\ \ldots\ b\texttt{=}c\texttt{>}\ d\ \texttt{</}a\texttt{>}}{\texttt{<}a\ \ldots\texttt{>}\ \texttt{<}b\texttt{>}\ c\ \texttt{</}b\texttt{>}\ d\ \texttt{</}a\texttt{>}}$$

Fig. 4. Element attributes may be treated as contents via this equivalence.

$$\frac{\texttt{<}\mathit{def}\ \texttt{name=}a\ \texttt{var=}b\texttt{>}\ \texttt{<}\mathit{val}\texttt{>}\ c\ \texttt{</}\mathit{val}\texttt{>}\ \texttt{</}\mathit{def}\texttt{>}}{\texttt{<}\mathit{def}\ \texttt{name=}a\texttt{>}\ \texttt{<}\mathit{val}\texttt{>}\ \texttt{<}\mathit{var}\ \texttt{name=}b\texttt{>}\ c\ \texttt{</}\mathit{var}\texttt{>}\ \texttt{</}\mathit{val}\texttt{>}\ \texttt{</}\mathit{def}\texttt{>}}$$

Fig. 5. Function definitions may be shortened via this equivalence.

Abstractions or anonymous function declarations. The special **<*var*>** tag is used to denote the free variable for an abstraction. The function **double** is here defined as a "lambda term" using **<*var*>** within the ***val*** element:

```
...
<example>
  <def name="double">
    <val>
      <var name="x"> x x </var>
    </val>
    ...
  </def>
</example>
```

The above is equivalent to the functional programming construct:

```
let double x = x ⌒ x in ...
```

and it means precisely to copy the list argument twice.

It is also permitted to elide the abstraction into the definition, thus:

```
...
<example>
  <def name="double" var="x">
    <val> x x </val>
    ...
  </def>
</example>
```

The general form of this elision is shown in Figure 5.

Abstractions can always be elided (see Figure 6). The abstraction can be bound to a name via a definition, and then the name used instead of the abstraction.

Functional Application. Wherever an ordinary XML tags is used, then it is used like a constructor function. Function applications themselves look just the same as ordinary XML tags:

```
<f> "a" "b" </f>
```

$$\frac{\texttt{<}var\ \texttt{name=}x\texttt{>}\ y\ \texttt{</}var\texttt{>}}{\texttt{<}def\ \texttt{name=}h\ \texttt{var=}x\texttt{>}\ \texttt{<}val\texttt{>}\ y\ \texttt{</}val\texttt{>}\ h\ \texttt{</}def\texttt{>}}$$

Fig. 6. Abstractions may be eliminated via this equivalence.

$$\begin{aligned}
&\text{HOAX}_\rho\ \texttt{::=}\ s\\
&\qquad \texttt{|}\quad \texttt{<}def\texttt{name="}t\texttt{"><}val\texttt{>}\,xs_2\,\texttt{</}val\texttt{>}\,xs_2'\,\texttt{</}def\texttt{>}\\
&\qquad \texttt{|}\quad \texttt{<}var\texttt{name="}t\texttt{"><}xs_2\,\texttt{</}var\texttt{>}\\
&\qquad \texttt{|}\quad \texttt{<}eval\ \texttt{name="}t\texttt{"><}xs_2\,\texttt{</}eval\texttt{>}\ \texttt{|}\ \texttt{<}t\texttt{>}\,xs_2\,\texttt{</}t\texttt{>}\\
&\text{HOAX}_{\rho\oplus\{t\}}\\
&\qquad \texttt{::=}\ \texttt{<}ref\texttt{name="}t\texttt{"/>}\ \texttt{|}\ t
\end{aligned}$$

$$\textit{where}\ xs_1 :: \text{HOAX}^*_\rho, xs_2 :: \text{HOAX}^*_{\rho\oplus\{t\}}, t :: \texttt{name}{-}\rho,\ s :: \texttt{string}$$

Fig. 7. The attributed grammar for higher order documents.

The above means the application of function `f` to the sequence `"a" "b"`. Normally, the type of the tag `<f>` and the fact that it is a function tag will be declared in the document schema, but to allow ad-hoc uses we provide a predefined function `eval` which has the same effect as functional application, but which takes the function name as an attribute instead:

```
<eval name="f"> "a" "b" </eval>
```

Both type and semantics for the function tags need to be defined before first use. The type will normally be declared in the document DTD or schema part.

Dereferences. The special `<ref>` tag can be used to dereference a variable, but in general it is only necessary if the variable has an exotic name, with spaces or reserved characters in:

```
<def name="double" var="x">
  <val>
    <ref name="x"/> <ref name="x"/>
  </val>
  ...
</def>
```

Elisions. `<eval>` and `<ref>` can both be elided, as in the table below, when the identifiers they mention have previously been declared:

undeclared	declared
<*eval* name="*f*"> ... </*eval*>	<*f*> ... </*f*>
<*ref* name="*x*"/>	*x*

And as explained, an abstraction that forms the body of a definition may have its variable declaration absorbed into the definition tag as an attribute.

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:hoax="http://hoax.example"
        targetNamespace="http://hoax.example">

  <element name="hoax" type="hoax:hoax_type"/>
  <complexType name="hoax_type">
    <choice>
      <element ref="hoax:def"/>
      <element ref="hoax:var"/>
      <element ref="hoax:eval"/>
      <element ref="hoax:ref"/>
      <!-- standard top-level xml productions go here --!>
    </choice>
  </complexType>

  <complexType name="name_and_hoax_type">
    <sequence>
      <element ref="hoax:hoax"/>
    </sequence>
    <attribute name="name" type="string"/>
  </complexType>

  <element name="def" type="hoax:def_type"/>
  <complexType name="def_type">
    <sequence>
      <element name="val" type="hoax:name_and_hoax_type"/>
      <element ref="hoax:hoax"/>
    </sequence>
  </complexType>

  <element name="var" type="hoax:var_type"/>
  <complexType name="var_type" type="hoax:name_and_hoax_type"/>

  <element name="eval" type="hoax:eval_type"/>
  <complexType name="eval_type" type="hoax:name_and_hoax_type"/>

  <element name="ref" type="hoax:ref_type"/>
  <complexType name="ref_type">
    <attribute name="name" type="string"/>
  </complexType>

</schema>
</xml>
```

Fig. 8. HOAX texts described via an XML schema.

The basic grammar of HOAX is expressed in Figure 7. To make it easier for an automatic type checker, `type=...` fields as well as `name=...` fields are also allowed in all the tags introduced in this section, but they are not depicted in the grammar within the figure.

3 HOAX Types and Semantics

The HOAX type system will not be described in great technical detail here. It suffices to understand that the types exist and that there are well-defined notions of equality and inclusion between types, although equality is not identity. That is to say there are usually at least two different ways of expressing the same type. The class of types contains at least

1. the string type, `string`, to which all atomic strings belong;
2. the empty type, `()`, or `nil`, to which only nothing belongs;
3. the function space constructor $\tau_1 \to \tau_2$, written $\tau_2(*)(\tau_1)$ to conform with a C language -like format in DTDs (the trick is to imagine a function name where the middle parentheses are, so that the type declaration for function f would read $\tau_2\ f\ (\tau_1)$) and as `<funType type=`τ_1 `\ResultType=`τ_2`>` in schemas;
4. the "in sequence construct", $\tau_1\ \ldots\ \tau_n$, which represents an n-part sequence, written `<sequence>` $\tau_1\ \ldots\ \tau_n$ `</sequence>` in schema notation;
5. the repeat construct τ^*, which represents an arbitrary finite sequence, and which is denoted by the `maxOccurs="unbounded"` modifier in schema notations;
6. the alternation construct $\tau_1|\ldots|\tau_n$, which represents the union of several types, `<alternate>` $\tau_1\ \ldots\ \tau_n$ `</alternate>` ;
7. the free type t of tagged data. The outfix free tag constructor `<`t`>` $-$ `</`t`>` is of type $\tau \to t$, where τ is the stipulated type of the contents of the tag;
8. type variables, which are implicitly universally quantified.
9. the type `anyType`, of which every other type is a subclass.

This contains the standard core XML type constructions.

The typing rules (using the abbreviated type formalism above) for the HOAX grammar of Figure 7 is shown in Figure 9, including a final generic rule for typing sequences. $a \Rightarrow b$ is written where a hypothetical type statement b with hypotheses a is needed. The tautological rule $t :: \tau \Rightarrow t :: \tau$, saying how to type place-holders, needs no special mention, nor does the equally tautological rule of type loosening, $t :: \tau_1 \Rightarrow t :: \tau_2$ where τ_2 is a larger type than τ_1.

Each type really represents a particular *ambiguous parser* of HOAX texts, one which returns as a result of the parse a HOAX document. In this sense, a type is the parser and interpreter of the set of documents which it can parse and interpret.

It can be proved that there is a unique parse for each HOAX document, given the particular parsing semantics used in HOAX. That does not mean that there is a unique type for each document, however. For example, any string will be a member both of the type `string` and the type `string`* (a sequence of strings).

The type system of Figure 9 only describes the (canonical) type of a HOAX document element. Each canonical type describes an interpreter for the document. The detail of the parsing of documents by types is set out in Figure 10. It can be proved that:

Proposition 1. *There is a unique valid parse of a document.*

$$\frac{}{\texttt{"}s\texttt{"} :: \texttt{string}}$$

$$\frac{xs_1 :: \tau_1 \qquad t :: \tau_1 \Rightarrow xs_2 :: \tau_2}{\texttt{<}def\ \texttt{name="}t\texttt{">}\ \texttt{<val>}\ xs_1\ \texttt{</val>}\ xs_2\ \texttt{</}def\texttt{>} :: \tau_2}$$

$$\frac{xs_1 :: \tau_1 \qquad t :: \tau_1 \Rightarrow xs_2 :: \tau_2}{\texttt{<}var\ \texttt{name="}t\texttt{">}\ xs_2\ \texttt{</}var\texttt{>} :: \tau_2(*)(\tau_1)}$$

$$\frac{t :: \tau_2(*)(\tau_1) \qquad xs_1 :: \tau_1}{\texttt{<}eval\ \texttt{name="}t\texttt{">}\ xs_1\ \texttt{</}eval\texttt{>} :: \tau_2}$$

$$\frac{xs :: \tau}{\texttt{<}t\texttt{>}\ xs\ \texttt{</}t\texttt{>} :: t}$$

$$\frac{t :: \tau}{\texttt{<}ref\ \texttt{name="}t\texttt{"/>} :: \tau}$$

$$\frac{t_1 :: \tau_1 \quad \ldots \quad t_n :: \tau_n}{t_1\ \ldots\ t_n :: \tau_1\ \ldots\ \tau_n}$$

Fig. 9. Typing rules for higher order documents.

$$\texttt{string}\ \ulcorner s \urcorner = \{\, [\texttt{Atomic}\ s] \,\}$$

$$\tau_2 \ulcorner \texttt{<}def\ \texttt{name="}t\texttt{"}\ \texttt{type=}\tau_1\texttt{>}\ \texttt{<val>}\ x\ \texttt{</val>}\ f\ \texttt{</}def\texttt{>} \urcorner = \{(\lambda t.\, \phi)(Y(\lambda t.\, \chi)) \mid \chi \in (t :: \tau_1 \Rightarrow \tau_1) \ulcorner x \urcorner,\ \phi \in (t :: \tau_1 \Rightarrow \tau_2) \ulcorner f \urcorner \}$$

$$(\tau_2(*)(\tau_1)) \ulcorner \texttt{<}var\ \texttt{name="}t\texttt{"}\ \texttt{type=}\tau_1\texttt{>}\ f\ \texttt{</}var\texttt{>} \urcorner = \{[\texttt{Abstract}\ (\lambda t.\, \phi)] \mid \phi \in (t :: \tau_1 \Rightarrow \tau_2) \ulcorner f \urcorner \}$$

$$\tau_2 \ulcorner \texttt{<}eval\ \texttt{name="}f\texttt{">}\ x\ \texttt{</}eval\texttt{>} \urcorner = \{\, \phi\, \chi \mid [\texttt{Abstract}\ \phi] \in (\tau_2(*)(\tau_1)) \ulcorner f \urcorner,\ \chi \in \tau_1 \ulcorner x \urcorner \,\}$$

$$t(\tau) \ulcorner \texttt{<}t\texttt{>}\ x\ \texttt{</}t\texttt{>} \urcorner = \{\, [\texttt{Apply}\ t\, \chi] \mid \chi \in \tau \ulcorner x \urcorner \,\}$$

$$(\tau_1\ \ldots\ \tau_n) \ulcorner x_1\ \ldots\ x_n \urcorner = \{\chi_1 \frown \ldots \frown \chi_n \mid p_1 \frown \ldots \frown p_n = x_1 \frown \ldots \frown x_n,\ \chi_i \in \tau_i \ulcorner p_i \urcorner \}$$

$$(t :: \tau \Rightarrow \tau) \ulcorner \texttt{<}ref\ \texttt{name="}t\texttt{"/>} \urcorner = \{t\}$$

Fig. 10. Parse semantics for higher order documents.

Proposition 2. *Application of a function tag* <h> ... </h> *is substitution of the tag contents* a *in the body* H *of the definition of* h*. I.e. The parse of*

```
⌜<def name="h" var="x">
   <val>
     H
   </val>
   <h> a </h>
 </def>⌝
```

is the same as the parse of $\ulcorner H[a/x] \urcorner$*.*

As a corollary, the laws of the lambda calculus hold.

<switch val=x> <case var=p_1> c_1 </case> ...</switch>
<def name=f_1 var=p_1> <val> c_1 </val> <f_1> x </f_1> </def>| ...

Fig. 11. Equivalence for switch statements.

4 Advanced Programming in HOAX

Is the language described practical? For example, can a function be defined which counts the number of leaf elements in an XML tree? To address that point, certain additional features of the language need to be described, in order to allow for legible programming!

The following definition of the higher order function `map` (which applies a parameter function to each of the members of a sequence) uses the `<case>` tag (see Figure 11) to match a value against several pattern/result pairs. The `<var name=...>` syntax is extended to allow *patterns* in the name field. The type given to `map` by this definition is $\tau_2^*(*)((\tau_2(*)(\tau_1))\ \tau_1^*)$.

```
<def name="map" var="#f #x..">
  <val>
    <switch val="x">
      <case var="#a #as..">
        <f> a </f> <map> f as </map>
      </case>
    </switch>
  </val>
...
</def>
```

There is some syntactic sugar (visible in the above) required in order to make patterns both visible and unambiguous. The head/tail pattern of a list is indicated by a space between the two parts in the pattern. The variables in a pattern are indicated by a leading hash-mark, in order to distinguish them from constants. The pattern which captures the tail of a list is indicated with two trailing dots, in order to distinguish it from a pattern which only captures a single member of the list.

A subtlety in the above is that the result of the switch statement is always a list. All possible matches are taken. Because the pattern matches are not ambiguous in the above example, only a singleton results.

As well as a literal match `<t> #as..  </t>` on a tag, a notation derived from XPath [6] is supported. In this notation, the `<t> #as..  </t>` pattern is written `t/#as..`, meaning the list of elements immediately below the tag `t`. The `/` means "immediately below" and `//` means "somewhere below". In conjunction with the interpretation of ambiguity as giving a list of results, this leads to a shorter definition of `map`. The `#..` is a throw-away match for a list, possibly empty:

```
<def name="map" var="#f #x..">
  <val>
    <switch val="x">
      <case var="#.. #a #.."> <f> a </f> </case>
    </switch>
  </val>
...
</def>
```

The sum of the elements of a list is defined similarly to map:

```
<def name="sum" var="x">
  <val>
    <switch val="x">
      <case var="#a #as.."> a + <sum> as </sum> </case>
      <case var="#"> 0 </case>
    </switch>
  </val>
  ...
```

With `sum` defined, the function `count` that counts the number of leaves in a tree can now be defined, and applied. The `#/#.. #a #..` pattern catches any element directly below a tag, and the `#` pattern catches anything else, i.e. a leaf.

```
<def name="count" var="x">
  <val>
    <sum>
      <switch val="x">
        <case var="#/#.. #a #..">&lt;count&gt; a &lt;/count&gt; </case>
        <case var="#"> 1 </case>
      </switch>
    </sum>
  </val>
  <count>
    <hello> "there" <every> "one" </every> </hello>
  </count>
  ...
```

One practical question is how to apply the functions defined in a HOAX document to XML trees found in other XML documents. The solution is to make use of *name-spaces* in naming tags, a theme that cannot be explored here, but which is integral to XML practice.

5 State of the Art

In addition to those approaches already mentioned in the introduction to this article, namely, XMLambda, HaXML and XMerL, among the approaches which

add XML support to existing FP languages, XEXPR (see [2]) is a scripting language that uses XML as its primary syntax, making it easily embedded in an XML document. XEXPR was published as a W3C Note in November 2000. XEXPR takes a functional approach, and maps well onto the syntax of XML.

XDuce ("transduce", see [7]) is a typed programming language that is specifically designed for processing XML data. One can read an XML document as an XDuce value, extract information from it or convert it to another format, and write out the result value as an XML document. Since XDuce is statically typed, XDuce programs never yield run-time type errors and the resulting XML documents always conform to the specified types.

A related approach to the separate treatment of data and transformations of that data is being popularised by XBRL [8], an XML-based "business reporting" language. It separates financial statements and the business rules that operate on them. In the words of the consortium developing the standard:

> XBRL provides a consistent framework that works for all companies and financial statements; XBRL International sponsors the creation of dictionaries of tags (called "taxonomies") that allow the preparation of financial information relative to different accounting and reporting standards, ...

and in that context HOAX applicative semantics may provide a yet more convenient way of expressing the accounting rules than do imperative language constructs.

6 Summary

This article has shown how the data language XML may be adapted to support higher order declarative programming in a natural way. HOAX has itself been prototyped in the functional programming language Gofer.

HOAX types are both the parsers and the interpreters of the language, and the parse of a HOAX document by its type is unique. This approach proves at once that its semantics is well defined. It can also be shown that the semantics given is correct, in the sense that it obeys the substitution rule of the lambda calculus, which means that one document may be referenced by another in HOAX without the reference affecting the semantics of either. Thus HOAX is suitable for use on the WWW, where documents will be widely scattered and independent.

References

1. Dimitre Novatchev, *The Functional Programming Language XSLT - A proof through examples*, November 2001, `http://www.topxml.com/xsl/articles/fp/`
2. Gavin Thomas Nicol, *XEXPR - A Scripting Language*, W3C Note 21 November 2000, `http://www.w3.org/TR/xexpr/`

3. Erik Meijer and Mark Shields, *XMLambda: A functional language for constructing and manipulating* XML *documents*, Draft, December 1999, `http://www.cse.ogi.edu/~mbs/pub/xmlambda/`
4. Malcolm Wallace and Colin Runciman, *HaXML 1.07b*, Manual, September 2002, `http://www.cs.york.ac.uk/fp/HaXml/`
5. Ulf Wiger, XMerl – Interfacing XML and Erlang, in *Proceedings of the Sixth International Erlang/OTP User Conference*, Tuesday October 3, 2000, Ericsson, Älvsjö, Sweden, `http://www.erlang.se/euc/00/`
6. James Clark and Steve DeRose (Eds), XML *Path Language (XPath) v1.0*, W3C Recommendation 16 November 1999, `http://www.w3.org/TR/1999/REC-xpath-19991116`
7. Haruo Hosoya and Benjamin C. Pierce, XDuce: A typed XML processing language, in *Proceedings of Third International Workshop on the Web and Databases (WebDB2000)*, volume 1997 of Lecture Notes in Computer Science, pages 226-244, May 2000. `http://xduce.sourceforge.net/`
8. The XBRL Consortium, *XBRL - the XML Based Business Reporting Standard*, Version 2.0a, 15 November 2002. `http://www.xbrl.org/`

A New Paradigm for Requirements Specification and Analysis of System-of-Systems

Dale S. Caffall[1] and James B. Michael[2]

[1] Missile Defense Agency, 7100 Defense Pentagon,
Washington, D.C. 20301-7100, USA
butch.caffall@mda.osd.mil
[2] Naval Postgraduate School, Department of Computer Science
833 Dyer Rd., Monterey, California 93943-5118, USA
bmichael@nps.navy.mil

Abstract. In a system-of-systems, the number of possible combinations of interactions among the systems is theoretically infinite. System "unravelings" have an intelligence of their own as they expose hidden connections, neutralize redundancies, and exploit chance circumstances for which no system engineer might plan. In this paper, we propose a new paradigm for system-of-systems design. Rather than decompose each system within the system-of-systems in a functional fashion, we treat the system-of-systems as a single entity that is comprised of abstract classes. We demonstrate how our paradigm can be used to both avoid the introduction of accidental complexity and control essential complexity by applying object-oriented concepts of decentralized control flow, minimal messaging between classes, implicit case analysis, and information-hiding mechanisms. We argue that our paradigm can aid in the creation of sound designs for the system-of-systems in contrast to creating a federation of systems through a highly coupled communication medium.

1 Introduction[1]

During the past decade, systems-of-systems have exploded into the battlespace of the joint and coalition warfighters. The acquisition community's response in the U.S. Department of Defense to the rabid craving for more accurate information and more lethal functionality has been a less than stellar hobbling of various legacy systems and ongoing system developments through tightly coupled and lowly cohesive communication shackles.

While there are many issues with system-of-systems acquisitions, the first issue that we must address is the requirements definition and allocation issue. Just as the requirements issue continues to plague single-system acquisitions, the requirements

[1] This research is supported by the U.S. Missile Defense Agency. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotations thereon.

M. Wirsing et al. (Eds.): RISSEF 2002, LNCS 2941, pp. 108–121, 2004.

issue is much more complicated in the acquisition of system-of-systems. For example, many of the systems that comprise a system-of-systems may be legacy systems that operate as stand-alone capabilities in the operational world, having been developed with specific sets of requirements and with specific system functionality in mind. Additionally, just as we developed the legacy systems, we are developing new systems that will become members of a system-of-systems under similar conditions. That is, we are developing these systems as stand-alone capabilities with specific sets of requirements and with specific system functionality in mind.

Now comes the desire to slam these various systems together and connect these systems through some communication medium in the hope of achieving greater functionality, although this tack does not necessarily result in the intended synergistic effect. One can identify the systems that will form the system-of-systems, and then set out to bend, fold, spindle, and mutilate these systems in the fevered hope of producing a functional composition: it is difficult to think about the system-of-systems as a single entity, which may explain why system developers sometimes mistakenly focus on modifying individual systems with little deliberation and consideration for the system as a whole.

1.1 Current Approach to Developing System-of-Systems

Our tools for reasoning about a system-of-systems typically consist of little more than a "sticks-and-circles" diagram. The "circles" represent the various systems that comprise the system-of-systems while the "sticks" are means of information transfer, a messaging protocol, and, perhaps, a translator box to translate the messaging format from one system to another. Armed with this sophomoric view of the system-of-systems, we attempt to analyze and describe the system-of-systems through a trivial picture of the various systems as connected by a convoluted labyrinth of lines. Unfortunately, sticks-and-circles diagrams lack both a formal semantics and the richness needed to express the many dimensions of system behavior [1]. Are the circles meant to represent systems, subsystems, modules, classes, objects, functions, hardware, or some other entity? Are the sticks meant to represent data flow, triggers, synchronization, calls, inheritance, or something else?

Much too often, we initiate detailed design and coding from reasoning about the sticks-and-circles diagrams. During the development, we add new layers of features and functional enhancements to the system software without clear insight into the organization of the system software. Inevitably, the basic organization of the software that seemed so reasonable at the beginning of the development process begins to break apart under the weight of the revisions made to the system software [2]. Sadly, the software development becomes another casualty to report in future studies as to why software developments are not successful.

Traditionally, this methodology failed to achieve an interoperable and integrated system-of-systems. With each new failure, the system engineers attempted to "tighten up" the protocol standard; however, the system-of-systems did not achieve the desired degree of interoperability and integration. The end-state is a collection of systems that are tightly coupled with a realized protocol standard that only serves to greatly increase the system-of-systems software complexity.

As we have witnessed time and again, system software critical interactions increase as the complexity of highly integrated systems increases. In the complex system-of-systems, these possible combinations are practically limitless. System "unravelings" tend to have an intelligence of their own as they expose hidden connections, neutralize redundancies, and exploit chance circumstances for which no system engineer might plan [4]. A software fault in one module of the system may coincide with the software fault of an entirely different module of the system. This unforeseeable combination can cause cascading failures within the system.

Software complexity and size are dramatically increasing in our delivered products. Customers are demanding more features in their systems in less time than ever before. Under the demands of management, software developers scurry to coding with only a minimal amount of planning and reasoning about software architectures and system requirements. As a result of this mad rush to the goal line, software developers are stumbling and fumbling the ball - rarely scoring a touchdown. Unfortunately, software developers are building software products that have about a twenty-six percent chance of completing on time and on budget. For Government software projects, developers have about an eighteen percent chance of completing their projects on time and on budget. Moreover, the delivered products will have fewer features and functions than originally desired by the customer [9]. According to Lesishman and Cook, only two percent of the software is usable as delivered [5].

1.2 A New Paradigm for Developing System-of-Systems

How do we reason about such a structure so that we have at least a modicum of chance to realize a functional system-of-systems? Can we extend the existing set of tools that we use in reasoning about a single system development to the more complex system-of-systems development? If true, can we use these tools to identify potential sources of accidental system software complexity?

A maxim espoused in all engineering disciplines is to "keep it simple." The best we can do in software engineering is to minimize "accidental complexity" and control "essential complexity." Accidental complexity occurs due to a mismatch of paradigms, methodologies, and application tools. Essential complexity is a fact of software engineering in that system software is inherently complex because software applications are the most complex entities that humans build. As system software is used in ways never envisioned by the developers, operators tend to demand that extensions be made to their system software [7].

While we cannot address all of the issues that negatively impact the development (and follow-on maintenance) of system software, we will examine the issue of requirements specification for system-of-systems. Typically, detailed system specifications address merely the leaves of the system-decomposition tree [10]. A software engineer would have extreme difficulty to develop a sound and complete software package from only the very detailed system specifications. Indeed, software engineers require several layers of abstraction beginning at the top layer of abstraction down to the detailed system specifications. It is at the upper layers of abstraction in which software engineers reason about the system and make architectural and design decisions.

We argue that a new paradigm needs to be adopted by the acquisition community in which the system-of-systems is treated as a single functional entity during requirements specification and analysis. Our initial research to develop the new paradigm has centered on the application of object-oriented design (OOD) techniques in conjunction with the Unified Modeling Language (UML), which we report on in the case study presented in the next section.

2 A Case Study from Missile Defense

We begin the discussion of our proposed paradigm by first introducing a hypothetical missile defense system: it is a system-of-systems made of legacy systems and systems to be constructed for providing new functionality for the composite system. Figure 1 is a sticks-and-circles diagram of our hypothetical system.

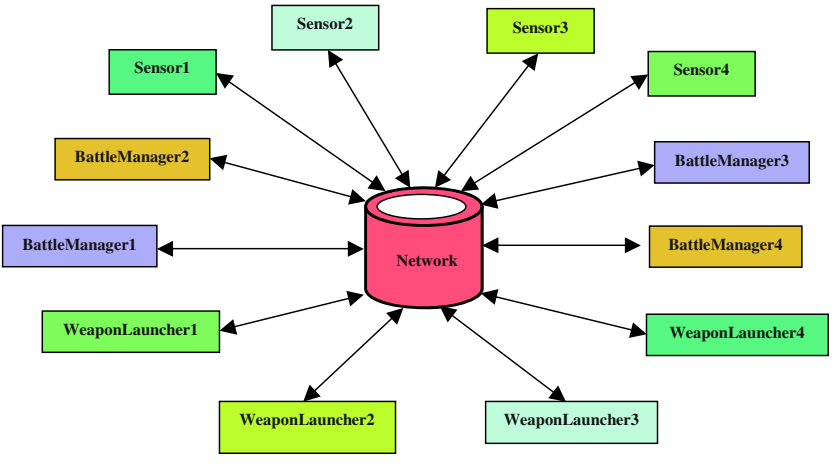


Fig. 1. Components of our hypothetical missile defense system

The greatest source of system software faults will occur in the integration of the various systems. With respect to our case study, the missile defense system will be a complex product that will contain many discrete software packages within each system. As a rule, these software packages will be developed independent of each other and programmed in many different languages. Additionally, the system will include legacy systems that are currently in operation. The means of integrating these elements and legacy systems are intricate tactical data links that support the message transfer within the system-of-systems.

It is difficult to both reason about requirements and analyze the system-of-systems by relying on the sticks-and-circles view. Although presented as a single entity, it is challenging to understand the affects of requirements changes and component limitations in this view. As previously mentioned, our reasoning tendency is to focus on the individual systems of the system-of-systems in the hope that the desired functionality wondrously appears.

Unfortunately, magic and marvel are not tools that are abundantly available to system developers. Their fervent yet futile hopes for integrated systems and desired functionality too often fall shattered on the road of broken acquisition dreams. Frustration and antipathy are the frequent products of system-of-systems development.

Let us propose another view of the missile defense system in which we apply UML and OOD techniques. We will develop a class diagram with abstract classes for the major components of the system-of-systems. We will reason about the class diagram in our attempt to develop subclasses to which we can begin to allocate requirements and analyze system capabilities and limitations. Additionally, we will identify message requirements and message flow in our attempt to reduce coupling in the system-of-systems by developing requirements for simplified interfaces between the components. Finally, we will propose a reassignment of methods to increase the cohesion of the components.

The object-oriented paradigm offers a new system-of-systems requirements and design methodology that provides for both minimizing accidental complexity and controlling essential complexity through the use of decentralized control flow, minimal messaging between classes, implicit case analysis, and information-hiding mechanisms.

While the hypothetical missile defense system will not be a pure object-oriented design, we can incorporate many of the principles of object-oriented design to decrease the complexity of the artifacts produced during the development of the system-of-systems. We believe that software engineers of system-of-systems can use this object-oriented paradigm to produce a sound design for the system-of-systems rather than the traditional federation of systems through a highly coupled communication medium.

2.1 Definition of Classes

The first step in modeling a system-of-systems using our paradigm is to develop a class diagram of abstract classes. For the hypothetical missile defense system-of-systems, we will use the following five classes, with the corresponding class diagram shown in Figure 2:

1. **Threat Missile:** The Threat Missile class is the enemy missile that contains a warhead of mass destruction: nuclear, chemical, or high-explosive munitions. The adversary will launch the threat missile within the confines of his state. The missile will climb into the exo-atmospheric region that constitutes up to eighty percent of the missile flight. The missile will re-enter the atmosphere over our forces or defended assets at which time it will impact at its aim point.
2. **Sensor:** The Sensor class is the object that detects the threat missile. Sensor is an abstraction of two subclasses: Infrared class and Radar class.
3. **BM/C2:** The Battle Manager/Command and Control (BM/C2) class processes track data from the sensor. The BM/C2 monitors the threat missile, develops firing solutions to negate the threat missile, and directs a weapon to launch its interceptor with the BM/C2-provided firing solution. The BM/C2 class is an abstraction for all system echelons of battle management.

4. **Weapon:** The Weapon class develops firing solutions, calculates the probability of kill, and implements the BM/C2 authorization to engage the threat missile.
5. **Interceptor:** The Interceptor class is the engagement mechanism that negates the threat missile. The Interceptor class is the abstraction for both directed and kinetic energy intercepts of the threat missile.

The message requirements in the above class diagram are specific in comparison to the single, large network interface in Figure 2. One can readily determine the messaging requirements of each class, such as the Sensor class determines the attributes of the Threat Missile class, the BM/C2 class needs formed track data from the Sensor class, Weapon class waits for control data from the BM/C2 class, and Interceptor class waits for the interceptor release command from the Weapon class.

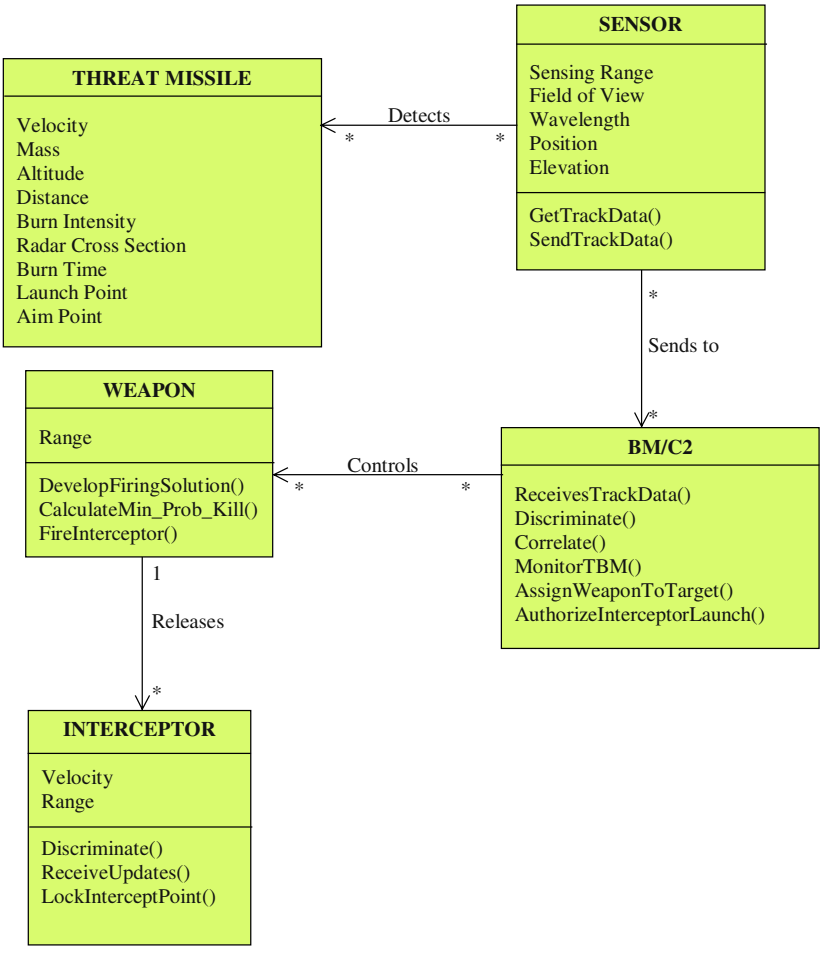


Fig. 2. Class diagram of our hypothetical missile defense system-of-systems

2.2 Definition of Abstract Interfaces and Subclasses

From the class diagram in Figure 2, we can begin to define abstract interfaces between the classes. Rather than the largely unmanageable and complex network interface of the sticks-and-circles diagram, we can begin to develop specific interface requirements from the class-diagram approach.

Let us add detail to the Threat Missile class as this is the point of reference for our missile defense system. We can develop subclasses (i.e., short-, medium-, and long-range threat missiles) of the Threat Missile class as depicted in Figure 3.

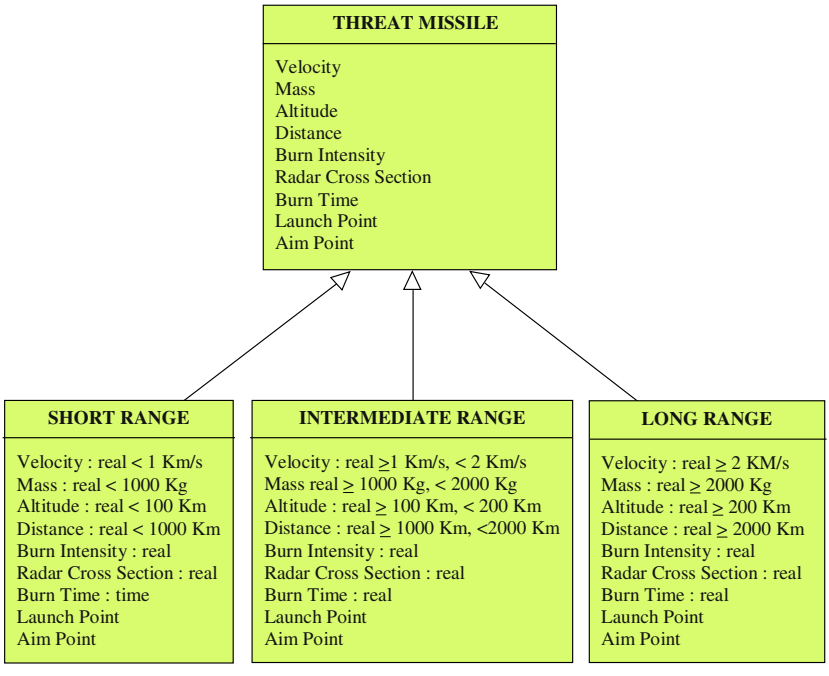


Fig. 3. Subclasses of Threat Missile class[2]

In our definition of the subclasses, we have assigned attribute values. In our example, we have assigned fictitious data so that our example remains out of the classified regime. These subclasses with the assigned attributes will form the basis for our reasoning about the missile defense system.

The Sensor class is responsible for detecting the Threat Missile class, so let us develop subclasses that can detect the Threat Missile subclasses that we have defined. The subclasses for the Sensor class are depicted in Figure 4.

By considering the subclasses of the Threat Missile class, we can design a sensor framework for which we can attain overlapping coverage of our sensor subclasses to greatly increase our opportunities for the detection of the threat missiles. We can also

[2] All attribute values listed in subclasses are fictitious and do not represent real threat missile data.

develop additional requirements to bolster our detection capability. For example, after considering the Threat Missile subclasses for a potential adversary, we may desire to increase the sensing range of the Sea-Based Sensor to extend our coverage into an adversary's territory into which a Ground Sensor solution is not feasible. We can now levy this requirement change on the Sea-Based Sensor subclass.

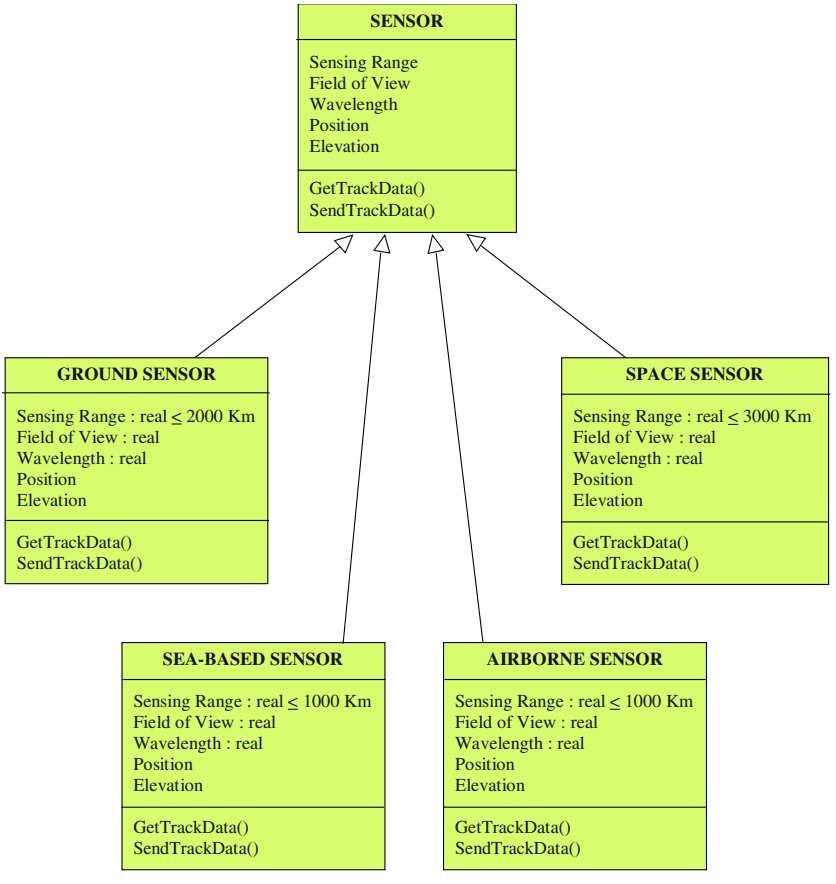


Fig. 4. Subclasses of Sensor class[3]

After we have detected a Threat Missile object, then we must develop a firing solution and engage the threat missile. As depicted in Figure 2, the BM/C2 class handles these functions and several other important functions. While these functions are related, the incorporation of these methods in a single class lessens the cohesion of the class. Rather than a single BM/C2 class, we might develop the BM/C2 class as an aggregate of several classes as shown in Figure 5.

As depicted in Figure 2, we separated the methods for developing and realizing a firing solution from the BM/C2 class and assigned these methods to the Weapon

[3] The attribute values, as in Figure 3, are fictitious and do not represent real threat missile data.

class. These methods are similar in function so the cohesion of this class is high. This separation is important as the realizations of the BM/C2 and Weapon classes may physically reside on different hardware platforms. So, in addition to increasing the cohesion, we reduce the coupling by substituting more interfaces that are small and better defined for the larger interface required for data flow and messaging of the sticks-and-circles architecture depicted in Figure 2. The Weapon class and its associated subclasses are shown in Figure 6.

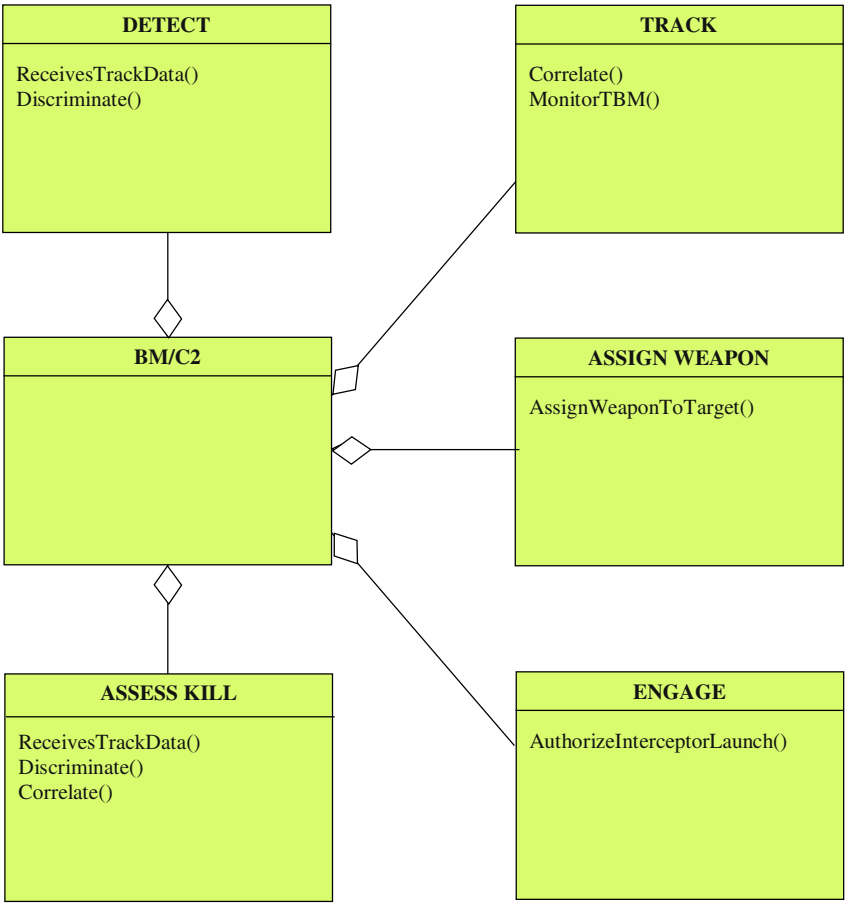


Fig. 5. BM/C2 class as an aggregate

Lastly, we consider the Interceptor class. Given the attributes of the Threat Missile class as well as potential deployment of our hypothetical missile defense system, we can develop the attributes and associated requirements for the Interceptor class. For example, the velocity of the Intermediate Range subclass of the Threat Missile class ranges between 1 km/sec and 2 km/sec and the distance of this same subclass ranges from 1000 km to 2000 km. As we consider the minimum altitude in which we must

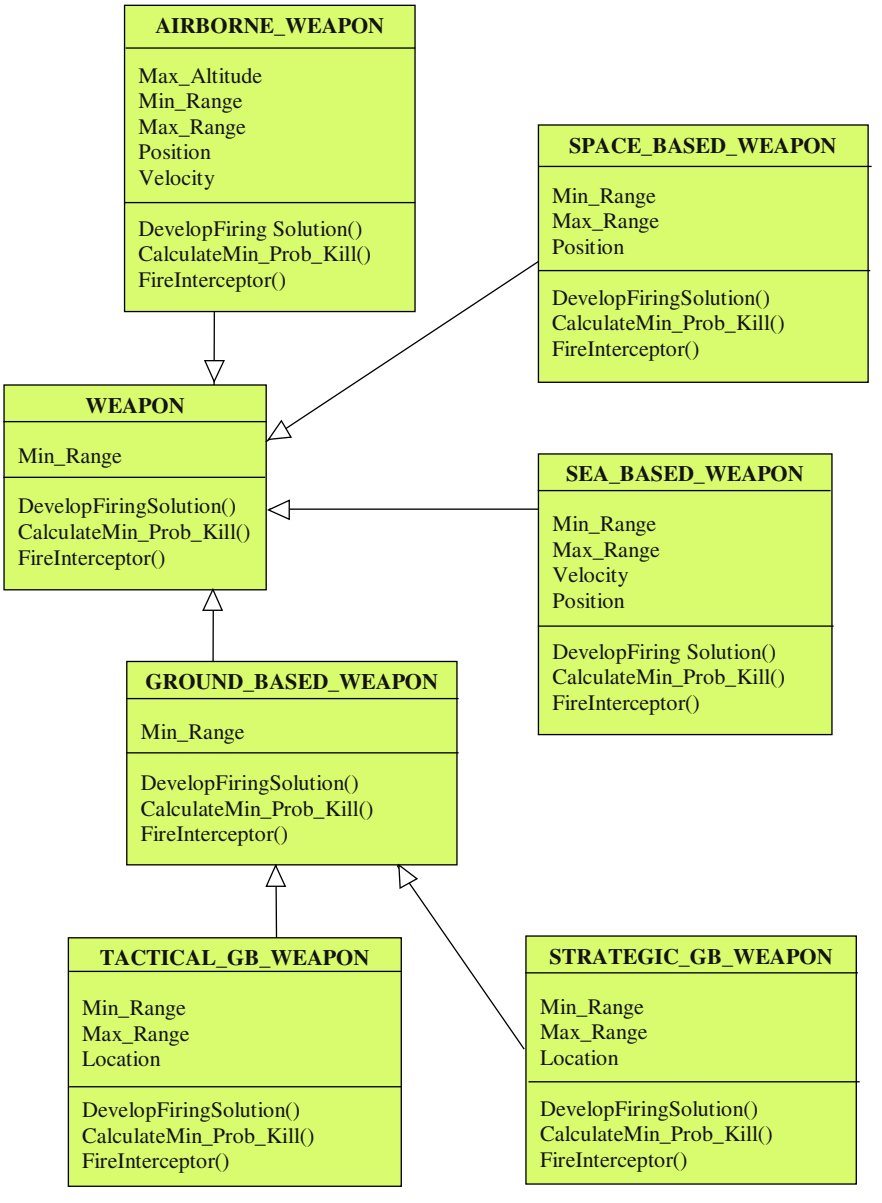


Fig. 6. Subclasses of Weapon class

negate the threat missile to ensure minimal ground effects of the resulting debris, we can determine minimum velocities for our three subclasses of the Interceptor class. These subclasses are depicted in Figure 7.

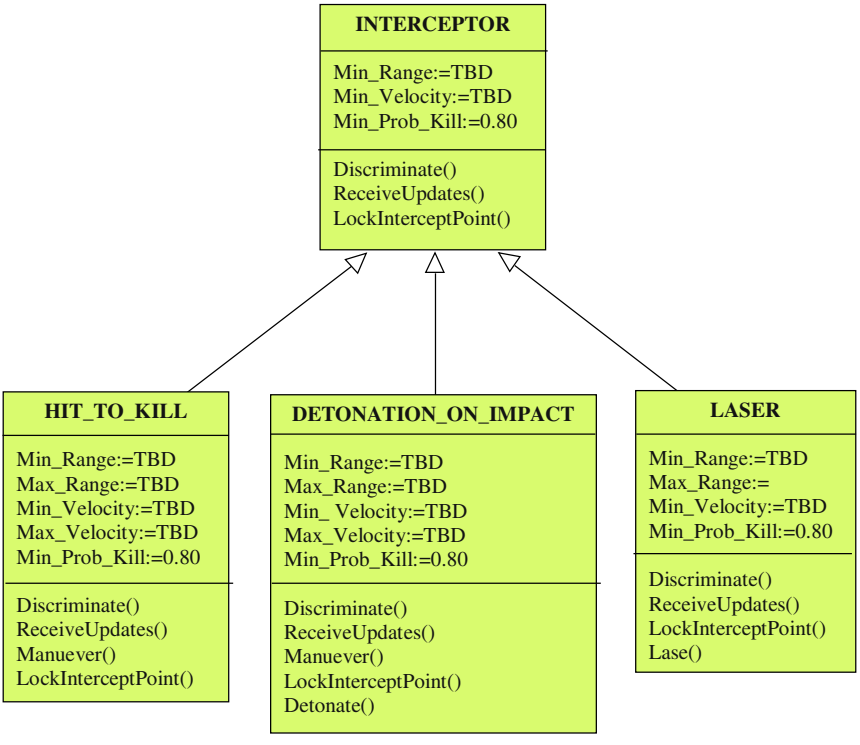


Fig. 7. Subclasses of Interceptor class

2.3 Minimal Messaging between Classes

As we reason about the classes and subclasses of our missile defense system, we can see that we will develop many interfaces in the realization that replaces the single, large network interface of the sticks-and-circles diagram of Figure 2. This is important to us in that we can manage a larger number of small, well-defined interfaces; however, the single, large network interface is much too unwieldy and complicated to manage effectively. We can reduce the messaging requirements of the large network interface to only that which is necessary for realizing the subclasses of our system-of-systems. Because the interface requirements are now manageable and known to all of the system developers, we have enhanced our ability to effectively integrate these systems into a system-of-systems.

Additionally, by treating the missile defense components as classes and developing concise interfaces that implement the minimum level of information sharing among the classes, we can define a data structure that implements data hiding. That is, by reducing the message traffic among classes to only that which is necessary to complete the missile-defense missions and functions, we can prevent external programs from inadvertently modifying the state of a given class or injecting superfluous message traffic that may cause undesired system-of-systems behavior [6].

By defining a data-only interface strategy, as described by Garland and Anthony [3], we can greatly reduce the coupling of the missile-defense components. A data-only interface design will result in a data-only integration realization. That is, each system within the missile defense system-of-systems will provide data that is suitable for transport and use by another system. Thus, *ceteris paribus*, the missile defense system-of-systems should exhibit the following properties:

- More likely to work with legacy software code
- No build-time coupling in any system
- Missile defense systems are not required to share a common platform
- Missile defense systems can share a database to store exchanged data

A final benefit of realizing many small, well-defined interfaces rather than a single large interface will be the flexibility for incorporating future changes in a given class without negatively affecting the other classes. By data hiding and minimal message traffic, the software within a missile defense class is effectively independent in structure and realization than the other classes. As such, an internal software change to any single missile defense class should not affect any other class given that the interfaces among the classes remain unchanged; this cause-effect relationship is discussed in [6].

2.4 Inheritance and Decentralized Control Flow

As we define the class and subclass attributes, the concept of inheritance becomes important in that the allocation of requirements through attributes and methods ensures consistency in the realization of the subclasses in our developments. Each system developer will know the minimum set of requirements that must be implemented and each developer knows what requirements the other developers will realize.

By careful assignment of methods to each class, we can avoid the creation of the so-called "god class" that performs the bulk of the work within the system-of-systems [7]. Typically we overload the battle manager function with the vast majority of the work. More often than not, the battle manager software contains many dissimilar tasks and requires a complex messaging network. Rather than primarily exchanging control or triggering messages among several classes, the typical battle manager requires the continual transport of great amounts of data that results in more complex rules of messaging and bandwidth requirements. By employing the aforementioned UML and OOD techniques, we can reassign methods to other classes in which these methods are better suited.

For example, consider the discriminate method listed in the BM/C2 class in Figure 2. This requires that the Sensor class send a great deal of data to the BM/C2 class. Perhaps we might reason that the Sensor class should contain the discriminate method and send a much smaller, refined track file to the BM/C2 class for prosecution. This would greatly reduce the messaging requirements and greatly simplify the interface between the Sensor class and the BM/C2 class.

2.5 Encapsulation

As we reason about the classes and subclasses of the hypothetical system, we find that we can modify the methods to maximize the benefits of data hiding within the

appropriate class. In the large sticks-and-circles network of Figure 1, nearly all data is public by definition of the single, large interface to each system. By developing appropriate methods for each class, we can begin to hide data within its class.

For example, consider the development of a firing solution for a given threat missile. In the large sticks-and-circles network, the firing solution uses public data that is visible to all other systems. Because the data is public and the network connects each system to all other systems, it is difficult for software designers to understand the impact on system behavior as it is not readily apparent what system functionality is dependent on the public data.

On the other hand, we can determine the data requirements for the development of the firing solution in the Weapon class in Figure 6, and understand that the software developers should hide that data within the Weapon class. While this data hiding may be more difficult in procedural software, the public data issue is more readily apparent in the class views of the system-of-systems than in the large sticks-and-circles network diagram.

3 Integration with Other OOD Techniques

Our approach is intended to be used to architect high-level frameworks for the system-of-systems. Rather than reinvent existing OOD techniques, the results of applying our approach serve as inputs to tools that support OOD techniques for refining high-level frameworks into detailed structural and behavior models with formal semantics. For instance, the Assess Kill subclass of BM/C2 could be refined using the Real-Time Object-Oriented Modeling language (ROOM) [8]: assessing the kill involves accurately discriminating in real-time - during all phases of ballistic missile flight - the payload of the threat ballistic missile from other objects such as deployed countermeasures and debris.

4 Conclusion

By applying UML and OOD techniques to the system-of-systems development, we can glean a great deal more insight into the system-of-systems requirements definition and allocation issues than with the conventional sticks-and-circles diagrams so often used to model these large, complex systems. By developing a class diagram with abstract classes for the major components of the system-of-systems, we can reason about the class diagram in our attempt to develop subclasses to which we can begin to allocate requirements and analyze system capabilities and limitations. Additionally, we can identify message requirements and message flow in our attempt to reduce coupling in the system-of-systems by developing requirements for simplified interfaces between the components. Finally, we can reassign methods to increase the cohesion of the components and we can hide data within a class to minimize the negative impacts of future modifications to either the system functionality or the data.

These aforementioned benefits of applying these UML and OOD techniques cannot be derived from the traditional views of system-of-systems designs. While software designers encounter other problems in system-of-systems designs, we believe

that software developers can more easily reason about the system-of-systems requirements and associated allocation, thereby improving the system-of-systems architectures and designs by employing the techniques previously outlined in this discussion.

References

1. Bachman, F., Bass, L., Carriere, J., Clements, P., Garlan, D., Ivers, J., Nord, R., and Little, R. Software Architecture Documentation in Practice: Documenting Architectural Layers. Special Report CMU/SEI-2000-SR-004, Software Engineering Institute, Pittsburgh, Penn., Mar. 2000.
2. Constantine, L. L., *The Peopleware Papers: Notes on the Human Side of Software*. Upper Saddle River, N.J.: Prentice-Hall, 2001.
3. Garland, J. and Anthony, R. *Large-Scale Software Architecture: A Practical Guide to Using UML*. New York: John Wiley & Sons, 2002.
4. Greenfield, M. A. Mission Success Starts With Safety. Presentation given at the Nineteenth Int. System Safety Conf., Huntsville, Ala., Sept. 11, 2001.
5. Lesishman, T. R. and Cook, D. A. Requirements risks can drown software projects. *CrossTalk* 15, 4 (Apr. 2002).
6. Parnas, D. L. *Software Fundamentals: Collected Papers by David L. Parnas*. Reading, Mass.: Addison-Wesley, 2001.
7. Riel, A. J., *Object-Oriented Design Heuristics*. Reading, Mass.: Addison-Wesley, 1996.
8. Selic, B., Gullekson, G., and Ward, P. T. *Real-Time Object-Oriented Modeling*. New York: John Wiley & Sons, 1994.
9. CHAOS: A Recipe for Success. The Standish Group International, 1999.
10. Weber, M. and Weisbrod, J. Requirements engineering in automotive development: Experiences and challenges, *IEEE Software* 20, 1 (Jan./Feb. 2003), 16-24.

Towards Ontology Driven Software Design

Paolo Ciancarini and Valentina Presutti

Dipartimento di Scienze dell'Informazione, University of Bologna
{ciancarini,presutti}@cs.unibo.it

Abstract. The World Wide Web represents a new space through which any kind of organization can offer services and data. The huge diffusion of this Internet service has led to develop a new kind of software systems, called *Web applications*.
With the new concept of the Semantic Web the development of web applications should evolve including, in their implementation, the use of knowledge representation technologies in order to obtain all benefits offered by the semantics annotation of documents and data.
We are studying the use of UML as a language for modeling organization ontologies as a foundation for designing Web applications which support organizations.

1 Introduction

During the last years the number of organizations which have "faced" themselves on the World Wide Web with their portals has incredibly increased and this trend probably will continue in the future. Organizations more and more rely upon these portals for offering services to their members or to other people. The large amount of information and services that these portals make available is accessible through the specification of URI addresses, the use of search engines, or following links from related arguments.

In order to support this new usage of Web technologies, the concept of a Semantic Web has been introduced by Tim Berners-Lee. The goal of the Semantic Web initiative is to give to all information available on the Web a machine processable form, so that it can be used for several purposes, including more interesting results when searching some specific information, better data integration from different sources, and the automation of organizational tasks across organizations.

The World Wide Web represents a new space through which any kind of organization can offer services and data. The huge diffusion of this Internet service has led to develop a new kind of software systems, called *Web applications*. Usually these are collections of web pages and services, connected through hypertextual links, and supporting a given domain. A Web application describes an organization structure, and provides a set of functionalities to its users which are members and have specific roles in the organization.

With the new concept of the Semantic Web the development of web applications should evolve including, in their implementation, the use of knowledge

M. Wirsing et al. (Eds.): RISSEF 2002, LNCS 2941, pp. 122–136, 2004.

representation technologies in order to obtain all benefits offered by the semantics annotation of documents and data. In fact, a *Semantic Web application* is intended to be a Web application, where all pages are annotated with semantics information, useful for describing machine processable organizational information.

In order to give a machine processable form to organizational information, it is necessary to define *domain ontologies*, which are shared vocabularies suitable to provide say a Web page with a semantic description of its content. A domain ontology maintains meta-information about information, thus it maintains information through the instantiation of the concepts. In order to define such ontologies, a set of standard technologies has to be identified. The inclusion of a standard semantics representation in a Web application enables the automation of tasks through knowledge sharing.

For the purposes of this paper a Semantic Web application is intended to be a Web portal that describes an organization structure, provides a set of functionalities to its users, and manages and shares knowledge on organizational information.

We investigate the design of a Semantic Web application from scratch, focusing on

- how the definition of the domain ontology affects design activities;
- the ontology lifecycle management, and its correspondence with system evolution;
- knowledge sharing among Web applications and automation of tasks;
- knowledge reuse in terms of ontology evolution.

The first issue concerns the technologies used to express a system specification design. The challenge here is to use the same paradigm to represent both the system design and the domain ontology.

In particular, we focus on the object oriented paradigm and the set of standard technologies it is associated to.

More specifically, in some recent papers Cranefield [7, 8] has shown that UML class and object diagrams can have a direct correspondence to ontology concepts. We are exploiting this work in order to improve the design process of web applications.

Let us sketch the general method. When drawing a UML class diagram for a Web application we are defining *de facto* the elements that characterize that domain ontology. For instance, when we design the portal of a University, we have to represent concepts like professor, student, faculty, etc. Then, via the XML Diagram Interchange (XMI, the standard DTD for UML) [6] we can obtain a translation from a UML class diagram to a corresponding RDF schema. The Resource Description Framework (RDF) [2] is the technology that the World Wide Web Consortium proposes to describe the data contained in the Web pages and services, thus actually implementing a Semantic Web. The RDF is a model for metadata and provides interoperability between applications exchanging machine understandable information on the Web.

The choice of UML for describing organization ontologies has the following motivations:

- UML is a standard language for object oriented modelling;
- it is fastly expanding its user community;
- it is supported by several tools that facilitate the design process;
- it is based on a graphical notation that is simpler and easier to read than most description logic formalisms, the usual choice to describe an ontology.
- there exists a DTD that describes the UML metamodel
- the XMI (XML Metadata Interchange) format for the exchanging models is XML-based.

With an ontology-driven system development, several benefits can be envisaged. For what concerns the ontology lifecycle management, an ontology domain changes or evolves when new concepts are added, or new elements replace existing ones to the language that represents an environment. These concepts can be logically managed with UML diagrams.

If the organization structure is linked to the domain ontology, they are both involved in each other changes.

Ontology lifecycle management includes:

- static aspects, reflecting the ontology model;
- dynamic aspects, influence that these changes describing the organization process workflow that has direct correspondence to a Web portal.

Directly connected to these arguments is the topic of knowledge sharing across organizations. With such a knowledge representation it is possible to integrate the domain ontology with knowledge concepts from other applications. This allows the communication between different sources of related environments, and leads to the automation of tasks based on data-sharing.

In our work we exploit a standard UML tool plus a special tool that has been developed by S. Cranfield as support for a new idea for the design of web applications. Our research consists of methodological considerations. The VESA agent, presented in the section 6, is currently being implemented.

2 The Development of Semantic Web Applications

Our ontology-driven development method is based on: the object-oriented paradigm, the use of UML as modelling language, and some tools that are built upon some Web standards. The method suggests a guideline about the technologies that a designer should use during the development of a Semantic Web application and it can be applied to any kind of process that is suitable for the development of Web-based applications.

The process of building Semantic Web applications (or more general hypermedia) applications is not intrinsically different from the one used when building conventional applications. What this new generation of systems has introduced are the concepts of *navigation* and *domain ontology*.

Different approaches have been investigated for the purpose of developing web applications. One of the most commonly used technique is WebML [5]. It is a method based on the Entity-Relation (E-R) model and its features focus mainly on web application appearance. WebML provides an Entity-Relation based notation for the definition of data structure and it can be exhaustive in the case of development of simple Web applications. In order to obtain a full specification of a complex web application, this language should be supported by another one providing enough expressive constructs to fill WebML shortcomings about computational behavior aspects.

Other two commonly used techniques, both based on the Object Oriented paradigm are: the Object-Oriented Hypermedia Design Method (OOHDM) [9], that was the first introducing the concept of navigational design and that has been a guideline for our method, and the Conallen UML extension for web application [14, 15] strongly based on business logic. This extension focuses, above all, on the implementation aspects of the development, in fact it contains concepts as: form, page, script, frame, client, server etc.

3 UML and Ontologies: Related Items

We now examine in some detail the mapping between UML elements and ontologies concepts and the use of this language in the context of some projects that deal with ontology development.

Given an information domain, the corresponding ontology is a schema. In general, a schema is a collection of classes and properties, the relations between them, and the constraints on their interpretation.

The RDF, that is the general model, provides constructs to describe these elements. The most common formalisms used for ontology design derive from Knowledge Representation that is an important sub-field of Artificial Intelligence. Research in this field has produced several knowledge representation languages. The most relevant, as the Knowledge Interchange Format language (KIF), are based on semantic networks, frame systems, and first order logic.

As underlined by Cranefield [1], unless a system that uses ontologies is constructed around a specific tool based on such technologies, the use of a description logic formalism to represent ontologies is not so inviting.

What he proposes is the use of a subset of the UML as ontology representation formalism, in order to describe ontologies in an object oriented view.

In [17] is shown that the RDF Schema is equivalent to a subset of the class model in UML. The equivalence can be stated through the comparison of the Direct Labeled Graph (DLG) representation of the RDF Schema and UML class schemas. It can be shown that the RDF Schema DLG is isomorphic to a subgraph of the UML class schema DLG.

Intuitively:

- UML and RDF classes map between each other;
- RDF Schema properties and UML attributes map between each other;
- The RDF Schema does not support UML operations;

– UML associations are expressed as RDF Schema properties;
– Reification in RDF Schema maps to UML association names and attributes.

The tool that Cranefield has realized is named "UML Data Binding" (UDB). The UDB implements the mappings from UML class diagrams to RDF schemas and sets of Java classes exploiting the XMI encodings of the class diagrams. This XML-based representation of the ontology model is the input of a set of stylesheet that realize the translation.

The generated Java classes allow an application to represent knowledge about objects in the domain as in-memory data structures. The generated schema in RDF defines domain-specific concepts that an application can reference when serializing this knowledge using RDF (in its XML encoding).

Another similar tool is in phase of development by the UBOT team. The UML Based Ontology Toolset (UBOT) [13] is a project that is part of the DARPA Agent Markup Language (DAML) program and is based on DAML ontology development. A schematization of the UBOT model is shown in figure 1 taken from [13].

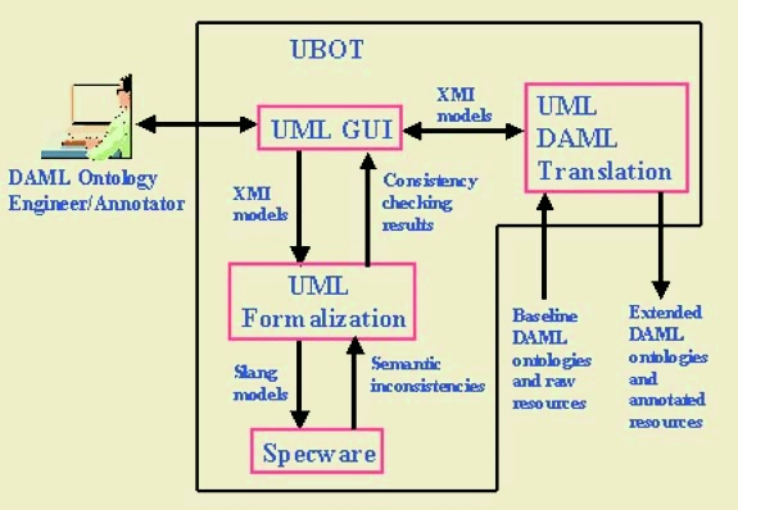


Fig. 1. The UBOT model from [13].

A DAML ontology engineer would graphically model a new ontology or extend an existing ontology with the UML GUI. The model will be exported from the UML GUI in XMI format to the UML Formalization component, which translates the model to Slang for consistency checking in Specware.

The engineer will correct the model based on consistency checking results. The model will then be exported to the UML DAML translation component.

The UBOT has also other objectives besides the creation of DAML ontologies. One of them is building a tool-set that supports the consistency checking of

ontologies. This issue is approached with formal verification and lexical semantics. Another objective concerns the annotation of pages for software agents.

This set of tools can be used, in an ontology-driven design approach, in place of the UDB tool for the definition and the automatic serialization of the ontology schema of the application domain.

4 Our Method Applied to a Case Study

In this section we will sketch a method for the ontology-driven design of a Semantic Web application. This ontology-driven design method will produce a model that will be used to obtain the correspondent RDF specific schema expressed in XML format and will also be used as a conceptual model for a Web portal. Actually, we will take in account only a subset of the entire domain, but it can be a starting point from which the representation of the domain can be completed.

To explain the method we have chosen as a case study a domain that is well known to all academics, namely the organizational structure of a university.

A university is a large and complex educational and research organization. Terms as student, professor, exam, research, library, etc. are typical of this domain. For instance, the University of Bologna (UniBO for short) is composed of faculties, which coordinate teaching activities, and departments, which coordinate research activities. These two structures are autonomous but strictly interrelated. Each faculty offers several courses and benefits of services from a certain number of departments. Professors are employed and work for one faculty. They are also affiliated to a department in which they act as researchers. Professors can teach more that one topic within different courses, etc.

Since UniBO is a large and complex organization the UML model which describes its organizational structure consists of a set of class diagrams each contained in a different package that underlies a specific aspect.

At the top level there are two packages: the General and the UniBOntology packages. The former one contains the model of a General ontology, it has been defined with the aim of showing an example, and it is a UML representation of a subset of the "General Ontology Draft" of the Simple HTML Ontology Extension (SHOE) project [16]. Here, concepts that can be used also for other domains, completely different from the university one, are defined. The General ontology is extended by the UniBOntology through the use of UML specialization constructs that correspond to the `rdfs:SubClassOf` property provided by the RDF Schema.

The two packages are shown in figure 2.

In order to define an ontology for UniBO, we choose to describe the university structure, its hierarchy and organization, using a vocabulary that expresses its typical concepts and relationships.

The elements for the UniBOntology are modelled in the UniBOntology package. The classes that it contains form a taxonomy for the domain while relationships between them and their attributes define the properties of the taxonomy's elements.

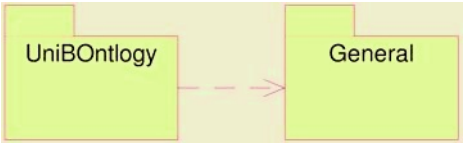


Fig. 2. Top level packages.

The figure 3 shows the set of packages that the UniBOntology one contains. An arrow connecting two packages means that the starting package depends on the ending package (e.g. the ending package defines a class that the starting package uses).

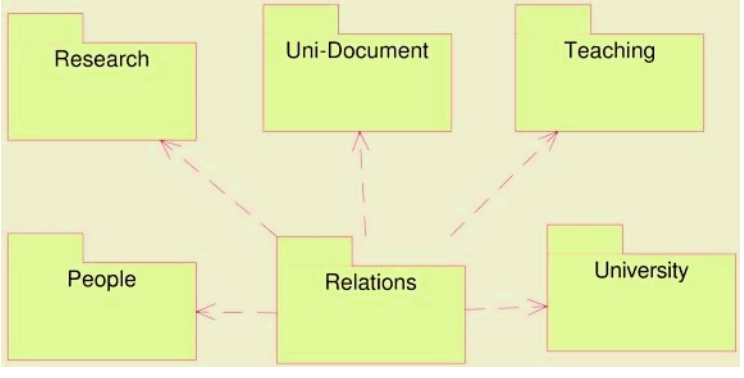


Fig. 3. UniBO uml model: the UniBOntology packages.

The `University` package can be seen as an abstract model of the UniBO organization that omits details, the `People` package contains the classes describing the roles that people act within the UniBO, the `Uni-Documents` package describes the types of documents that can be created as reports of university people's work, the `Teaching` and `Research` packages describe all the elements that contribute to perform the main activities of the UniBO. The `Relations` package contains other packages, each showing the relationships between classes of the various packages.

At the end of the design process we have a set of class diagrams that describes the typical elements of our domain and how these elements relates to each other. Thus, we have a specific schema to express the entities composing the University of Bologna and make them publicly available on the World Wide Web.

The result should be a model for the web portal where all the pages composing it will be automatically annotated with semantics information. A web resource is said to be annotated if it is explicitly and formally associated with information describing its semantic meaning.

The other side of the development should be the specification of a collection of pages and services. The Web portal system specification will evolve with its navigational design, interface definition and implementation. The ontology schema will be used to annotate the pages. They both combined together will give the aimed result.

This description seems to completely separate the two models, suggesting the idea that the ontology serves to integrate the portal with metadata, but it is separately defined. This situation is realistic if we have an existing web portal and we want to provide it with semantics annotations. On the other side, if we are creating from scratch a novel Semantic Web application it is appealing the idea of obtaining the desired ontology schema without an increase of the design activity efforts.

This can be achieved considering that the ontology model is essentially very similar to the conceptual model. In fact, the conceptual model can be obtained directly from the ontology model by simply performing a dropping of the classes representing concepts that are significative only for ontological purpose and that not correspond to any object in the final web application. For instance, considering the Object-Oriented Hypermedia Design Method (OOHDM) [9] this selection can be done without any additional design efforts. It is important to notice that this integration can apply to any kind of process method and that extending the OOHDM is just an example of its possible application.

The figure 4 shows a schematic description of the steps performed in the ontology-driven development process, that extends the OOHDM; some of these steps can be iteratively repeated.

The activities that this method includes are: Ontology Design, Conceptual Design, Navigational Design, Object Model, Interface Definition, and Implementation.

The object model is a collection of UML object diagrams that describes an abstract representation of the knowledge information that will be contained in the specific Semantic web application we are developing. It can be constructed on the base of the ontology and navigational models, that together give the structure of the application. This abstract representation can be used to automatically generate a serialization format to allow web publication and transmission of the knowledge.

The object model will be integrated with diagrams every time new information has to be added, so it can be seen as a component that evolves during the application life cycle.

The ontology-driven process is based on the UML as the modeling notation for the design activities. We have used some tools, allowing the automatic translation from class diagrams and objects diagrams to a correspondent serialization format that allows Web publication. In the context of this work we have used for this purpose the UDB technology [8] together with a Rational UML supporting tool.

The UniBO Semantic Web application will then integrate its realization with the semantic annotation of its pages. This will allow, as said above, the commu-

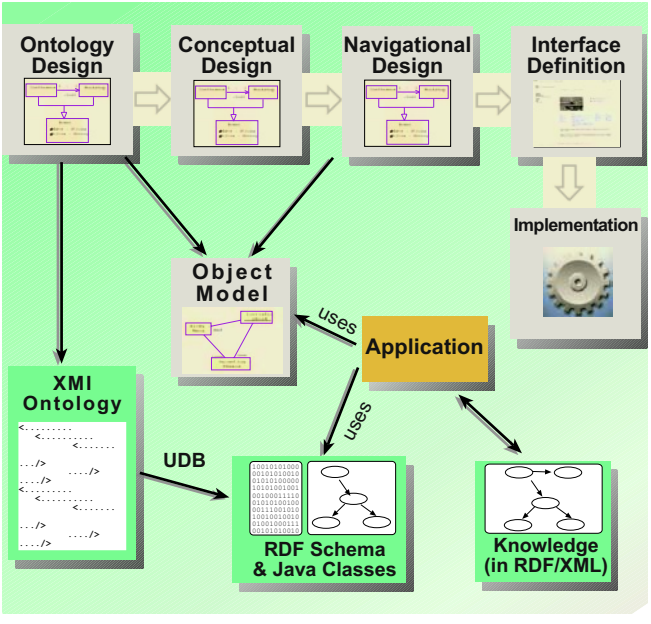


Fig. 4. The ontology-driven method for developing Semantic web applications.

nication between different sources of related environments, and the automation of tasks based on data-sharing. In the next section we describe two possible scenarios where knowledge representation features can be exploited.

5 Two Example Scenarios

A main goal of a Semantic Web application consists of enabling more exact results when a user performs a web search on its specific domain.

Developing an ontology for managing knowledge about academic entities and relationships can have several benefits. For instance, suppose that an undergraduate student has to collect documents about some topic. He could use the UniBO portal to inquire about which are the researchers working on a particular area and the articles that they have written about that topic. Or he could design an agent to search for articles on a particular subject whose authors are member of a particular set of institutions. If the search is based on current search engines (not based on semantic web technologies) the answer could include a large number of uninteresting answers because currently engines rank each page based upon how many of the query terms it contains.

When all the pages of the UniBO web portal are annotated with metadata describing semantics information, the result of the query could be a graph of all possible pathways that match the query. The figure 5 shows a possible layout of the result of a search about all the articles whose subject is the “Ontology

research area" and whose authors are members of departments belonging to UniBo. This example does not refer to a particular implementation for the search engine but just to the environment supporting the engine. For this reason, the syntax of the query can be different depending on the particular implementation of the search engine.

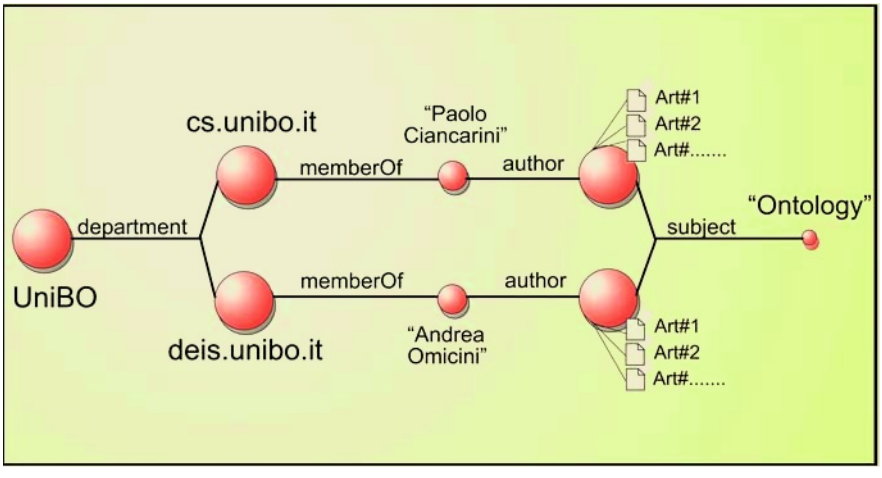


Fig. 5. An example of searching result as a semantic graph.

The nodes of the graph represent resources, and the connecting arcs represent properties. Each arc has a label indicating the property name it represents. All nodes are hypertextual links to the URL containing that resource information. For example, in the case of an article the URL could refer to a downloadable version (eg. a pdf file) or to a web page containing the article. Thus, the user can follow the links he finds more interesting for his purposes.

The second example concerns the interoperability aspect of ontologies. A frequently realistic scenario in our case study happens when a student asks for a transfer from his current university to the UniBO. In this case it is necessary to identify which are the exams that the student has taken in the old university and that will be considered valid also for his career inside the UniBO.

In a situation of ontology sharing, the ontology of the university from which the student comes from has a non-empty intersection with the UniBOntology. A possible way of addressing the validation of exams issue could be the application of the following rule. Each exam taken at the university of origin that has a corresponding one in the UniBOntology will be considered valid and will be stored in the new student's profile created for the University of Bologna. The matching is regulated by semantics constraints defined by the two ontologies.

The figure 6 shows an example instance of a student transfer. In this case the student "Pippo" has taken four exams at the University of Pisa (UniPI for short). By applying the rule described above it results that two of these exams

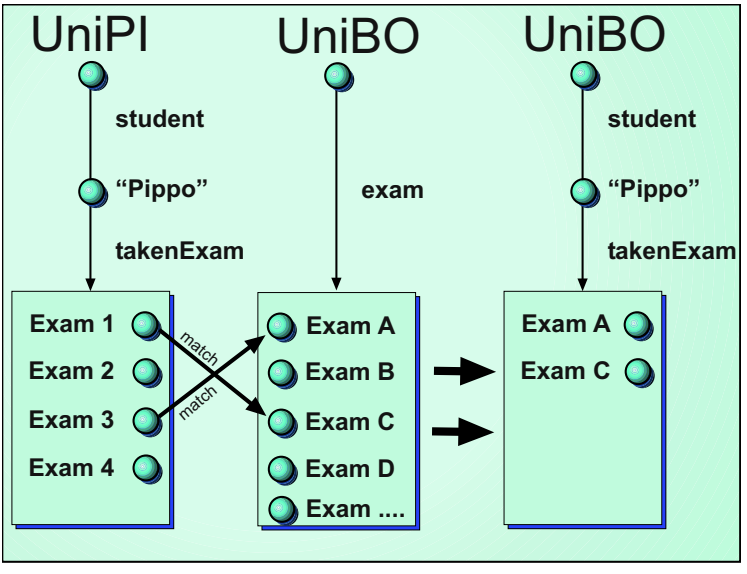


Fig. 6. Exams validation for a student transfer from the UniPI to the UniBO.

can be validated at the UniBO. Thus, these two exams will be stored in a UniBO student profile for "Pippo".

6 The Development of the VESA Agent

We are developing a software agent in order to realize the automation of the exams validation task presented as an example in the previous section. Our agent's name is Validation Exams Software Agent (VESA) and it exploits the annotation of resources.

The first issue is the identification of the environment in which the agent will operate, say the elements it will use and interact with.

The figure 7 shows the input that the VESA agent needs in order to compute its main task and the output it produces.

As it can be seen, it needs two elements as input: one or more ontology references and a set of semantic data. The ontology references allow it to find the definitions of the concepts that are used in the annotated resources.

In this way it can understand those pieces of information. In the figure the ontology is written in OWL as well as the resource semantic annotation. This is only a possibility because the mechanism would be the same if we use other ontology languages.

The VESA output is an XML document containing semantic annotation that describes the new student's profile in the destination university database.

The figure 8 shows, at a high level of abstraction, the whole architecture of the VESA environment.

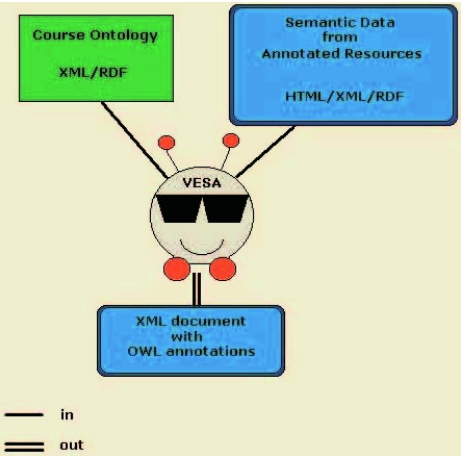


Fig. 7. VESA input-output.

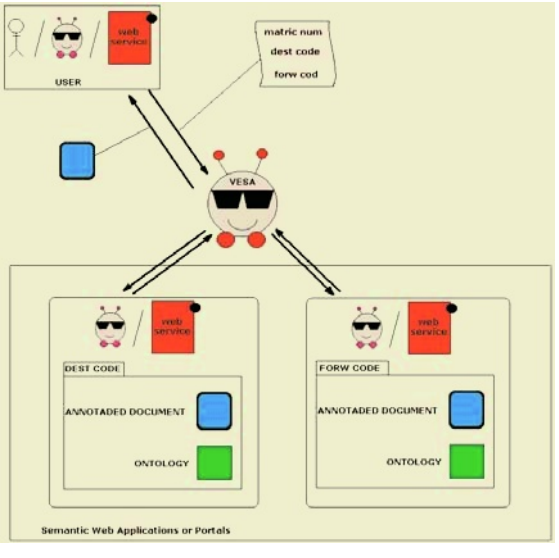


Fig. 8. A high level of abstraction for the VESA environment architecture.

In particular, there is a user that asks for a student transfer providing the necessary information to activate the agent. The user can be any of these: a human person, a software agent, or a web service.

An example of input data are the matriculation number of the student, the code of the forwarding university and code of the destination university.

Then the agent decides if the task can be accepted by computing a validation operation. If it can then it uses the information it holds in order to find the document containing the current student's profile.

If that document contains the proper semantic information then it searches for the ontology references he needs to understand the document and to apply the rules of the destination university for the student transfer.

All the ontology references, the student's current profile and the rules of the destination university are provided by the Semantic web portals that represent the two universities. These portals can supply these information either through a web service or a software agent.

7 Conclusion and Future Works

We have investigated the design of a Semantic Web application from scratch, focusing on how the definition of the domain ontology affects design activities. The challenge was to use the same paradigm to represent both the system design and the domain ontology. In particular we focused on the object oriented paradigm and the set of standard technologies it is associated to.

More specifically, Cranefield [7, 8] has shown that UML class diagrams and object diagrams have a correspondence to ontology concepts and the Object-Oriented Hypermedia Design Method (OOHDM) is an object-oriented-based technic for Web-based software systems development.

In particular our contribution relates on the definition of a method (the ontology-driven design), we have investigated the correspondence between the Semantic web application conceptual design and the domain ontology design. We find that the former can be derived from the latter. Then, via the XML Diagram Interchange (XMI, the standard DTD for UML) [6] we can obtain a translation from a UML class diagram to a corresponding RDF schema and a set of Java classes in order to annotate the web pages composing the Web portal with semantics information about their content. This automatic translation was already possible thanks to related works. In particular, in the context of our research we have exploited the UDB tool [7, 8].

The result has been the definition of a process for the development of Semantic web applications called *ontology-driven method* that consists of two new activities concerning the ontology design and the modeling of knowledge as object diagrams, and the use of supporting technologies like the UDB for the semantics annotation of web pages. Actually, this approach can be applied to all kind of Object-Oriented development processes, in this paper the OOHDM has been taken as a suitable example because it contains concepts that we found interesting.

The real aspect that has been underlined is that the introduction of an ontology design activity in a development process does not introduce new design efforts, and allows to exploit knowledge representation in order to achieve the communication between different sources of related environments, and the automation of tasks based on data-sharing.

As future extension of our work, it is desirable the creation of a UML extension for the treatment of Semantic Web concepts. An agent-based UML extension has been already proposed to the OMG [10]. The definition of a UML

specific extension for ontology modeling would make it easer to handle imported ontology schemas and the modeling of interaction scenarios between agents and Semantic Web applications. Such an extension could be associated to a UDB-like procedure thus allowing the automatic translation from UML models to a correspondent serialization format that allows Web publication. This format could be based on the Web Ontology Language (OWL) [11], the candidate standard for representing ontology on the Web.

References

1. Cranefield, S. and Purvis, M.: UML as an ontology modelling language, in Proc. Workshop on Intelligent Information Integration, 16th International Joint Conference on Artificial Intelligence (IJCAI-99), http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-23/cranefield-ijcai99-iii.pdf"
2. World Wide Web Consortium: Resource Description Framework (RDF) Model and Syntax Specification (1999), http://www.w3.org/TR/REC-rdf-syntax
3. OMG Unified Modeling Language Specification version 1.4, Object Management Group 2001 Recommendation, http://www.omg.org/cgi-bin/doc?formal/01-09-67
4. RDF Core Working Group members: RDF Vocabulary Description Language 1.0: RDF Schema, World Wide Web Consortium, 2002 Working Draft http://www.w3.org/TR/rdf-schema/
5. Bongio, A. and Ceri, S. and Fraternali, P.: Web Modeling Language (WebML): a modeling language for designing Web sites. In Proceedings of the 9th World Wide Web Conference (WWW9), 2000, vol. 33 pages 137-157. Amsterdam, the Netherlands. Computer Networks.
6. Object Management Group: OMG XML Metadata Interchange Specification version 1.2, 2001, Recommendation, http://www.omg.org/cgi-bin/doc?formal/2002-01-01
7. Cranefield, S.: UML and the Semantic Web, Department of Information Science, University of Otago, New Zeland, Discussion paper, 2001
8. Cranefield, S.: Networked Knowledge Representation and Exechange using UML and RDF. Journal of Digital Information, 1:8 (2001)
9. Rossi, G. and Schwabe, and D. Lyardet, F.: Web Application Models are more than Conceptual Models. In Proceedings International Workshop on the World Wide Web and Conceptual Modeling, 239-252 (1999). http://www.dsic.upv.es/west2001/iwwost01/files/contributions/DanielSchwabe/WWWCM99.pdf
10. Odell, J. and Van Dyke Parunak, H. and Bauer, B.: Extending UML for Agents. In Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence, AOIS Worshop at AAAI 2000, 3-17 (2000)
11. Web Ontology Working Group: OWL Web Ontology Language Guide. W3C Candidate Recommendation (2003). http://www.w3.org/TR/owl-guide
12. Schwabe, D. and Rossi, G. and Barbosa, S. D. J.: Systematic Hypermedia Application Design with OOHDM. In Proceedings of the UK Conference on Hypertext, 116-128 (1996). citeseer.nj.nec.com/schwabe96systematic.html
13. DAML/UML Based Ontology Toolset Home page: http://ubot.lockheedmartin.com/

14. Conallen, J.: Building Web Application with UML. First Edition. Addison-Wesley (1999)
15. Conallen, J.: Modeling Web Applications with UML. (1999). http://www.conallen.com/whitepapers/webapps/ModelingWebApplications.htm
16. Heflin, J.: General Ontology (draft). (2000). http://www.cs.umd.edu/projects/plus/SHOE/onts/general1.0.html
17. Chang, W.W.: A Discussion of the Relationship Between RDF-Schema and UML. World Wide Web Consortium Note (1998). http://www.w3c.org/TR/1998/NOTE-rdf-uml-19980804

A Model Based Development Approach for Distributed Embedded Systems

Frédéric Gilliers[2,⋆], Fabrice Kordon[1], and Dan Regep[1]

[1] Laboratoire d'Informatique de Paris 6, 4 place Jussieu,
F-75252 Paris Cedex 05, France
{Fabrice.Kordon,Dan.Regep}@lip6.fr
[2] Sagem SA,
21 Avenue du Gros Chêne, 95610 Eragny,
BP51, 95612 Cergy Pontoise cedex, France
Frederic.Gilliers@lip6.fr

Abstract. Design of reliable distributed systems is stretching limits in terms of complexity since existing development techniques are usually not fully accurate for this type of applications. The main problem is the gap between the various notations used during the development process. Even if UML is a significant step forward, it is not fully suitable for model based development of distributed systems.
We present a model based development approach based on **L*f*P** (Language for Prototyping) applied to distributed systems. It emphasizes the use of a model serving as a basis for automatic code generation; strong connections with formal verification techniques enforce correcteness of the system. The paper focuses on the description of code generation techniques.

1 Introduction

The fast evolution of distributed technology has led to systems stretching that stretch the limits of complexity and manageability [12]. A major problem when distributed systems have to be certified resides in both the design and coding phases: collected requirements may be incomplete, inconsistent or misunderstood, and the numerous interpretations of a large specification often leads to unexpected implementation and additional debugging costs.

The problem comes from the gap between the various notations used in the software life cycle (natural languages, specification languages, programming languages). A first solution is to use a methodology that provides a coherent set of notations to solve this problem. The UML standard [18] represents a significant advance to system specification, however:

- UML semantics is not sufficiently formally defined to enable formal verification unless strong restrictions and hypotheses on its use are introduced (like in [2, 5]);
- Good code generators available for information systems are lacking for distributed applications since UML is not fully adapted to capture all aspects of distributed architectures [15];

⋆ This work is done within an industrial grant provided by SAGEM S.A.

M. Wirsing et al. (Eds.): RISSEF 2002, LNCS 2941, pp. 137–151, 2004.

- UML relies on object orientation, that can be easealy implemented using middleware such as CORBA [17] or Ada-DSA [7] that implement distributed objects. However, the use of other classes of middleware (such as MPI [14]) requires the implementation of adaptation components to handle object oriented mechanisms. Similarily, implementation of an UML specification according to profiles for embedded systems such as ravenscar [4] is delicate since dynamic mechanisms are forbidden in such systems;
- The behavioral semantics of UML will remain informally defined for several years since the 2.0 initial submission claims it essentially formalize static/structural aspects [19]. It introduces OCL to define constraints precisely but only a very limited number of pages are dedicated to the description of unambiguous behavior of a system.

Therefore, for distributed systems, UML is mostly valuable in the early stages of the software life cycle. When a preliminary object-oriented solution is sketch, there is a need for another type of description closer to implementation (e.g. that does not necessarily rely on object oriented middleware). This new description should enable both formal verification (a well accepted approach to ensure high quality in distributed systems) and automatic program generation (to ensure coherence between specification and program).

This paper presents our proposal for a model based development approach. It emphasises the use of a model that formally defines behavioral aspects of the system. This model is used to automatically generate the control code of a distributed application. It is also suitable for formal verification.

Our methodology is presented in section 2. It relies on a notation briefly described in section 3. We then focus on how programs can be derived from these specifications in section 4.

2 Model Based Development

Model-based development [22] focuses on the use of a model that serves as a basis for two main objectives: formal verification and automatic program generation. We favor this approach and consider that it corresponds to an evolutionary prototyping approach [11]. Our proposal relies on **L***f***P**, a high-level modeling language that unambigously describes the behavior of a distributed systems using Behavioral diagrams (automata with structuring facilities). Behavioral diagrams express contracts to be obeyed by components of a distributed system. For example, a class C may state that method M_1 has to be executed prior to the execution of mehod M_2. It may also state that method M_2 cannot be executed twice. **L***f***P** also provides facilities to insert assertions like invariants of temporal logic formulas into the model. This information is suitable for verification purposes and can be used by code generators to optimize programs.

We aim to formalize relations between system modeling, formal verification and code generation of distributed systems in order to provide:

- transparent formal verification to enable its use in an industrial context without requiring both a heavy training and some specific skills, as outlined in [13],

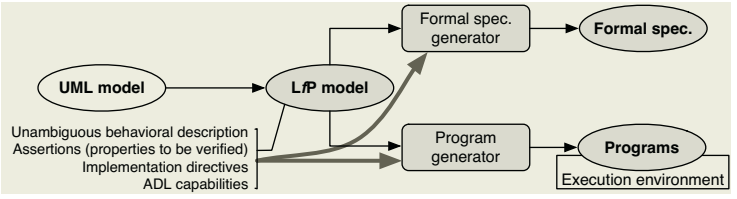


Fig. 1. Model based development using evolutionary prototyping.

– strong correspondence between the detailed description of a system, its proofs and its implementation. In other words: "*what you describe is what you check and implement*".

As shown in Fig. 1, we aim to plug our model based development approach into a "classical" requirement/analysis phase that produces an UML model. First, a reformulation of this model into **L*f*P** must be considered. Most of the work should be done by a tool (producing behavioral state machines from collaboration diagrams cannot be completely automated [9]) but designers have to add information such as the behavioral contracts and assertion to be verified (e.g. "this server has to provide an answer") that cannot be automatically deduced from UML diagrams. When program generation is used, designers also add informations used by code generators (e.g. "components affected to this host are coded in Java"). This additional information is sometimes located in UML tagged values supported by some CASE tools. However, such information is potentially non standard.

From this central description two types of operations can be performed:

– Formal specification generators produce formal specification to be verified: the transformation is optimized according to the property to be verified (i.e. the appropriate couple <*formal method, used technique*> is selected).
– Program generators produce source files to be compiled and integrated in the target execution environment. These generators have to deal with code generation techniques (how to translate the **L*f*P** semantics into a given language) but also with configuration and deployment of the system on top of the target architecture.

Model-based development, similarly to model driven architecture [20], is a development strategy promoting the use of various techniques: modeling (as precise as possible), verification techniques (formal is safer but any other evaluation techniques can also be considered as a first step), and program generation.

In that context, a modeling language such as **L*f*P** serves as an interface to elaborate, by successive refinements, a very precise view on the system, taking advantage of:

– information provided by formal verification (e.g. are all assertions verified?),
– information provided by the execution of previous prototypes (e.g. are performance goals met?).

This paper focuses on program generation techniques. More information concerning formal verification from **L*f*P** can be found in [10].

3 The L*f*P Formalism

This section summarizes the main features of **L*f*P**. It is an Architecture Description Language with coordination facilities that focus on distributed systems. In order to enhance UML models with information that enables automatic code generation of distributed programs as well as formal verification, we define three orthogonal views:

- The *functional view* describes the system software architecture, and links classes (that are execution units) to media (that are communication mechanisms). Both classes and media are described in terms of execution workflow in order to precisely establish behavioral contracts to be analysed and programmed.
- The *implementation view* describes the system implementation constraints (target executive, programming language, communication infrastructure) and the deployment topology.
- The *property view* specifies properties to be verified by the system (analogous to the notion of proof obligation in B [1]). Such properties are stated by means of invariants (for example, confirmation of a mutual exclusion), temporal logic formulas (for example, to the availability or fairness of a service) or other assertions that can be converted into a given formal method. This view can be exploited to perform computer-assisted formal verification. Moreover, it introduces relevant information for code generation (e.g. runtime checks).

The Gas Station Example. To illustrate **L*f*P** features, let us present a model first introduced in [6] and then widely used to demonstrate various verification techniques, as in [3, 24].

It models the simplified behavior of a self-service gas station (see class diagram in Fig. 2). When entering the station, a client prepays the operator, and receives a *ticket* bearing his *id*. The client then proceeds to the pump, inserts his *ticket*, pumps up to *sum* gas, and finishes his pumping operation. He then returns to the operator to get his change and receipt before leaving the station. Information that concern clients being processed is centralized in infosystem. The operator registers the client when it prepays, and unregisters him when he returns. The pump accesses the infosystem when a client activates the pump, and updates the credit when the client puts back the nozzle.

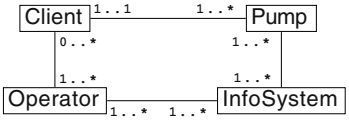


Fig. 2. Class diagram of the gas station.

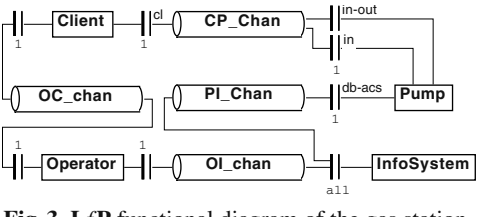


Fig. 3. L*f*P functional diagram of the gas station.

System Architecture. Fig. 3 presents the architectural diagram of the station example. It contains additional information that may be not present in the UML diagram, in order to specify communication patterns between classes. Typically, relations between UML classes (sometimes represented using associations) lead to the creation of media (here *OC_chan*, *CP_chan*, *OI_chan* and *PI_chan*) to describe communication semantics (behavior of communication elements).

Media and classes are connected by means of binders. This notion is inspired from the notion of binding points in RM-ODP [8]. Binders define interaction points between a class instance and a media instance. They correspond to interface buffers associated with characteristics like maximum size, management strategy (FIFO, etc.) and overflow strategy (message loss, client blocked, etc.). Deployment of binders is defined by means of the cardinality (`1` or `all`) specifying if they are shared or not. To avoid overloading Fig. 3, we only list the binders related to *CP_chan* and pump (cl, in-out and db-acs); they are referenced later in the paper.

Let us illustrate how the cardinalities of the gas station model should be interpreted. Fig. 4 shows a class architecture and Fig. 5 corresponds to the corresponding object architecture with two instances of A and C and three of B and D. Each instance of class A (as well as those of class B) has its own buffer connected to the communication system while each instance of class C has its own buffer. Instances of D share a single buffer.

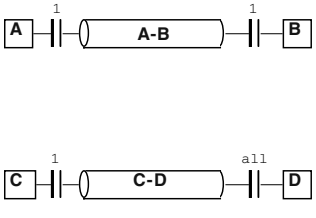


Fig. 4. Class connections: examples.

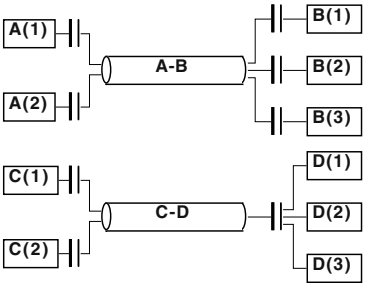


Fig. 5. Class connections: instanciation.

There may be several interaction points (and thus binders) between a class and a media when different characteristics are required. This is the case between *CP_chan* and pump, in Fig. 3: in-out is a two way binder (to support remote method invocation) and sb-acs is a one way binder (used to propagate events).

The behavior of classes and media introduced in the functional diagram is formally described using *behavioral diagrams* (**L***f***P**-BD). They are hierarchical state machines defining what action must be executed based on the internal state of a class instance.

The architecture diagram also declares the number of classes and media instances to be elaborated when the system starts (this is not represented in Fig. 3).

Behavior of a Class. The behavioral contract of a class deals with methods and triggers (an activation condition + code to be executed when the condition is satisfied).

Methods are invoked by other components and triggers are activated by internal conditions. Methods and triggers cannot be executed in parallel. The behavior of a class is expressed using **L*f*P**-BDs, a notation to express state machines. The main level defines the activation conditions of methods and triggers; each transition of the automaton correspond to a method or a trigger. Then, each method and trigger behavior is described using a **L*f*P**-BD, where transitions represent atomic actions to be performed.

Fig. 6, defines the relationship between pump's methods: when `start` is operated, `pump_gas` can be executed until `finish` is called. Asynchronous methods can be compared to message passing (like the *asynchronous* pragma in Ada-DSA [7]). This diagram also declares variables known to the class. In pump, there are only local variables (i.e. each instance of pump has its own copy) but class variables can also be declared (i.e. one copy for all instances of a given class).

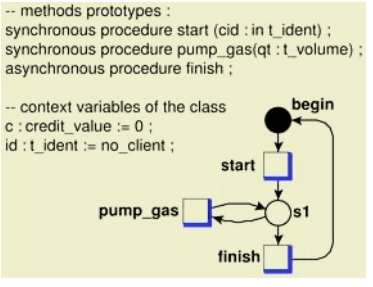


Fig. 6. Behavioral diagram of class pump.

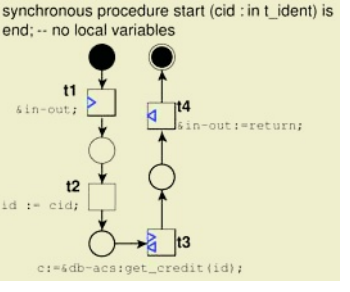


Fig. 7. L*f*P-BD diagram of method `start`.

Methods have to be connected to binders through which they get parameters, send results or invoke services provided by other classes. Triggers may also be connected to binders if they send/receive information from other classes. These connections are defined when describing the execution flow of a method (or trigger). Fig. 7 shows the **L*f*P**-BD that specifies the execution flow of `start`:

- at **t1** and **t2**, the parameter `cid` of the method is extracted from the `in-out` connection point when the query is issued and then copied into variable `id`,
- at **t3**, the credit value for the client is requested through an invocation of method `get_credit`; the query message is issued and the pump instance waits until the value is available to assign to variable `c`,
- at **t3**, an empty message is sent back to the client that issued the query to signal the execution end (`start` is declared as a synchronous method in Fig. 6).

In order to get a complete view of a class behavior, the sub-diagrams that describe individual methods are inserted in the behavioral contract. For instance, the begin state of the `start` automata is merged with begin in the main diagram and the end state of the `start` automata is merged with s1.

Media Behavior. Media have no methods. The associated **L*f*P**-BD describes the communication semantics to be supported.

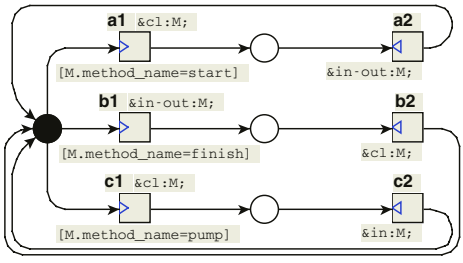


Fig. 8. Partial behavioral description of the *CP_chan* media.

Let us consider the specific protocol associated with media *CP_chan*, that connects a client to a pump (Fig. 8). The **L*f*P**-BD states that:

- only one message is handled at a time by a given instance of media **CP_chan**,
- variable `M` stores a **L*f*P** messsage,
- messages coming from the binder cl are routed according to the method parameter they transport; this is stated by means of the guards associated to transitions **a1** and **c1** (reference to the predefined operator `method_name`). Method `start` goes to binder in-out (an answer is expected, it will be sent via transition **b1**) and `pump` goes to binder in (asynchronous method, no answer expected),
- messages coming from binder in-out are all routed to cl,
- no message originates at binder in.

Other capabilities of **L*f*P** that are not listed here (but presented in [10, 23]) are:

- definition of constructors to dynamically create new instances of a given class,
- use of enhanced data structures such as arrays, records and bags,
- definition of critical sections to protect shared variables,
- use of predefined instructions to label transitions (basically, loops and tests),
- assertions that are "proof obligations" for verification, and may lead to the generation of runtime checks.

4 Code Generation from L*f*P

Code generation translates structures and operators into calls to the primitives of a runtime that provides procedures and services required by the **L*f*P** semantics. Generated programs handle distributed control of the application and thus are executed over several hosts. We first study the general architecture of generated applications and then focus on the translation procedure itself.

4.1 Architecture of the Generated Code

Each host that supports execution runs a partition of the system (we call it a *node*) according to the architecture presented in Fig. 9. A partition consists of classes, media and/or binders instances.

The generated application is built on top of a system-dependent layer, the *runtime*, which provides a set of standard subroutines required to support the **L*f*P** semantics. The interface provided by the runtime to the generated programs remains the same, whatever the target architecture. Therefore, for a given language, the generated code may be deployed on various operating systems for which a runtime is available. To ease the porting of the runtime, it is split in high level services containing non-platform specific services (garbage collection if any, buffer management, etc.), and low-level services containing platform specific services (such as threading mechanisms, memory allocation, etc.). Low-level services rely on the execution environment (operating system and/or communications libraries and/or middleware).

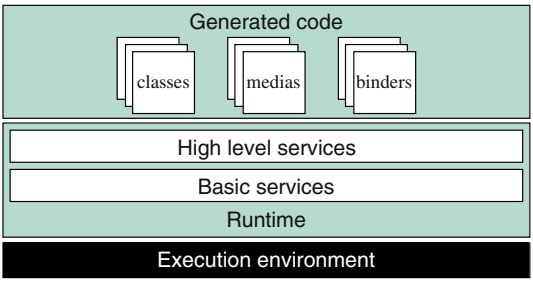


Fig. 9. Architecture of the generated Code and its environment.

To ensure that generated code has a minimal footprint, high-level and low-level services are divided into components corresponding to operations in **L*f*P**. The component corresponding to a given operation is embedded in the runtime if it is activated in the corresponding node.

When a class or media is tagged *external* (e.g. it is legacy software), only interfaces are generated. These interfaces should be enriched if necessary to fit the implementation of the **L*f*P** interactions strategy. For example, if one wants to use sockets, a media provides an abstraction of socket mechanisms that is used, first for modeling and verification, and subsequently to generate an empty interface that can be invoked by the generated code. Mapping of the generated interface to sockets primitives has to be done manually. External components are also a way to insert hand written code into an application when the generated programs do not respect performances requirements.

4.2 Generating Code for Classes

Classes are the smallest unit of concurrency. Therefore the hierarchical automaton of a class, defined by means of **L*f*P**-BD, is translated into a sequential program. It appears useful to maintain the hierarchical structure for two reasons: readability/traceability, and optimization of the code. The code generator must consider the following elements:

1. *data types* used by the class,
2. *local variables* that are the non static attributes of the class,

3. ***LfP-BD methods*** that describe the execution flow of a method,
4. *shared variables* that are the static attributes of the class (i.e. the variable is shared among all the instances),
5. *critical sections* that specify synchronizations in the **L*f*P** model,
6. *evaluation of transitions guards* that select the next transition to execute,
7. *evaluation of assertions* to perform runtime checks on the model,
8. *connections to binders* that correspond to synchronization points with a media,
9. the ***LfP-BD class behavioral contract*** that implements the **L*f*P**-BD diagram of the class.

The architecture of a class is presented in Fig. 10. The generated code for the **L*f*P** class is made of several interacting software units. For example, the code embedded in `PumpPackage` contains of the following elements: `PumpTask` which implements the class's contract, `PumpMonitor` to synchronize the access to binders, `PumpImplementation` that embeds the class's methods and local attributes, `PumpPredicates` which contains the assertions and the guards associated to the class, `PumpSharedVariables` which contains shared variables, and `PumpTypes` which implements local data types.

The class implementation is also related to global units: `GlobalTypes` defining model level types, and some `Binder` elements implementing the connected binders. For the pump class of the example, they are: *in*, *in-out* and *db-acs* (see Fig. 3).

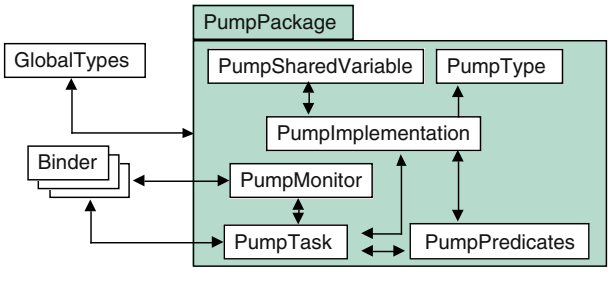


Fig. 10. Structure of the code generated for the pump class.

Global and specific types are embedded into separate packages that easily translated into instructions since construction rules in **L*f*P** types are very similar to those proposed in conventional programming languages. Types and variables visibility is preserved to avoid complex renaming and ensure readability. Global types definitions are produced in a `GlobalType` component that is imported by the `PumpPackage`.

Shared variables of an **L*f*P** Class are handled by a specific package (`PumpSharedVariables`). Instances of a class may be created on several nodes of the application and variables remain global whatever the distribution is. Therefore, the generated code handles consistency of these variables and provides synchronization as well as `set` and `get` primitives.

Transitions guards enable or disable transitions. If a guard only refers to local variables, it is implemented as a boolean condition. If it refers to the body of an incoming message, it is implemented via a two phase transaction handled by the related binders. There are three primitives: `Get` retrieves the message from the buffer, `commit` completes the current transaction when the guard is satisfied, and `rollback` aborts it. When a state is followed by alternative transitions, a select-like structure is generated, as shown for s1 in the behavioral contract of pump (see Fig. 6).

Assertions are also implemented as boolean expressions that may raise an exception at runtime. Assertions and guards are embedded into the `PumpPredicates` program unit.

Methods and local variables are both implemented in `PumpImplementation`. Code generation reproduces the automata hierarchical structure: code for transitions performs operations, states are associated with labels. For a given `State` the execution sequence is: *1)* evaluation of guards, *2)* execution of the transition body, *3)* verification of assertions if any, *4)* jump to next state.

Relation to binders are implemented using a monitor (`PumpMonitor`). There is a monitor for each class instance. Its role is to synchronize execution of the class with incoming messages. When a message is required to fire a transition, the program registers with the binder and waits until the message arrives. If a message is already in the queue, the invocation of the registration primitive is non-blocking. When a message is read, the `PumpTask` unit (that handles the execution contract) may evaluate a guard to decide if the message has to be consumed or not.

The behavioral contract is implemented as a separate task for each instance and located in `PumpTask`. It only accepts messages corresponding to methods that are enabled in its current state. The current state is encoded as a local variable that selects an case alternative.

To read messages from binders, the task first `reset`s the associated `monitor`, then `register`s itself with binders. When a message can be read (e.g. there is a message respecting the transition guard), the class consumes it, unregisters from the binders, executes the selected method, verifies the corresponding assertions and goes to the next state. When no message is available or conform to the guard, the task waits on the monitor (`sleep` entry of the monitor) until a message arrives to one of the connected binders.

Writing into a binder is simpler, a class instance `put`s the message into the binder.

Fig. 11 illustrates the structure of the code generated for state s1 of class pump (see Fig. 6). In this state two methods are enabled; the one to be fired depends on the next binder that will receive a message. First the class resets its monitor and register itself with the two binders. Then it loops to check the content of both binders. If any valid message (e.g. satisfying the transition's guard) is found, the class executes corresponding code, unregisters itself from the binders and jump to the next state. If no valid message is found (there is no message or the first one is not valid) the class waits on the monitor. Invalid messages are left in the binder.

```
S1_State:
   -- reset the monitor
   Pump_Monitor.reset();
   -- register to connected binders
   in-out.register(this);
   in.register(this);
   loop
      -- non-blocking retrieval of an eventual message
      in-out.get(message);
      -- if a message exist in the in-out binder
      if message /= null then
         -- if it is a pump call and guard is true
         if Pump_Predicates.Pump_gas_pre(message) then
            -- consume the message
            in-out.commit();
            -- execute the pump_gas Method
            Pump_Implementation.Pump_gas(message.qt);
            -- check assertions
            Pump_Predicates.Pump_gas_assert();
            -- unregister from binders
            in.UnRegister(this);
            in-out.UnRegister(this)
            -- jump to next state (S1_State)
            goto(S1_State);
         else
            -- unexpected messages or unverified
            -- guard, rollback transaction and
            -- try another binder
            in-out.rollback();
         end if;
      end if;

      -- non-blocking retrieval of an eventual message
      in.get(message) ;
      -- if a message exists in the in binder
      if message /= null then
         -- if it is a finish call and guard is true
         if Pump_Predicates.Finish_pre(message) then
            -- consume the message
            in.commit();
            -- unregister from binders
            in.UnRegister(this);
            in-out.UnRegister(this);
            -- execute the finish method
            Pump_Implementation.Finish();
            -- check assertions
            Pump_Predicates.Finish_assert();
            -- jump to next state (Begin_State)
            goto(Begin_State);
         else
            in.rollback();
         end if;
      end if;
      -- Sleep on the monitor until next message on
      -- registered binders
      Pump_Monitor.Sleep(wait_delay);
   end loop
```

Fig. 11. Pseudo-code of pump related to state s1 and activation of methods `pump_gas` and `finish`.

4.3 Generating Code for Media

Media are communication mechanisms between classes. Two strategies are considered.

When it is tagged "*external*", the media corresponds to legacy software (a communication library). Only an interface defining the functions required to interact with associated binders has to be generated. The media definition is used first for modeling and verification purposes, then to generate an empty interface to be invoked by classes. For example if a designer wants to use sockets, the corresponding media provides appropriate interfaces and abstraction. Mapping to the socket library has to be done manually; the resulting component is reusable. Off-the-shelf media will be provided (such as sockets or RPC).

When it is tagged "*internal*", a media is translated into an automaton implementing the specified protocol.

Media also allow support the definition of implementation-independent higer level communication mechanisms. For example, a channel media may encapsulate various types of implementations: sockets, shared data segments, etc. Such abstractions are of interest to ensure portability over several target architectures (hardware + operating system).

4.4 Generating Code for Binders

Binders are connection objects between instances of Classes and Media. They are implemented as distinct code units. There are several implementation schema depending on their characteristics: multiplicity, blocking/non-blocking primitive access, etc. Implementation strongly relies on the **L*f*P** runtime that provides generic message passing and instantiation services.

```
generic class Blocking_fifo (Size, Message_type, Client_ref_type)
begin
     protected entry Put (Message : in  Message_type) when not transaction;
     protected entry Get (Message : out Message_type) when not transaction;
     protected entry Commit () when transaction;
     protected entry Rollback () when transaction;

     protected entry Register (Client : in Client_ref_type);
     protected entry UnRegister (Client : in Client_ref_type);

  private :
     Buffer is array (1..Size) of Message_type;
     First, Last := 1;
     No_items := 0;
     Client_list is list of Client_ref_type;

     Transaction is Boolean := FALSE;
end;
```

Fig. 12. Specification of a blocking FIFO.

Binders are implemented as instances of generic templates to enable pattern reuse. Fig. 12 shows the pseudo-code of binders interface (here, a FIFO buffer). This template has three generic parameters: `size` (capacity of the buffer), `message_type` (type of transported data), `client_ref_type` (reference type to designate clients).

Private data implement a fixed size FIFO, a list of subscribers and a boolean variable to indicate if a `get` transaction is currently running.

Access to the template services are protected (i.e. both mutually and self exclusive). `Put`, `get`, `commit` and `rollback` are I/O primitives while `Register` and `Unregister` are dedicated to client management. `Get` starts a transaction. `Commit` or `rollback` entries end the transaction. Pending `Put` or `Get` are executed only when no transaction is running.

4.5 The L*f*P Runtime

The runtime provides a set of services to handle the **L*f*P** semantics using primitives of the target execution environment. The runtime provides the following services:

- *Task management service* deals with creation, synchronization and termination of the tasks that handle the execution contract of classes and media (e.g. `PumpTask` in Fig. 10). It is also a basis for implementing class monitors (e.g. `PumpMonitor` in Fig. 10).
- *Resource management service* handles creation, initialization and destruction of **L*f*P** code elements (classes, media and binders instances).
- *Registry service* (in the meaning of Java-RMI [16]) stores global references to generated code units and runtime units at execution time. It is required for distributed deployment when naming conventions have to be preserved over several adresses spaces. This is the case when a prototype is deployed on two middlewares (e.g CORBA and Ada/DSA).

Fig. 13 presents in the form of an UML class diagram the relationship between the **L*f*P** runtime and the application. This interaction model is inspired by RM-ODP [8]. The runtime contains three logical units dedicated to management:

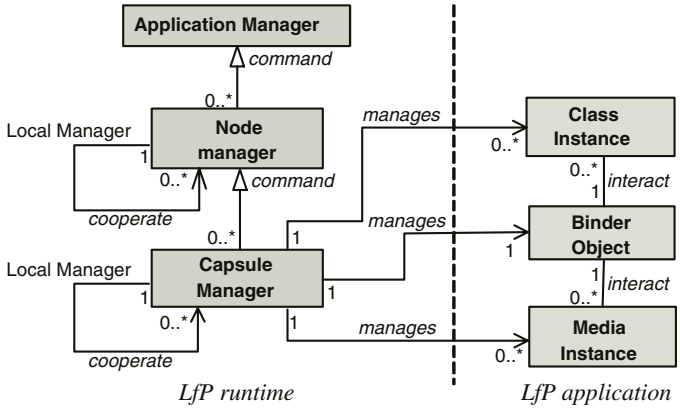


Fig. 13. Relationship between the **L*f*P** runtime and applications.

- The *application manager* handles initialization, termination and error management for the application. It relies on node managers to supervise hosts specific tasks. The Registry service is implemented in the application manager since it is used by all the components of the application.
- Each *node manager* handles a partition of the application on a given node. It implements process creation which is part of the runtime task management service.
- Each *capsule manager* handles instances of a given class or media within a given partition. It supports both task and resources management services.

The runtime implementation depends upon the selected execution environments. There are two types of execution conditions:

- Some applications focus on the use of thick execution environment such as CORBA, Ada-DSA or JAVA-RMI. These environments offer sophisticated services such as naming, dynamic remote creation of objects, etc.
- Other applications have critical time and/or memory constraints. They require thin execution environment such as QNX [21]. Code generation also requires specific strategies to minimize memory footprint or optimize execution time.

The runtime architecture presented in Fig. 13 is able to fit those two types of constraints with tolerable performances. The use of a thick execution environment is not a problem since they support most of the functions required in **L*f*P**. The runtime is then minimal but relies on more complex services. The use of thin execution environments requires a more complex runtime that relies on very simple but efficient services. However, it is possible to write most of the **L*f*P** capabilities with respect to the requirements of embedded systems. For instance a partition may include a static scheduler that handles instances of **L*f*P** classes as threads taken from a pool whose size is fixed at compilation time and yet be compatible with an interpretation of Fig. 13; then, many components are reduced to very limited code.

5 Conclusion

This paper presents a model based development approach for distributed applications. It relies on **L*f*P**, a notation to capture the behavioral semantics of such systems, and serves as a basis for both formal verification and automatic code generation.

We consider that such an approach is a valuable extention to UML based design methods. The combined approaches (UML for object-oriented design and **L*f*P** for a process-based implementation) offer a way to move from an object oriented design to a communicating processes oriented implementation (which is more natural for distributed systems) and provides independence from middleware. Our approach also enables the use of formal methods as described in [10].

We propose a mapping of **L*f*P** concepts to a generic architecture that can be implemented on top of various execution environments. This is a way to help engineers to design and implement complex systems without getting into the often complex and delicate task of using sophisticated middleware services.

Our generic architecture relies on a runtime that virtualizes the execution environment. This is of particular interest when the application executes on several hosts running different operating systems. Effort expended on the implementation of the runtime on a given target architecture can be reused for future applications.

Future work aims to provide a set of coherent tools based on **L*f*P**. This is the goal of a project founded by RNTL (Réseau National des Technologies Logicielles, a french label and founding provided by the government for cooperation between industry and universities), dedicated to embedded distributed systems, that started in July 2003.

References

1. J. Abrial. *The B-book*. Cambridge University Press, 1995.
2. P. Bose. Automated translation of UML models of architectures for verification and simulation using SPIN. In Robert J. Hall and Ernst Tyugu, editors, *14th IEEE International Conference on Automated Software Engineering, ASE'99*. IEEE, 1999.
3. J. Corbett. Evaluating deadlock detection methods for concurrent software. *Software Engineering*, 22(3):161–180, 1996.
4. B. Dobbing and A. Burns. The Ravenscar tasking profile for high integrity real-time programs. In *Proceedings of SigAda'98*, Washington, DC, USA, November 1998.
5. S. Gnesi, D. Latella, and M. Massink. Model checking uml statechart diagrams using jack. In *4th IEEE International Symposium on High-Assurance Systems Engineering*. IEEE, 1999.
6. D. Helmbold and D. Luckham. Debugging Ada tasking programs. *IEEE Software*, 2(2):47–57, March 1985.
7. ISO. *Information Technology – Programming Languages – Ada*. ISO, February 1995. ISO/IEC/ANSI 8652:1995.
8. ITU-T. Open Distributed Processing, X.901, X.902, X.903 and X.904 standard. Technical report, ITU-T, 1997.
9. I. Khriss, M Elkoutbi, and R. Keller. Automating the Synthesis of UML StateChart Diagrams from Multiple Collaboration Diagrams. In Jean Bézivin and Pierre-Alain Muller, editors, *First International Workshop on The Unified Modeling Language, UML'98: Beyond the Notation*, volume 1618 of *LNCS*, pages 132–147. Springer-Verlag, 1999.

10. F. Kordon, , I Mounier, E. Paviot-Adet, and D. Regep. Formal verification of embedded distributed systems in a prototyping approach. In *International Workshop on Engineering Automation for Software Intensive System Integration*, June 2001.
11. F. Kordon and Luqi. An introduction to rapid system prototyping. *IEEE Transaction on Software Engineering*, 28(9):817–821, September 2002.
12. N. Leveson. Software engineering: Stretching the limits of complexity. *Communications of the ACM*, 40(2):129–131, 1997.
13. Luqi and J. Goguen. Formal methods: Promises and problems. *IEEE Software*, 14(1):73–85, January / February 1997.
14. S. Marc, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, 1996.
15. N. Medvidovic and R. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering*, 26(1):70–93, 2000.
16. SUN Microsystems. Java Remote Method Invocation (RMI), version 1.3. Technical report, SUN, 2001.
17. OMG. The common object request broker: Architecture and specification, revision 2.2. Technical report, OMG, 1998.
18. OMG. Omg unified modeling language specification, version 1.3. Technical report, OMG, 1999.
19. OMG. Initial Submission to OMG RFP's: ad/00-09-01 (UML 2.0 Infrastructure) ad/00-09-03 (UML 2.0 OCL). Technical report, OMG, 2001.
20. OMG. Model Driven Architecture (MDA), Document number ormsc/2001-07-01. Technical report, OMG, 2001.
21. QNX. System Architecture Guide - QNX RTOS v6, 2002.
22. D. Quartel, M. van Sinderen, and L. Ferreira Pires. A model-based approach to service creation. In *7th IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, pages 102–110. IEEE Computer Society, 1999.
23. D. Regep and F. Kordon. **L***f***P**: a specification language for rapid prototyping of concurrent systems. In *12th IEEE International Workshop on Rapid System Prototyping*, June 2001.
24. J. Zhao. A slicing-based approach to extracting reusable software architectures. In *CSMR*, pages 215–223, 2000.

Pervasive Challenges for Software Components

Thomas Gschwind, Mehdi Jazayeri, and Johann Oberleitner

Distributed Systems Group
Technische Universität Wien
Argentinierstraße 8/E1841
A-1040 Wien, Austria
{tom,jazayeri,joe}@infosys.tuwien.ac.at

Abstract. Software components have been a long-standing dream of the software engineering community since the birth of software engineering itself in 1968. Every few years, it appears that we are on the verge of discovering exactly what a software component is. On the other hand, due to changes in technology and application environments and domains, our view of, and requirements for, components also changes. We review the changing nature of software components and discuss some of the challenges to the idea of components to be faced due to the advent of pervasive computing environments.

Keywords: pervasive computing, components, generic component model, component interoperability, dynamic adaptation

1 Introduction

The essential idea of software engineering is to systematically construct software out of parts that we now call components [17]. There are many definitions for components and the definitions evolve over time, sometimes due to new capabilities in programming languages, sometimes due to new programming or design paradigms, and sometimes due to an application domain requirement. Over time, components have referred to routines, abstract data types, objects, executable modules, etc.

Today, our agreements about the nature of software components are codified in the many available component models, including OMG's CORBA Component Model (CCM), Microsoft's COM+ and .NET, and Sun's JavaBeans and Enterprise JavaBeans.

Yet, as we are beginning to agree on what software components are, the emerging pervasive computing environment is set to once again challenge our notion of software components by introducing a completely new set of requirements for software components. In this paper, we review the evolution of software components and the general trends in the area. We present the Vienna Composition Technologies (VCT) as representatives of the state-of-the-art component and composition techniques. We then discuss the new pervasive computing paradigm and the resulting requirements imposed on software components. We present some ideas on how VCT may be changed to meet these new requirements.

The paper is structured as follows. Section 2 presents state-of-the-art component and composition techniques. In Section 3, we formulate the requirements

M. Wirsing et al. (Eds.): RISSEF 2002, LNCS 2941, pp. 152–166, 2004.

of component based software engineering in pervasive computing environments and in Section 4 we show two typical applications that benefit from a pervasive computing environment. On the basis of these requirements and application scenarios we discuss in Section 5 how VCT will have to be extended for pervasive computing environments. In Section 6, we present Work related to our approach and in Section 7, we draw our conclusions.

2 State of the Art in Software Components

Before discussing the role of software components in a pervasive computing environment we review the current state of the art of component technology. On the basis of this discussion, we can then define the requirements for the pervasive computing environment.

A *software component* is a unit of software that adheres to a well-defined contract defined by the interfaces it implements [18, 22]. These interfaces define the *features* a component provides. Such features may be methods, properties, events, or other communication and configuration mechanisms. A software component does not live on its own but is part of a *component model* that defines the basic architecture and characteristics of the components [4]. Run-time query capabilities reduce the limitations of earlier strongly typed models to allow dynamically defined interactions among modules.

The component-based software engineering paradigm is based on assembling (or composing) applications from independently developed components. On the other hand, independently-developed components may have incompatible interfaces in the sense that components that could interact with each other on a conceptual level may not be able to because of a mismatch in their interfaces. To solve this problem, different *adaptation techniques* have been developed. Example adaptation techniques are wrapping [11] and type-based adaptation (TBA) [8]. Wrapping is a manual adaptation technique that requires the developer to define the adaptation manually wheres TBA selects previously-written adapters from an adapter repository and automatically determines how the adapters need to be combined in order to translate between the components' interfaces.

We can say that the major characteristic of the current software component technology is components with well-defined interfaces. This characteristic supports composition of components based on an application's architectural description, and also automatic matching and possible adaptation of components.

As a concrete example, in the next subsection, we will discuss how the issues of component models, component adaptation, and architecture description are addressed by the VCTs. After that, we will review the requirements of pervasive computing and discuss how our solutions must be changed and adapted to meet the new requirements.

2.1 The Vienna Component Framework

One technology of the VCT is the Vienna Component Framework (VCF) [20], supports interoperability and composability of components from different com-

ponent models. It provides a single Java API that allows developers to reuse components implemented for different component models from within one single application. Since VCF abstracts from the internals of the component models, the developer no longer needs to deal with the difficulties and differences inherent to these models.

VCF uses a plugin architecture which allows developers to add support for a new component model through the implementation of a single plugin. Currently, we have implemented plugins for JavaBeans and Enterprise JavaBeans, Microsoft COM+, CORBA distributed objects and are working on plugins for SOAP, used for web service communication, the CORBA component model (CCM), and the Microsoft.NET framework.

Each plugin has to provide the access mechanisms to the features of its underlying component model. This includes features to control the life cycle of a component instance, features for accessing a component instance's state and using its operations and those to make a component instance persistent. How a feature is modeled and accessed internally is up to the plugin, but externally the plugin provides a standardized interface. For instance, a method feature exists in all component models and handling the abstraction was relatively by introducing a method interface. Events, on the other hand, have different concepts in different component models. Nevertheless, VCF allows the use of events with plugin implementations.

A single façade class is used to access the features exposed by the plugins. With VCF, components, such as COM+ or .NET components, that have no direct support for the Java programming become accessible.

2.2 Type-Based Adaptation

One of the key problems of building applications out of components is what to do if the component you need is unavailable in the catalogs you have. Clearly, no catalog will have every component that an application developer needs. But, often, there will be a related component, or one that is almost the one needed. One option for the developer is to modify the related component to make it fit his needs. This approach defeats the purpose of component-based development because for the fundamental reason that it breaks the separation of concerns between component development and component usage. A better approach would be to automatically adapt the components.

VCF abstracts the access to the features of components of different component models but does not try to adapt their interfaces. Such kind of adaptation, however, is necessary if one component provides the functionality expected by another component but provides an interface slightly different from the one expected. This kind of adaptation, however, is accomplished by TBA [7, 8].

TBA relies on a repository that stores previously built adapters and a meta-description of the transformations the adapters perform. The minimal requirements for this meta-description is the list of interfaces that an adapter is able to understand and those it is able to provide. TBA requires the ability to identify the required and provided interface during run-time. The required interface can

be identified on the basis of how the component is instantiated and the provided one using an interface repository which is provided by VCF and by modern component models. Once a collection of adapters is available, TBA adaptation exploits these features to automate the adaptation process by deciding when a adapters are needed and how they are to be applied.

More importantly, TBA can determine when it is necessary to chain several existing adapters together to effect an adaptation that is more powerful than any one existing adapter can do by itself. This ability to chain adapters together greatly increases the power of the process and requires many fewer adapters to be written by the programmer directly. We only have to define rules on when two adapters may be combined. In the simplest case, one adapter provides the interface that can be used by another adapter and hence, they may be combined. A detailed discussion on how more complex adapters can be built out of simpler ones can be found in [7].

The advantage of component adaptation is that a developer can provide a minimal catalog of components while the user still gains the benefits of a larger catalog. This was also the goal of the work reported in [15] but in a context that did not require the automated adaptation of software components.

2.3 Application Composition Language

To generically describe applications built from components, we have developed an Application Composition Language (`ACL/1`) [19]. This language allows for the explicit description of the application's architecture and its architectural properties. Similar to ADLs, our language provides constructs to describe components and their connections. The advantage of `ACL/1` over typical ADLs, however, is that it can be executed by means of interpretation or compilation.

`ACL/1` uses VCF for the representation of components and hence supports the composition of components that have been developed for different component models. ACL uses four different concepts: component descriptions, connector descriptions, configurations and frameworks. Components are the elements of computation in our language while connectors comprise the glue between components. Configurations describe how particular components and connectors are instantiated and arranged to make a concrete application. Frameworks are configurations in which several components and connectors remain unbound to concrete components or connector types. Frameworks are comparable to generics in object-oriented programming languages.

Unlike most ADLs, our composition language is not restricted to a predefined set of components or connector types but can be extended arbitrarily. `ACL/1` allows developers to provide new types of connectors or to define frameworks that act as generics for components or connectors. Our current language format is based on XML since XML parser frameworks are widely available and ready to use. Although XML can be easily analyzed with software tools human beings find XML difficult to read. Hence, we have built a graphical composition environment that can store, read and execute ACL descriptions [19].

3 Requirements of Pervasive Computing

The pervasive computing environment of the future will be characterized by many invisible sensors and computing elements, autonomously interacting with each other to dynamically construct and provide services to users that enter and leave the environment. Pervasive computing lies at the intersection of distributed systems, embedded systems, and telecommunications. Pervasive computing poses many challenges to software engineering and other disciplines. We certainly will have to revisit the ideas of components and component models, and how we should model, analyze, build, and deploy such components in a pervasive computing environment.

3.1 What Is a Component

The pervasive computing environment will force us to face the need for components and their boundaries more clearly. Indeed, complex applications will have to be composed from individual *components* residing in a large number of heterogeneous computing elements. The hardware environment itself will force a natural boundary between components. A component is an independently deployable piece of software that resides on one hardware element and provides a service element. Of course, there may be more than one component on each hardware element. Just as Web Services are emerging in the Web computing infrastructure as a component, some form of components will have to emerge in the pervasive computing infrastructure. Perhaps a component will also be a Web service, accessible through a URL? More likely, however, it will be a peer service with a well-defined protocol.

Component interaction mechanisms will also have to evolve. Traditional invocation based interaction mechanisms are probably inadequate to meet the needs of a large number of expected components and the wide heterogeneity of their interactions. Invocation mechanisms will have to allow for a much more loose coupling of components. Mechanisms such as event-based [1] interaction provide a higher degree of component independence and application scalability.

3.2 Need for a Scalable Component Model

Current component models are rather homogeneous in the kind of components they support. Components are basically of the same *size* and *power*. For example, JavaBeans components [10] are for desktop environments while Enterprise JavaBeans [4] are for server and enterprise-wide components. To make application development manageable, we need a component model that is *scalable* in the sense that it supports the development of components of various granularities, components that can reside in tiny computing elements, such as wearable computers, but can also grow to take advantage of resource-rich computing elements such as laptop machines and even larger fixed server machines. This is currently a real problem, even for a single programming language. Java has several versions of its JVM [24] for resource-poor and resource-rich environments.

You need to think differently about the component model if you are designing for a smart phone or for a laptop. In a scalable component model, at least the modeling and analysis techniques must span a range of granularities.

3.3 Need for Dynamic Adaptation and Composition

The pervasive computing use-case scenario envisions that the environment adjusts to new participants in the environment. It should not only provide services to the new participants but also use the resources offered by them to construct new services. For example, someone with a digital camera enters a room, automatically enabling a photo printing service, by combining the digital camera and the already-existing printing service. If another component exists that can convert photo resolutions, it is then possible to offer photo printing at various resolutions. If an image database exists, and photo album services are also available, new services can be dynamically configured.

This scenario implies that the computing infrastructure must be able to dynamically identify new software components to cooperate with and that they must be adapted to a constantly changing component catalog. This kind of ad hoc composition requires a higher degree of interface adaptation and mapping than is common today. Perhaps some components can provide interface *adaptation services* to enable a wide variety of component interoperation. Such adaptation services should probably be a standard but evolving part of the common infrastructure.

In a pervasive computing environment in which the catalog of available components changes constantly the adaptation that has to be performed is only known at run-time. Hence, an effective approach that automatically adapts the existing component to the need of the application is necessary. The VCTs deal with the concern of automatic adaptation with TBA. In pervasive computing, however, adaptation may also be necessary to adapt components to different devices, enabling components to move from device to device which is not yet addressed by TBA.

3.4 Need for Security and Access Control

An apparent serious conflict in the pervasive computing environment is the interplay between openness and security. A basic premise of the environment is that components dynamically recognize each other and cooperate with each other. On the other hand, such an open environment requires strict control over resources. You may not want to offer all available services to every new element that enters the environment. We need a lightweight mechanism because it has to be used heavily. A reasonable idea is that the application components carry "tokens" or capabilities that authorize them to access or provide services. Once an application component has obtained a token, the component uses this token for its authentication. The low-level communication mechanism is responsible only for secure communication among authorized parties.

The need to combine security and access control with dynamically interacting components of varying granularity may be the biggest challenge of component composition in the pervasive computing environment.

4 Two Scenarios of Pervasive Computing

The term pervasive computing is applied broadly and many visionary scenarios are offered as applications of pervasive computing. Applications have been described for the smart home [14], smart office [25], smart classroom [2], smart healthcare [23], and other such environments. The research also covers sensors and other hardware devices, networking, and software architectures. Here we describe two simple scenarios that exhibit the characteristics of pervasive applications. We concentrate on software components and do not refer to any particular embedded hardware devices. The purpose of these scenarios is to show that simple pervasive computing capabilities can indeed provide useful services, and that we can try to build the software infrastructure for supporting these services and experimenting with them without waiting for future embedded hardware to become available.

4.1 Meeting Manager

In today's globalized environment, it is common to have project or other kinds of meetings where the participants are from different countries and different organizations. The group meets in a hotel or company conference room. In a typical meeting, all participants own their own laptop or digital organizer. The participants may take turns to make presentations to the rest of the group. There are discussions about particular issues, decisions are taken, and action items are agreed upon.

Let us take a very simple example: agreeing on the next meeting for the project. The typical situation is that all pull out their electronic calendars and start checking for available dates. A very successful technique consists of one person writing their proposed dates on a blackboard and others crossing out some of those dates based on their current commitments as shown on their calendar. The hope is that some date will be free from all conflicts. This situation should lend itself to the pervasive computing paradigm: after all, all the relevant information is already stored digitally and all that is required is the seamless integration of the data and their coordination. In this sense, the pervasive computing hardware infrastructure already exists.

From the software point of view, if we imagine a meeting-manager application, the application faces the important pervasive computing requirements:

Heterogeneous components: The application consists of components that reside on heterogeneous devices (computer or organizer), each supporting a different calendar program; there is no possibility to limit the list of calendar components to only a single manufacturer.

Spontaneous and ad hoc networking: The components that form the application are not predetermined; they come together just because the owners happen to be part of this particular meeting; the components must be prepared to communicate and cooperate with a dynamically determined set of components.

Security administration: The calendar components should somehow be able to provide public information to legitimate components and protect private information from everyone.

We can imagine a simple controller application that is able to communicate with diverse calendar components, request information from them about particular dates and appointments, applying some algorithm to determine the best dates, and upon approval of the date, sending the accepted date to all calendars to update their individual entries. Where this application resides is an interesting question. It can reside on a server at the meeting conference room or it could be hosted by any of the devices that also has a calendar. Indeed, it can even float from one machine to another, as a piece of mobile code, depending on its design. The main point is that the application takes advantage of the pervasiveness of computing elements and the existence of all the information in digital and communicable form.

4.2 Museum Tour Guide

The meeting-manager scenario presents our view of pervasive computing applications: an application is a skeleton or description of a service whose implementation relies on components that become available, or are discovered, dynamically depending on the environment. In that scenario, the components were rather homogeneous in that they were all calendars. In this scenario, we consider many different kinds of components. Consider a museum-tour-guide application. A museum patron, equipped with a computing device enters a museum. The Tour Guide application designs and proposes a particular path for the patron to follow through the museum, on the basis of the patron's preferences and choices. As the patron walks by each painting, descriptions and information of particular interest to the patron is displayed on his organizer or spoken through his phone.

Again, the question of where the application resides is interesting. There could be tour and album organizer on the museum's server, which negotiates parameters of interest with the patron's computer. Or the application could be on the patron's computer, already customized for his preferences. Perhaps an album is already partially filled with data from other museums and only a few more items will be added from this museum. The common theme between this scenario and the previous one is that many components are involved: notepad, tour planner, and so on, and that these components may be offered by different computers that are dynamically determined.

5 Component Composition in Pervasive Computing Environments

As we have seen in the scenarios, the pervasive computing environment moves far away from the current component models (and long-term trends) that assume a mostly static environment. The pervasive computing environment is in fact characterized by volatility. Before we present how VCT need to be adapted, we discuss how we map components and component descriptions to the pervasive computing environment.

We assume a large number of devices, which we call pervasive computing devices. A component resides on a single pervasive computing device. Each component can provide functionality that is primarily of interest to the local device such as an image viewer component. Additionally, it may provide functionality that might also be of interest to other components probably residing on a different device. Such a service, for instance, could be a dictionary or a printing service. Since VCF is able to cope with desktop component models and server component models, it provides already a good fit for pervasive computing environments.

To build more powerful applications out of the available components components need to be composed. This can be done using `ACL/1`, our application composition language. In our current model, a single application description is available on a single device and is not distributed over multiple devices. An application description again can be seen as a component, with the exception that its functionality may be provided by components located on several different devices.

Application descriptions will have to communicate with many different components that provide the same kind of service but through different interfaces. Since these interfaces and the adaptations necessary, we need a powerful adaptation mechanism such as TBA. TBA has already been used for dynamic distributed systems which provide a similar environment but on a larger scale.

For this setup to work correctly, however, the following questions need to be answered:

- On which device should the application description be located?
- How can components be located on other devices?
- How should components be adapted? What types of adaptation exist?
- How do we deal with devices and components that enter or leave the environment?

To answer these questions, we imagine a pervasive computing environment as shown in Figure 1. The following sections discuss how this environment addresses the above problems.

5.1 Application Descriptions

The application server is intended to contain standard application descriptions in the particular physical location where this environment exists. For example,

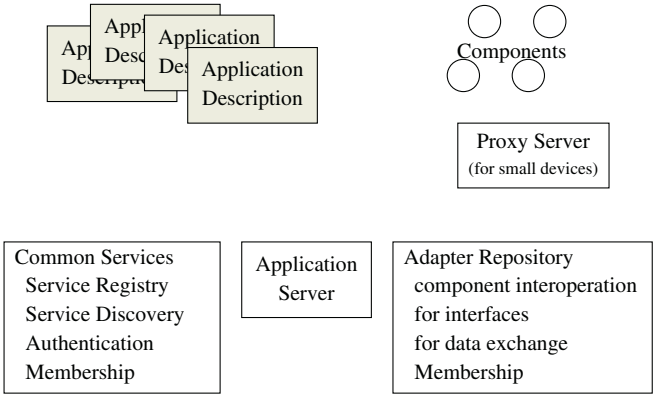


Fig. 1. Component-Based Pervasive Computing Environment.

it might run a meeting manager application or a museum tour guide application. The application descriptions and the components that form the application reside in different places and may move in and out of the environment.

We can identify two possible alternatives for where an application description can reside: either it is resident at a home location and uses the services or components that enter its neighborhood, or it may reside on one of the devices that enters the neighborhood. Both possibilities are viable and offer distinct advantages and design choices. In the meeting scenario, for example, the application description might be present on a notebook computer of one of the participants. In the guided tour scenario, on the other hand, the application description could be located at the museum's server and could be downloaded by its visitors into their PDAs.

We can see the difference between the pervasive computing environment and the current component technology by trying to imagine what application descriptions might look like. In conventional environments, the components that form the application and their interconnections are listed. In a pervasive computing environment the list of the components is not necessarily known and certainly the interconnections are not known. Therefore, the components must be identified mostly by their properties.

Our application composition language allows us to use both approaches. Firstly, it stores the names of the components that have been used for the application and secondly, it stores the features of the components that have been used. This allows the system to look up the component by its name and if that component is not available to use the features of the component as the list of requirements for a possible substitute component [9].

5.2 Locating Components

In a static computing environment, a central service could be used for the location of the available components. Due to the variety of different devices and

their mobility, however, this solution cannot be chosen. A possible solution is to use a distributed membership service where the devices register themselves as well as the components they offer. Since not every device is capable of providing such a service we propose the use of supernodes that maintain such a registry, an approach similar to that used in peer-to-peer networks [16].

A simple device may broadcast a request to locate the supernodes located within its vicinity. As soon as a supernode replies, it adds the node to its list of reachable supernodes and registers its own services with the supernode. Whenever the device needs to locate a component it queries the supernode.

5.3 Adaptation

Another element in our environment is *component adaptation*. Since we expect much more heterogeneity in the pervasive computing environment, we expect the need for many more component adaptation technologies than exist today. In a pervasive computing environment, the devices that need to interact with each other are constantly changing. Hence, components need to be able to interact with a wide range of components found on other computing devices. The framework we have presented in Section 2.1 is able to provide an environment to allow components implemented for different component models to interact with each other.

One drawback of VCF is that it still requires the interfaces provided by one component to match those requested by another component. In a pervasive computing environment such a component might not always be available but there might be a component providing the same functionality with a different interface. The adaptation has to be performed at run-time since the devices that need to communicate with each other and the communication protocols they use are not known a-priori. Since users of such devices do not want to deal with incompatibilities, the adaptation has also to be performed automatically.

One such adaptation technique, operating on the component's interface level, is TBA [7]. As explained in Section 2.2, this kind of adaptation is achieved with the use of an adapter repository. Indeed, we categorize adapters into many different classes, some of which are shown in figure 1. Adapters may be needed for adapting the interface of a component, or transform the data it provides to its clients, or the protocol it follows for providing the data.

We believe the organization of this repository, and the issues surrounding its population are particularly interesting. The organization of the adapter repository is key to the success of TBA. Without such a repository, adaptation is not possible. Since pervasive computing devices have a small memory capacity compared to desktop computers and since we cannot assume that these devices will have Internet connectivity, a central adapter repository cannot be used.

An interesting way to exploit component adaptation to deal with the volatility and unpredictability of pervasive computing is to maintain an adapter repository at each site. The idea is that each site would maintain a set of adapters that can be used to adapt the components available on its own device and hence increasing the number of components it is able to interact with. When a new

device enters the site, the adapters in the repository can be used to help it match the interface requirements of available services.

Such a setting makes it also possible for the new device to share its adapters with other devices available. This allows a device to download an adapter that could be useful for the device and carry it away. Other types of adapters may be used to control the behavior of incoming components to respect the requirements of the host environment, for example in terms of security. Yet other adapters may be used to provide more limited versions of existing services to untrusted arriving components.

5.4 Environment Volatility

Components and application descriptions *float* in and out of the environment, and sometimes compose to provide a service. Hence, pervasive computing environments carry the concept of partial failure to an extreme. At any moment, components of an application may disappear, as indeed may the whole application, leaving its components as orphans. Thus, a strategy is needed to deal with not only newly arriving components but also departing components.

Depending on the kind of the departing component, we have to deal with the failure differently. If the departing component does not maintain any state on behalf of the application, the infrastructure can deal with this kind of failure and look for a different component providing the same type of service. If the component, however, maintains part of the application's state, we have two choices: transferring the state before the component departs and maintaining checkpoints.

The transfer of the component's state is only possible if we can anticipate when it is likely for a component to become unavailable. If the devices use wireless communication, this could be identified using the signal strength of the device. In this case, we would have to transfer the state maintained by the device to another device. The representation of the state, however, needs to be converted to that used by the target device which again involves one of the adaptation approaches mentioned above.

If we cannot identify when a device leaves, or if the state of the departing device leaving cannot be used by any other devices left, the only chance is to revert to a previously stored checkpoint. Unfortunately, maintaining such checkpoints cannot be done in a general way, hence this issue has to be dealt with by the application itself.

5.5 Resource-Poor Devices

The heterogeneity of devices poses a challenge to the uniform integration of components and devices in an environment. On the one hand, we would like to treat all devices uniformly so that they can respond to queries and engage in interactions following similar protocols. On the other hand, many pervasive devices do not have sufficient power to process, or memory to perform such tasks. We, therefore, assume a proxy server at each physical site that represents small

embedded devices in the environment. These devices register or are registered at the proxy server along with their relevant properties to allow the proxy server to manage the environmental interaction with these small devices.

6 Related Work

Project Aura at Carnegie Mellon University (http://www-2.cs.cmu.edu/~aura/) is a very large project addressing the whole range of pervasive computing topics including speech recognition, wireless networking, and other areas. It is billed as *distraction-free ubiquitous computing* and its aim is to do as much as is possible automatically so that the user can concentrate on end-user tasks [6].

The software architecture envisioned by project Aura consists of three layers: a runtime layer that manages the environment, a model layer that maintains a model of the system architecture and ensures that needed resources are available, and a task manager that is responsible for performing user tasks. A user task, which is similar to what we have called application in this paper, is a collection of abstract services provided by the environment. The task manager monitors and optimizes resources for the use of the tasks while the model layer optimizes across the whole architecture in the environment.

Project Aura is focused on system aspects, while we are most concerned with component composition issues. They assume component interaction mechanisms to be built on top of available mechanisms such as provided by CORBA. The project deals with architectural adaptation in the sense of adapting to changes in the environment such as effected by user motion or departure of needed components [3]. This kind of adaptation is different from what we have described as type based adaptation. However, such adaptations are also important and it is interesting to see if it is possible to apply them as component adaptation techniques rather than architectural adaptations.

The Rome architecture [13] is based on context triggers. Triggers invoke the right services when the context requires it. For example, a driver is notified when he is driving near a supermarket that he needs to stop and pick up some new drinks for his party the next day. The context of the party and lack of drinks in the refrigerator are maintained in the infrastructure and appropriate actions are triggered when necessary.

A number of authors have discussed infrastructure services and designs for pervasive computing such as for a museum [5]. The Gaia operating systems [21] extends the services of a traditional operating system to a pervasive environment. An overview of software engineering research challenges in pervasive computing [12] includes a discussion of components and their requirements.

7 Conclusions

The ever-advancing hardware possibilities have necessitated fundamental changes in the way we design and build software systems. The major shifts in the past have been from centralized software to client-server to distributed. One major

shift appears to be towards pervasive software which is supported by a hardware environment that consists of large numbers of computing and communication elements. The environment is characterized by heterogeneity and volatility, introducing challenges and open problems in every software lifecycle phase: modeling, analysis, specification, validation, deployment, and additionally, security. Traditionally, software engineering solutions have tended to favor static descriptions and validation approaches. This tendency will have to be abandoned in order to deal with the highly dynamic nature of pervasive environments.

Independently of how we approach the software lifecycle phases, and how we structure the infrastructure, middleware, and applications in this new environment, one of the fundamental issues we have to face is the role of software components. We need to be able to compose services out of a highly ad hoc and dynamic set of potentially untrusted components. In this paper, we have presented the requirements for software components in a pervasive computing environment and defined a software component as a unit of deployment. We have reviewed the VCT as representatives of today's component composition environments. We have discussed how the basic component operations of definition, composition, and adaptation have to be modified in order to cope with the requirements of pervasive computing.

References

1. Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.
2. Alvin Chen, Richard R. Muntz, Spencer Yuen, Ivo Locher, Sung I. Park, and Mani B. Srivastava. A support infrastructure for the smart kindergarten. *IEEE Pervasive Computing*, 1(2):49–57, April–June 2002.
3. Shang-Wen Cheng, David Garlan, Bradley Schmerl, João Pedro Sousa, Bridget Spitznagel, Peter Steenkiste, and Ningning Hu. Software architecture-based adaptation for pervasive systems. In H. Schmeck, T. Ungerer, and L. Wolf, editors, *International Conference on Architecture of Computing Systems (ARCS'02): Trends in Network and Pervasive Computing*, volume 2299 of *Lecture Notes in Computer Science*, pages 67–82. Springer-Verlag, April 2002.
4. Linda G. DeMichiel, L. Ümit Yalçinalp, and Sanjeev Krishnan. *Enterprise JavaBeans Specification, Version 2.0.* Sun Microsystems, April 2001. Proposed Final Draft 2.
5. Margaret Fleck, Marcos Frid, Tim Kindberg, Eamonn O'Brien-Strain, Rakhi Rajani, and Mirjana Spasojevic. From informing to remembering: Ubiquitous systems in interactive museums. *IEEE Pervasive Computing*, 1(2):13–21, April–June 2002.
6. David Garlan, Dan Siewiorek, Asim Smailagic, and Peter Steenkiste. Project aura: Towards distraction-free pervasive computing. *IEEE Pervasive Computing*, 1(2):22–31, April–June 2002.
7. Thomas Gschwind. *Adaptation and Composition Techniques for Component-Based Software Engineering.* PhD thesis, Technische Universität Wien, February 2002.

8. Thomas Gschwind. Type Based Adaptation: An adaptation approach for dynamic distributed systems. In *Proceedings of the 3rd International Workshop on Software Engineering and Middleware*, volume 2596 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.
9. Thomas Gschwind, Johann Oberleitner, and Mehdi Jazayeri. Dynamic component extension to support cross-platform development. Technical Report TUV-1841-2002-19, Technische Universität Wien, March 2002.
10. Graham Hamilton, editor. *JavaBeans*. Sun Microsystems, http://java.sun.com/beans/, July 1997.
11. George T. Heineman and Helgo M. Ohlenbusch. An evaluation of component adaptation techniques. Technical Report WPI-CS-TR-98-20, Worcester Polytechnic Institute, Computer Science Department, March 1999.
12. Karen Henricksen, Jadwiga Indulska, and Andry Rakotonirainy. Infrastructure for pervasive computing: Challenges. In *Workshop on Pervasive Computing and Information Logistics at Informatik 2001*, September 2001.
13. Andrew C. Huang, Benjamin C. Ling, Shankar Ponnekanti, and Armando Fox. Pervasive computing: What is it good for? In *Proceedings of the ACM International Workshop on Data Engineering for Wireless and Mobile Access*, pages 84–91. ACM Press, 1999.
14. Stephen S. Intille. Designing a home of the future. *IEEE Pervasive Computing*, 1(2):76–82, April–June 2002.
15. Mehdi Jazayeri. Component programming: A fresh look at software components. In *Proceedings of the 5th European Software Engineering Conference (ESEC'95)*, volume 989, pages 457–478. Springer Verlag, September 1995.
16. Roman Kurmanowytsch. An overview of peer-to-peer topologies. Technical Report TUV-1841-2003-04, Technische Universität Wien, February 2003.
17. M. D. McIlroy. Mass produced software components. In *Proceedings of the Nato Software Engineering Conference*, pages 138–155, 1968.
18. Oscar Nierstrasz and Dennis Tsichritzis. *Object-Oriented Software Composition*. Prentice Hall, 1995.
19. Johann Oberleitner and Thomas Gschwind. Requirements for an architectural composition language. Technical Report TUV-1841-2002-20, Technische Universität Wien, June 2002. Presented at the 2nd International Workshop on Composition Languages.
20. Johann Oberleitner, Thomas Gschwind, and Mehdi Jazayeri. The Vienna Component Framework: Enabling composition across component models. In *Proceedings of the 25th International Conference on Software Engineering*, 2003.
21. Manuel Román, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, 1(4):74–83, October–December 2002.
22. Johannes Sametinger. *Software Engineering with Reusable Components*. Springer-Verlag, 1997.
23. Vince Stanford. Using pervasive computing to deliver elder care. *IEEE Pervasive Computing*, 1(1), January–March 2002.
24. Sun Microsystems. *Java 2 Platform Micro Edition (J2ME) Technology for Creating Mobile Devices*, May 2000.
25. Stephen Voida, Elizabeth D. Mynatt, Blair MacIntyre, and Gregory M. Corso. Integrating virtual and physical context to support knowledge workers. *IEEE Pervasive Computing*, 1(3):73–79, July–September 2002.

Model Generation for Legacy Systems

Hardi Hungar, Tiziana Margaria, and Bernhard Steffen

University of Dortmund
{Hardi.Hungar,Tiziana.Margaria,Bernhard.Steffen}@udo.edu

Abstract. We propose the use of (semi-) automatically *extrapolated models* as a means for coping with legacy systems: a focused way of testing systems for their behavioral properties allows the construction of expressive behavioral *hypothesis models*, and therefore extends the range of formal methods to 'black box' scenarios, which are dominant in industrial practice. Keeping these models up to date by continuous adaptation may provide an ideal way for controlling the evolution of large systems during their whole life cycles. Bottleneck of this approach is the size of the extrapolated models: particularly for distributed systems the *state explosion problem* strikes back. This paper focusses on a particularly promising cure: *view-oriented* model construction allows a new way of size control that complements other powerful techniques, which together have the potential to scale to systems of realistic size. This is illustrated by considering *small instance* views in the context of Computer Telephony Integrated Systems.

1 Motivation and Background

Models are the key for the design of complex systems, accompany their development, and serve as powerful means during the maintenance. In fact, without them, the interaction with a potential customer remains vague, and there is no sound basis for a contract. This commonly agreed upon statement has however little practical impact, because - despite the recent developments like the Unified Modelling Language (UML) - building models, and even worse, keeping them up to date, is time consuming and expensive, thus not adequately taken care of. The lack of adequate, up-to-date models indeed causes misunderstandings, leads to unexpected side-effects in new releases and is the reason for many unsatisfied customers. The following three subsections summarize this situation with special attention to legacy systems, which, in fact, are dramatically dominant in the industrial practice, and propose a pragmatic way out: automating the bulk of the model construction and maintenance effort by exploiting techniques from automata learning. In fact, (semi-) automatically generated models (almost) 'for free' seem to provide a good compromise in the precision/cost trade-off. Once this technology is accepted, deeper modelling techniques, like those e.g. aimed at with UML, may be easier to introduce into current design practice.

M. Wirsing et al. (Eds.): RISSEF 2002, LNCS 2941, pp. 167–183, 2004.

1.1 Status Quo and Needs

Component-based software engineering as supported by CASE tools, portable programming systems (like those centered around Java), and universal modelling tools like those based on UML have changed the pragmatics of modelling and programming environments. The new tool support has successfully enabled a wealth of applications that now ease - or at least accompany - our daily life. However, there is still a long way to go before the promises of modern software engineering become everyday reality. This concerns at least the "popular" expectations, easily excited by slogans like "write once, run everywhere" (which is true only once you have the adequate virtual machine, which might be cumbersome in practice), or by the hope-belief that complexity is tamed by reusability, and that reusability comes for free as soon as one follows a few lightweight rules of thumb. Even worse: there are still major problems to solve, which seem to be out of reach of the state-of-the-art paradigms. Examples are:

- **Customer integration:** how can we build models that are comprehensible to the customers, but at the same time sufficiently precise so that they can constitute a good basis for design and implementation? Or, even better, how can we integrate the customer more closely into the design and implementation process? Extreme Programming is a functioning solution, but it does not scale.
- **Continuous process:** how can we extend and improve the usability of the models and modelling techniques used for design and implementation so that they also become a practical means of reference for the (sometimes extremely long lived) expensive maintenance phases? This must happen so that these models become an accepted support rather than a burden in the continuous release evolution.
- **Legacy systems:** how can we deal with existing, "historically grown" systems, which typically lack an underlying formal model? This aspect gains ever increasing importance as complex industrial systems typically depend on third party-subsystems, for which only APIs and user manuals are available. Moreover, most industrial systems too soon become de facto legacy systems, since it is usually impossible to maintain and update the documentation and modelling of the development over the many release cycles.

1.2 The Legacy Problem

These problems are gaining increasingly vital importance, and they will reach be increasing criticality in the future. Some reasons are easily listed:

- If the specification is wrong, the whole ongoing process of design, implementation, release is deemed to eventually fail or, at least, the required adaptations become extremely expensive. Thus we need to be able to check specifications directly with the customers in order to validate their real requirements as early as possible. Unfortunately, use cases as found in UML

are not expressive and precise enough, and almost all available and successful formal modelling methods and languages (like e.g. Petri Nets, State Charts, SDL, MSCs or language like OCL and Z and tools like the B tool) are typically by far outside the skills of a customer.
- Systems become increasingly complex, thus it is impossible for a single person to know all their details. Thus improved support by means of adequate models is needed, in order to capture and communicate all the required information on various levels of abstraction. More concretely, we need modelling tools that capture *global* aspects, and that support refinement, aspect-oriented concretization, and checking, as well as several ways of dealing with syntactic and semantic concepts of abstraction and hierarchy. This is particularly crucial for the maintenance phase, where good models are ideal means for understanding the effects of upgrades or the localization of errors.
- Modelling is needed in particular for already existing systems. Most systems in use today lack adequate specifications or make use of un(der)specified components. In fact, the much propagated component-based software design style naturally leads to produce under-specified systems, since most component specifications are quite partial. In these cases, global aspects need to be modelled on a higher level, which does not require implementation details. We may think e.g. about the software that implements real-time inter-bank money transfers: it requires the cooperation of middleware, operating systems, databases, ERP systems and a huge number of specific software packages in a significant number of banks in order for a payment to be effectuated correctly around the globe. In cases like this the global aspects should be specified on the component-coordination level, while suppressing as much detail about the involved components as possible. In fact, this global modelling should not even try to pinpoint errors within the various components, but rather concentrate on their interplay.

To attack the problems above we propose an observation-based approach. This approach is characterized by its global perspective (observations are taken at the system level), by its view-based construction (models are constructed according to device or aspect-specific projections of the global behavior), and, perhaps most importantly, by its high potential for automation, that makes it particularly valuable for capturing the evolution of a system during its life cycle.

In the following, after a brief overview of our approach (in Sect. 2), we sketch the contributions of this paper (Sect. 2.4), which in particular indicate its practicality. We then present our application scenario in Section 3 and sketch the foundations of our approach in Section 4. The main discussion is entered in Section 5, which establishes our view-based approach for treating parameterized systems. Finally, we draw our conclusions in Section 6.

2 System-Level Observation-Based Model Construction

Our observation-based approach is intended to support the continuous-engineering process along all the phases of a system's life cycle. Its focus is therefore on

easy, user-level abstractions of the global system behavior, non-invasiveness of the technology, and sensitivity to the system changes during the life cycle. This lead to the following profile:

2.1 (Semi-)Automatic Model Construction

Using techniques from automata learning, combined with automata-theoretic and logic-based approaches, it is possible to *extrapolate* expressive models of legacy applications without any access to source code or formal specifications [6,5]. These extrapolated models are typically neither sound nor complete, but they serve as a powerful means for consistently presenting all the knowledge available about the system. Typically, this is used in practice to capture and communicate the *user-level knowledge*, which is on the one hand very abstract, but on the other hand the typical knowledge (still) available for legacy systems. This approach is ideal for accompanying the software life cycle or even steering the systems' evolution, as the quality of the extrapolated behavioral models increases during the systems' life cycle. As most of the model construction is automatic, extrapolated models can be kept up-to-date at very low cost.

2.2 Interleaving Learning and the Use of Extrapolated Models

Extrapolated behavioral models provide a solid basis for runtime monitoring, test generation, test coverage evaluation, test result analysis and, in fact, for automating large portions of regression testing. Taken together, these are the major cost factor for the release and maintenance of industrial systems. Besides profiting from the information captured in the model, model-based testing and monitoring of running systems (see e.g. Sect. 5) can also be regarded as a means to implement *model evaluation*: whenever an observation is in conflict with the extrapolated 'hypothesis' model, experts need to be consulted in order to decide whether the system or the model needs to be altered. Both, the system correction, as well as the modification of the model along the observed discrepancy automatically improve the system/model relationship. Thus by design our approach enhances and steers the software maintenance phase without imposing significant additional effort.

Please note, however, that our approach addresses only the analysis of (legacy) systems. Their correction and evolution is an orthogonal matter and should be done by the corresponding responsible teams.

2.3 Many Very Abstract, Aspect-Specific Modelings

Adequate levels of abstraction make observations directly comprehensible for customers, both for validating the system requirements and for judging the impact of conceptual system changes [10]. In fact, various such levels are typically required in practice in order to capture different views onto the system, since e.g. different people get different projections of the overall behavior: the customer

needs the user view, the data security expert the corresponding protocol view, and the quality controller needs specific views that capture exception handling, warnings, and all kinds of potential misbehaviors.

Using different abstraction mappings during our extrapolation procedure, we automatically obtain view-specific models of the global system. In our applications these views reflected projections onto the context-specific behavior of devices (which can easily be identified at the protocol level) and device-overlapping aspects like payment or exception handling. More complicated are abstractions that project systems with n devices onto systems with a much smaller number of devices. The paper will focus on this scenario (see Sect. 5).

2.4 Technical Contributions of This Paper

As described above, the main characteristics of our approach are ensured by design. But how realistic is this design in practice? A proof of concept has been given directly on the basis of our integrated testing environment (ITE), which provides the technical means required during the extrapolation process, as well as for monitoring the running system. But does our approach scale? In this paper we discuss scalability in the context of parameterized systems. The considered setting stems from an industrial project (see Sect. 5), where the treatment of large numbers of devices in a distributed telecommunication system was of vital importance.

3 Observing and Testing Distributed Systems

Modern telecommunication and IP-based applications are multi-tiered, distributed applications that typically run on heterogeneous platforms. Their correct operation depends increasingly on the interoperability of single software modules, rather than on their intrinsic algorithmic correctness. A scalable, integrated test methodology capable of granting adequate coverage of functional testing, yet still easy to use has been presented in [12]. This methodology bases on our previous work on system level test of Computer Telephony Integrated (CTI) and Web-based applications, where we applied very successfully our coordination-based coarse grain approach to modelling and design of complex distributed systems [13].

To test complex cooperating systems of the kind shown in Fig. 1 we add a Test Coordinator to the system under test: the test coordinator is an independent system that drives the generation, execution, evaluation and management of the system-level tests in highly heterogeneous landscapes. This introduces the required flexibilization of the overall architecture of the test environment: it leads to a modular and open environment, so that diverse test tools and units under test can be added at need. The Test Coordinator has access to all the involved subsystems and can manage the test execution through a coordination of different, heterogeneous test tools. These test tools, which locally monitor and steer the behavior of the software on the different clients/servers, are technically

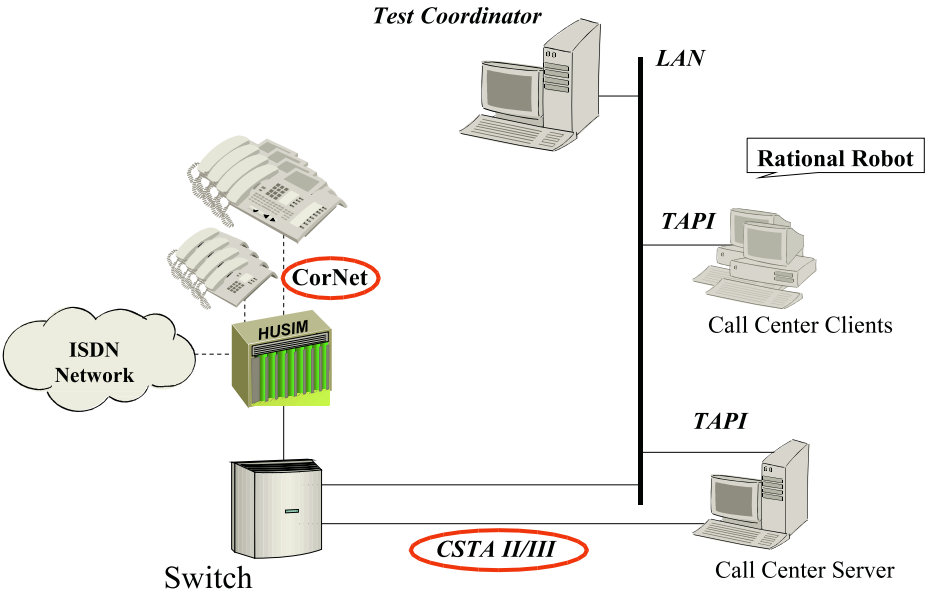


Fig. 1. Overview of the system to be learned and its environment.

treated just as additional units under test. The resulting test environment, called ITE (Integrated Test Environment [7]) has been successfully applied along real life examples of IP-based and telecommunication-based solutions: the test of a web-based application (the Online Conference Service, used e.g. for the support of the Program Committee operations of four ETAPS 2003 conferences) and the test of IP-based telephony scenarios (e.g. Siemens' testing of the Deutsche Telekom's Personal Call Manager application [11], which supports among other features the role based and web-based reconfiguration of virtual switches). In both cases, no fine grained formal model of any subsystem was available, but a rich collection of expressive test cases - adequate for applying our model generation methodology - was available already a short time after the adoption of the ITE test environment.

In this paper, we show the use of our model generation technique by applying it to the system from Fig. 1 which we briefly describe (details can be found in [6]). The system's main part is a midrange telephone switch which is connected to the ISDN telephone network and acts as a 'normal' telephone switch to the phones. Additionally, the switch communicates directly via a LAN or indirectly via an application server with CTI applications that are executed on PCs. Here in essence two communication protocols are used: *CorNet* for the communication between the switch and its peripheral devices (phones), and *CSTA/Tapi* for the communication between the switch and the CTI client/server applications that communicate with the switch over LAN. Like the phones, the CTI applications are active components: they may stimulate the switch (e.g. initiate calls), and they also react to stimuli sent by the switch (e.g. notify incoming calls).

As just introduced in the general case, also in this concrete setting the technical realization of the necessary interface to this System Under Test is provided by the *integrated test environment* ITE, and in particular it is coordinated by the *Test Coordinator*, which – among performing other functions – controls two different test tools, i.e. a test tool to control the telephone switch (*Husim*) and one for the interaction capabilities of the CTI application (*Rational Robot*)[1]. In our experiments [9] we have derived several models of the switch with the help of this interfacing machinery.

4 Foundations of Model Extrapolation

In this section, we sketch the cornerstones of our regular extrapolation approach to model construction. In particular, this comprises the concepts of *abstraction*, *learning*, and *moderation*. The following subsections address these issues using the previously described *Computer Telephony Integrated (CTI) system* (Fig. 1) for illustration. A more detailed account of these concepts can be found in [18].

4.1 Abstraction

We use abstraction to reduce the models we want to construct to a manageable size. The main concern must be to achieve a level which retains useful information about the system while permitting automatic analysis and modification techniques to be applied. We chose to model on a propositional level, including relevant control aspects and causal dependencies of the system into the model while leaving out data and real-time aspects. This works particularly well for telephony systems, whose behavior usually does not depend on *what* is transmitted, but *how*, and which are rather loose in their timing constraints.

Let us consider our example system which employs an instance of the *Computer-Supported Telephony Applications (CSTA)* protocol for some of its communications. A typical CSTA record contains several components. Some of them convey essential information relevant to the model, while others can be safely ignored. For most modelling purposes it will be sufficient to project such records to something as abstract as, for instance,

```
(hookswitchOnHook,500)
```

where `hookswitchOnHook` denotes the event that a telephone receiver has been replaced, and `500` identifies the phone device.

This 'hiding-based' reduction is the first step towards a propositional view of a system's behavior. However, more refined abstraction techniques are required for our approach to work in the considered scenario.

[1] This is described in more detail in [7].

Event Separation. The systems operate in real time in an environment that exhibits much parallelism. For example, when observing a CTI system in operation, interactions belonging to different conversations will typically overlap. Permitting such interleavings in the model would obfuscate the really essential dependencies. Instead, we collect all the system's reactions to each single stimulus by waiting until the system has reached a stable state. This is similar to the point of view taken in the synchronous systems approach [2,8], and it is also the approach typically taken as the basis for testing. In particular, it was the approach taken by our industrial partner.

Based on this notion of event separation, we arrive at a view of a reactive system as an input/output transition system whose outputs are determined by previous inputs (an *input-deterministic* I/O transition system according to the following definition). It may also be assumed that the system is *input enabled*, i.e., that it accepts all inputs regardless of its internal state.

Definition 1.
An input/output transition system *is a structure* $\mathcal{S} = (\Sigma, A_I, A_O, \rightarrow, s_0)$*, consisting of*

- *a countable, non-empty set* Σ *of states,*
- *countable sets* A_I *and* A_O *of input, resp. output, actions,*
- *a transition relation* $\rightarrow \subseteq \Sigma \times A_I \times A_O^* \times \Sigma$*, and*
- *a unique start state* s_0*.*

It is called input deterministic *if at each state* s *there is at most one transition for each input starting from that state. It is* input enabled *if there is at least one transition for each input.*

Small Instantiations. Another property indispensable for our approach is the finiteness of the model. Usually, the number of components connected to a system like the telephone switch might be rather large, thus modelling a system with the maximal number of components (if known) would be impractical. Also, a new release might increase this parameter, thus invalidating the previous model. Such a large model would also not reveal much additional information about the system. In fact, both protocol specifications and practical tests usually work with small, finite instantiations of a system environment. We do the same, and thereby arrive at a manageable system size where address spaces can be represented by few discrete symbols. In Section 5, which discusses the use of models for monitoring running systems, specific issues concerning the reduction to few components are presented.

4.2 Learning

Techniques from the field of *automata learning* were the guideline when implementing our regular extrapolation approach. Generally speaking, automata learning tries to construct an automaton matching the behavior of a given target automaton based on observations of the target automaton and perhaps some

further information on its internal structure. We will not discuss the theory of learning here (the interested reader may refer to [18] for our view on and use of learning). Instead, we present the algorithm L* from [1], which we then adapt to fit our purposes.

L* learns a finite automaton by posing membership and equivalence queries. A membership query tests whether a string is contained in the target's language, and an equivalence test compares a hypothesis automaton with the target and, if they are not equivalent, it returns a string in the difference of the accepted languages. The basic idea behind L* is to systematically explore the system's behavior using membership tests, and resorting to equivalence tests whenever no 'direct evidence' for inconsistencies is available. On this basis, L* incrementally builds a minimal deterministic finite automaton (DFA) equivalent to the target, in a time which is polynomial in the size of the target and the counterexamples.

Adapting L to Our Scenario.* In our scenario, we rely on being able to evaluate membership queries with our testing machinery (more on that later). Equivalence queries are beyond practical realizability, but already [1] contained the result that by replacing precise equivalence tests by approximative sets of membership queries, L* can be turned into an algorithm which learns in an approximative sense: with high probability the learned automaton will accept a language very close to the given one, and this result will again be reached in polynomial time in the size of the target. We have adapted it in a different, but similar way [9] to arrive at an adequate approximate equivalence test based on membership queries.

An interesting technical point is the fact that L* is constructed to learn DFAs that accept regular languages, whereas our scenarios are modelled best as (combinations of) I/O transition systems. The assumption of input determinism makes it possible to translate membership tests for certain strings to the application of input sequences to an I/O transition system. All further adaptations and optimizations become technicalities, though of sometimes major practical importance for the consumption and management of time and resources.

Realizing Membership Tests. Membership queries can be answered by testing the system we want to learn. This is not easy, because the sequence to be tested is an abstract, propositional string, but the system is a physical entity whose interface follows a real-time protocol for the exchange of digital (non-propositional) data. Thus we have to drive the system with real data, which requires to reverse the abstraction and produce a concrete stimulation string.

In practice, we need a concretization function where things once abstracted from the observations have to be filled in dynamically, taking the reactions of the system and consistency criteria into account. For instance, time stamps have to increase, and instead of symbolic addresses and symbolic tags their concrete counterparts have to be used consistently. Finally, these data must be transformed into communication signals and fed to the real system.

In the case of our example telephone switch, all this is done via our testing environment, the already mentioned ITE. ITE performs these tasks using

specific hardware as well as predefined code blocks for generating stimuli and for capturing responses and glue code, which together solve the problems connected with generating, checking and identifying the non-propositional protocol elements. Thus much of the work of putting our approach into practice relies on the preexisting capabilities of the ITE system and on its diverse components.

Realizing Equivalence Tests. Though there is no exact equivalence test (and in fact, our extrapolated models will only rarely be equivalent to the abstracted system in a strict sense), one can approximate such a test pretty well in practice: the basic idea is to scan the system in the vicinity of the explored part, looking for discrepancies from the expected behavior.

One particularly good approximation is achieved by performing a test in the spirit of [3,19]. This test is based on first computing a set of stimuli sequences by which all states in the conjecture automaton can be distinguished. Then, each transition in the conjecture automaton is validated by checking that, after moving to its start state and then firing it, the outcome in the real system behaves as it should with respect to all relevant distinguishing sequences. This test has polynomial complexity in the size of the conjecture automaton. Another possibility lies in checking consistency within a fixed lookahead from all states. In the limit, by increasing the lookahead, this will rule out any uncertainty, though at the price of exponentially increasing complexity.

Further substitutes of an equivalence tests are derived with the help of expert knowledge. We describe these in the following section on moderation, since they involve human interaction.

4.3 Moderation

Moderation is the process of matching automatically generated models and human concepts about the system behavior, with the behavior of the real system, and resolving any discrepancies.

System Constraints. One point of interaction consists in the evaluation of whether a tentative model produced by the learning algorithm is acceptable, i.e. deciding whether the equivalence check of L* is passed or fails. The adequate incorporation of expert knowledge requires formalization. Here, (linear) temporal logic [4] serves to formulate specifications which limit the model from above. I.e., experts formulate properties which they believe to be true of the system, and extrapolation results should be limited by them. Temporal-logic model checking is employed to check adherence of the model to these constraints. Counter examples generated by the model checker in case of a violation are studied to pinpoint the source of the discrepancy.

First, it is checked whether the counter example of the model (which is a trace of the model not consistent with a constraint) is also a counter example for the real target system. Two outcomes are possible:

1. If the observed system behavior is consistent with the hypothesis automaton, a discrepancy between a constraint and the system has been detected. Thus either the system itself has an error, or the specification is wrong. This has to be resolved manually, i.e. by consulting experts for the system or the application domain. If the error can be attributed to the specification, its correction is easy: the specification is corrected (or dropped), and the learning process can continue.
   If it is a system error, we can report success in detecting a real problem as a side-effect of the model generation. In this case, the error in the system should be corrected and the model subsequently adapted accordingly.
   While the correction of the model is usually easy, the correction of the real system is not always immediate: especially in case of third-party/legacy components the error or discrepancy cannot be fixed directly, but must be reported to a different team or company that is in charge of maintaining the source code.
2. If the observed behavior of the target system deviates from that predicted by the hypothesis automaton, this trace is a counter example as expected from an equivalence oracle during the learning process. The learning procedure automatically takes the appropriate actions to improve the hypothesis model.

5 Treating Parameterized Systems

Regular extrapolation has turned out to be particularly well suited for application in regression testing, where a previous version of the system is available as approximate reference. Learning the behavior of previous versions therefore allows a cost-effective and flexible testing of new versions (cf. [7]): rather than running the reference system and the new version in parallel while comparing their results, which requires

- two complete test systems,
- a (typically) bit-wise comparison of test results at the protocol level, and
- an expensive analysis of the detected discrepancies - which, to our experience are mostly false alarms due to the too tight way of comparison

extrapolated models

- can easily run in parallel with the system under test, without requiring any specific hardware,
- allow for a much more flexible notion of comparison, e.g. by accepting any equivalent interleaving of events within the distributed system as a match, and
- drastically reduce the analysis of discrepancies, firstly, because the number of false alarms is significantly reduced, and secondly since the model will typically provide certain diagnostic information (e.g. error and warning states).

In the following, we will focus on learning concise models (in fact specific views) that capture parameterized systems of this kind. The ideas we present do not

exclusively apply to this situation: another very important area of application is e.g. system monitoring (also often called online testing), which requires exactly the same kind of models. It should, however, become clear that the structure of extrapolated models may significantly depend on the intended use, and that certain optimizations, like the ones presented here, cannot always be applied.

Parametric Reference Systems

We consider parametric systems consisting of a central component which is connected to a (parametric) number of structurally interchangeable peripheral components, and use A^∞ to denote the set of finite and infinite sequences over the alphabet A. In order to be able to focus on the essence, we will refrain from the technical details concerned with input/output transition systems.

Definition 2 (Parametric System). *A* parametric system $P(n)$ *consists of* $n+1$ *components* $C_0, \ldots, C_n$*, where* $C_1, \ldots, C_n$ *are called* parametric*. Given a finite set* T *of message denominators,* $T \times \{C_1, \ldots, C_n\}^k$ *for some* $k \in \mathbf{N}$ *is the set of* communication records[2]*. The semantics* $[\![P]\!]$ *of a parametric system* P *is given in terms of a prefix-closed subset of the set of traces of the corresponding communication records* $(T \times \{C_1, \ldots, C_n\}^k)^\infty$*.*

For a parametric system $P(n)$ *and* $m \leq n$ *let* $P(m)$ *denote a system with* $[\![P(m)]\!] = [\![P(n)]\!] \cap (T \times \{C_1, \ldots, C_m\}^k)^\infty$*, i.e. the set of traces of* $P(n)$ *mentioning only the components* $C_1, \ldots, C_m$*.*

A parametric system P *is called* symmetric*, if for each permutation* ρ *of* $\{C_1, \ldots, C_n\}$*, we have* $\rho([\![P]\!]) = [\![P]\!]$*, where* $\rho([\![P]\!])$ *is the obvious extension of* ρ *to sets of traces of communication records. I.e., the semantics of* P *is closed under permutations.*

We use parametric systems to represent the behavior of systems, abstracted to a propositional level, where only the number of peripheral components (like telephones or clients) remains variable. C_0 corresponds to a central component (like the switch or a server) to which all the peripheral components are connected.

Small Instance Views

The point of our technique is to learn a model of a symmetric parametric system capturing its behavior for **small** instantiations (*small instance view*, and to use this view for testing and monitoring. More concretely, when learning a small instance of a system $P(n)$, we build a model for $P(m)$ for some $m \leq n$. This model has the specific form of a DFA accepting a prefix-closed language of potentially infinite strings.

Definition 3 (m-Model). *Let* $P(n)$ *be a parametric system with traces over* $T \times \{C_1, \ldots, C_n\}^k$*. For* $m \leq n$*, an* m-model for $P(n)$ *is a DFA* M *(or, explicitly,* $M(m)$*) with alphabet* $T \times \{D_1, \ldots, D_m\}^k$ *(*called observation records*) where all states except one are accepting and the non-accepting state is a sink. The* language *of* M*, denoted by* $[\![M]\!]$*, is the set of all finite and infinite alphabet*

[2] k is the maximal number of identifiers occurring in a protocol message.

sequences labelling transitions on paths in M which start in its initial state and do not enter the non-accepting state.

Let μ be the function that maps C_i to D_i, for $i = 1, \ldots, m$. The model $M(m)$ is correct *iff $\mu([\![P(m)]\!]) = [\![M(m)]\!]$.*

Learning $M(m)$ is possible due to the *active* learning approach taken by L*, which can be directly steered according to the chosen set of components as long as it is possible to identify these components by observation. In our application scenario this information is fully available in the communication records. Thus, even though the system under test is really huge (it may consist of potentially hundreds of parametric components), the learning process focusses just on the part relevant for the considered small instance view.

Assuming that the parametric system is symmetric, a correct model does indeed provide complete information about all traces with at most m involved parametric components:

Proposition 1 (Symmetric Correctness). *Let $M(m)$ be a correct model for a symmetric, parametric system $P(n)$. Then $M(m)$ is itself symmetric and for all subsets $CS = \{C_{i_1}, \ldots, C_{i_m}\}$ of $\{C_1, \ldots C_n\}$ and every one-to-one mapping ν from CS to $\{D_1, \ldots, D_m\}$ it holds that $\nu([\![P(n)]\!] \cap T \times CS^k) = L$, the language of $M(m)$.*

Small Instance Views in Use

To construct a monitor from a symmetric model, we equip the monitor with two state components to hold

- the current state of the model, and
- an assignment of identifiers of peripheral components of the system being observed to the component identifiers $D_1, \ldots D_m$ within the model.

This construction is the basis for *matching* the concrete traces of the system with traces of the model.

Definition 4. *Let $M(m)$ be an m-model for a parametric system $P(n)$. An* observation state *is a pair (ν, s) of a partial one-to-one mapping ν from component identifier in the model $\{D_1, \ldots, D_m\}$ to concrete components of the system $\{C_1, \ldots C_n\}$ and a state s of M.*

Let now ν be extended to observation records in the straightforward way. A communication record c of $P(n)$ is matched *by a pair of observation states (ν, s) and (ν', s') if ν' is an extension of ν and there is a transition $s \xrightarrow{b} s'$ of M such that $\nu'(b) = c$.*

A trace τ of $P(n)$ can be matched by M if there is a sequence (ν_i, s_i) of observation states of the same length as τ where

- *s_0 is the initial state of M and ν_0 is the empty mapping, and*
- *for each i, the pair consisting of (ν_i, s_i) and its successor matches the observation record τ_i.*

If a model M is symmetric, the check whether a trace is matched can be done easily online: the mapping ν is extended only if needed, the choice of the minimal possible extension is arbitrary due to symmetry, and once the extension is chosen there is at most one transition due to the determinism of M.

Theorem 1 (Correct Matching). *A model $M(m)$ of a symmetric parametric system $P(m)$ is correct if and only if the set of traces matched by $M(m)$ is the subset of traces in $[\![P(n)]\!]$ where at most m parametric components appear.*

Thus correct small instance views (m-models) mimic the system under test perfectly well as long as the number of components encountered is smaller than m.

Small Instance Views: Where to Go

The ultimate goal of our approach is to make it effective as long as the number of components being concurrently active in the system under test does not exceed the number modelled in the small instance view.

Example
Between independent usages, a phone typically returns to its initial state, at which it is indistinguishable from the behavioral point of view from all the other inactive phones. This offers the possibility to change the matching between inactive concrete components on demand, and therefore to increase the monitoring power of a parametric model $M(m)$, in order to capture all traces where no more than m phones are active *at the same time*.

This 'dynamic scheduling' of symbolic components requires the identification of those points, where one concrete device (e.g., phone) can be exchanged by another, which is difficult just on the basis of observations. For our application this boils down to decide at the model level that a component (phone) is inactive, resp. is in its initial state.

Assuming that our model is correct, we have a criterion at hand: a component D_i can be considered to be inactive in a state s of a model, if s is reachable in M from the initial state without taking any transition mentioning D_i. This directly leads to the following definition:

Definition 5 (Activity of Components). *Let s be a state in a model $M(m)$ of a parametric system $P(n)$. A component $D(i)$ is said to be* inactive in s*, if there exists a path from s_0 to s s.t. D_i does not occur in any transition label on the path. Otherwise, the component is said to be* active in s*.*

Basing the notion of matching on active components only, we straightforwardly arrive at the following notion of liberal matching.

Definition 6 (Liberal Matching). *An communication record c of $P(n)$ is* liberally matched *by a pair of observation states (ν, s) and (ν', s') if ν' is an extension of the restriction of ν to the components active in s, and there is a transition $s \xrightarrow{b} s'$ of M s.t. $\nu'(b) = c$.*

A trace is liberally matched *if there is a liberal matching sequence of observation states.*

Obviously, the set of traces liberally matched can be much larger than the set of (strictly) matched traces. Therefore, monitors constructed accordingly will be able to follow the behavior of an observed system much longer and, to our experience, very accurately. Unfortunately, as will be explained in the next subsection, our formal conditions are not sufficient to justify the generalization of Theorem 1 from *matching* to *liberal matching*.

Limitations of Liberal Matching

The reason for the failure of the generalization of Theorem 1 from *matching* to *liberal matching* lies in the fact that the central component may, in principle, behave history dependently in arbitrary way, as e.g. in the following situation:

Consider a system $P(8)$ which, whenever three *different* peripheral units have tried one specific action (and have been denied permission to perform it), will enable that action. A correct model $M(2)$ will never permit that action, because it will never encounter a situation where three *different* peripheral units have tried this specific action. This does not change when moving to liberal matching mode. Thus the according monitor will remain too strict.

Similarly, in the dual situation, where an action is prohibited after some history, the according monitor may be too loose. Thus none of the implications of Theorem 1 can be established in this more general setting. Still, as discussed in the final section, this does not affect our overall goal, which is anyway bound to deal with the 'vagueness' of hypothesis models.

6 Conclusion and Perspectives

We have presented an approach for view-based model construction, which aims at empowering moderated extrapolation [6] as a strong means to support the development, construction and maintenance of complex industrial systems. This approach is characterized by its global perspective (observations are taken at the system level), its view-based construction (models are constructed according to device or aspect-specific projections of the global behavior), and, perhaps most importantly, by its high potential for automation. Its focus on easy, user-level abstractions of the global system behavior, non-invasiveness of the technology, and sensitivity to the system changes makes it particularly valuable for capturing the evolution of a system during its life cycle.

In order to judge the practicality and in particular the scalability of this approach, we have looked at particular views of parametric systems. These so-called small instance views are obtained from the global system by observing only very few of the parametric components, say n. It turned out that the small instance views are sufficient to fully capture the behavior of the global system as long as no more than n components have become active.

Simple examples unfortunately revealed that this result cannot be generalized to capturing the full system as long as the number of concurrently active components is smaller than n. This observation, although sad from the theoretical point of view, is characteristic for our regular extrapolation approach: in practice, we will typically never arrive at a sound or complete model, but the model will successively reflect more and more properties, and increase its value for its various applications. In fact, in practice, liberal matching wrt. the learned models turned out to be much superior to 'precise' matching.

This more liberal approach to modelling very much resembles the approach proposed by [14], where observation-based 'guess-work' about invariants is proposed to the users in order to help them understand their programs better.

As a case study, we have looked at a complex CTI system, in order to show a pragmatic approach for dealing with parametric systems. Here it is important to find the right compromise in the precision/cost trade-off. Accepted are typically all low cost means that provide serious warnings or error messages: a system that detects only 50% of the errors, but with a 90% hit rate (only 10% false alarms) is in practice much preferred over a system that covers 90% of the errors with a hit rate of 10 %. The latter systems are typically disabled after the treatment of a few false alarms. We are convinced that our pragmatic integration of formal methods into industrial practice will give the formal methods approaches a strong push, because it enables practitioners to feel the power of the techniques without suffering from too much pain. We observed this already in the application to the treatment of legacy systems: there the pain is often so strong that users are grateful for new alternatives.

The first practical results are very encouraging, but there is still a long way to go until our approach can be used in the large. This concerns enhancing the automation, optimizing the learning procedure, investigating the frame conditions for certain application-specific optimizations [9], improving the way to integrate expert knowledge into the model construction process, and concisely representing large models. This requires a lot of theoretical and conceptual work, which, however, should be based on empirical results in order to guarantee its practical impact.

Acknowledgement

We are very grateful to the ITE team, in particular to Oliver Niese, for discussions and fruitful comments.

References

1. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 2(75):87–106, 1987.
2. The Esterel Synchronous Programming Language: Design, Semantics, Implementation (1992) (Make Corrections) (352 citations) Gerard Berry, Georges Gonthier Science of Computer Programming

3. T.S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978.
4. E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of theoretical computer science*. Elsevier, 1990.
5. A.Groce, D.Peled, M.Yannakakis: *Adaptive model checking*, 8th Conf. on Tools and Alg. Construction and Analysis of Systems (TACAS 2002), LNCS N.2280, pp.357–370, Springer Verlag.
6. A. Hagerer, H. Hungar, O. Niese, and B. Steffen: *Model Generation by Moderated Regular Extrapolation*. Proc. of the 5th Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2002), Grenoble (F), LNCS 2306, pp. 80-95.
7. A. Hagerer, T. Margaria, O. Niese, B. Steffen, G. Brune, and H. Ide. Efficient regression testing of CTI-systems: Testing a complex call-center solution. In *Annual Review of Communication*, volume 55. Int. Engineering Consortium (IEC), 2001.
8. D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 3(8):231–274, 1987.
9. H. Hungar, O. Niese, B. Steffen: *Domain-Specific Optimization in Automata Learning* Proc. 15th Int. Conf. on Computer Aided Verification (CAV 2003), Boulder (USA), July 2003, LNCS 2725, Springer Verlag, 2003, pp . 315-327.
10. B. Jonsson, T. Margaria, G. Naeser, J. Nyström, and B. Steffen. Incremental requirement specification for evolving systems. Nordic Journal of Computing, vol. 8(1):65, Also in *Proc. of Feature Interactions in Telecommunications and Software Systems 2000*, 2001.
11. T. Margaria, O. Niese, B. Steffen, A. Erochok: *System Level Testing of Virtual Switch (Re-)Configuration over IP*, Proc. IEEE European Test Workshop, Corfu (GR), May 2002, IEEE Society Press.
12. T. Margaria, O. Niese, B. Steffen: *A Practical Approach for the Regression Testing of IP-based Applications*, invited contribution to the volume "IP Applications and Services 2003: A Comprehensive Report" IEC, Int. Engineering Consortium Chicago (USA), ISBN 1-931695-12-1.
13. T. Margaria, B. Steffen: *Lightweight Coarse-grained Coordination: A Scalable System-Level Approach*, invited contrib. to the Special Section on 'Formal Methods in Industrial Critical Systems' (Ed. JAco van de Pol) of STTT, Int. Journal on Software Tools for Technology Transfer, Springer Verlag - to appear.
14. Toh Ne Win, Michael D. Ernst, Stephen J. Garland, Dilsun Kirli, and Nancy Lynch: *Using simulated execution in verifying distributed algorithms*, in VMCAI'03, 4th Int. Conf. on Verification, Model Checking and Abstract Interpretation, (New York, NY), January 2003, LNCS 2575, Springer Verlag pp. 283-297.
15. O. Niese, T. Margaria, A. Hagerer, M. Nagelmann, B. Steffen, G. Brune, and H. Ide. An automated testing environment for CTI systems using concepts for specification and verification of workflows. *Annual Review of Communication*, Int. Engineering Consortium Chicago (USA), Vol. 54, pp. 927-936, IEC, 2001.
16. O. Niese, B. Steffen, T. Margaria, A. Hagerer, G. Brune, and H. Ide. Library-based design and consistency checks of system-level industrial test cases. In H. Hußmann, editor, *Proc. FASE 2001*, LNCS 2029, pages 233–248. Springer Verlag, 2001.
17. B. Steffen. Unifying models. In R. Reischuk and M. Morvan, editors, *Proc. STACS'97*, LNCS 1200, pages 1–20. Springer Verlag, 1997.
18. B. Steffen H. Hungar, Behavior-based model construction. In S. Mukhopadhyay and L. Zuck, editors, *Proc. 4th Int. Conf. on Verification, Model Checking and Abstract Interpretation*, LNCS N.2575, Springer Verlag, 2003.
19. M.P. Vasilevskii. Failure diagnosis of automata. *Kibernetika*, 4:98–108, 1973.

Automatic Failures-Free Connector Synthesis: An Example

Paola Inverardi and Massimo Tivoli

University of L'Aquila
(Dip. Informatica)
via Vetoio 1, 67100 L'Aquila, Italy
{inverard,tivoli}@di.univaq.it

Abstract. Many software projects are based on the integration of independently designed software components that are acquired on the market rather than developed within the project itself. This type of components is well known as COTS (Commercial-Off-The-Shelf) components. Nowadays component based technologies provide interoperability and composition mechanisms that cannot solve the COTS components assembly problem in an automatic way. One of the main problems in components assembly is related to the ability to establish properties on the assembly code by only assuming a limited knowledge of the single components properties. Our answer to this problem is a software architecture based approach in which the software architecture imposed on the assembly, allows for detection and recovery of COTS integration anomalies. We build applications by assuming a defined architectural style. Then, we compose a system in such a way that it is possible to check whether and why the system presents some software anomalies (e.g.: deadlock, livelock). Depending on the kind of failures a recovery policy which can avoid the anomalies and obtain a correct assembly can be performed. A tool can then synthesize the assembly code (as a failures-free connector component) to glue together a set of COTS components. This glue code must be synthesized in such a way that (a well defined set of) functional properties required for the composed system are automatically guaranteed. In the paper we briefly describe our approach and then we present its application to an example.

1 Introduction

Many software projects are based on the integration of independently designed software components that are acquired on the market rather than developed within the project itself. This type of components is well known as COTS (Commercial-Off-The-Shelf) components. One of the main problems in assembling COTS components is related to the ability to establish properties on the assembly code by only assuming a limited knowledge of the single components properties. Our answer to this problem is a software architecture based approach in which the software architecture imposed on the assembly, allows for detection and recovery of COTS integration anomalies. Notably, in the context of

M. Wirsing et al. (Eds.): RISSEF 2002, LNCS 2941, pp. 184–198, 2004.

component based concurrent systems, COTS components integration may cause deadlocks or other software anomalies within the system [18,1,12,13]. The use of COTS components in system construction presents new challenges to system architects and designers [14]. Building a system from a set of COTS components introduces a set of problems. Many of these problems arise because of the nature of COTS components. They are truly black-box and developers have no method of looking inside the box. This limit is coupled with an insufficient behavioral specification of the component which does not allow to understand the component interaction behavior. Component assembling can result in architectural mismatches when trying to integrate components with incompatible interaction behavior [3]. Thus if we want to assure that a component based system validates specified behavioral properties, we must take into account the component interaction behavior. In this context, the notion of software architecture assumes a key role since it represents the reference skeleton used to compose components and let them interact. In the software architecture domain, the interaction among the components is represented by the notion of software connector.

Our aim is to analyze and fix dynamic behavioral problems that can arise from components composition. We propose an architectural connector-based approach [8,10,9]. The idea is to build applications by assuming a defined architectural style, namely a modified version of the C2 architectural style [15]. We compose a system in such a way that it is possible to check whether and why the system presents some software anomalies (e.g.: deadlock). We can then derive, in an automatic way, directly from the COTS (black-box) components, the code that implements a new component to insert in the composed system. This new component implements an explicit software connector. This code is derived in such a way that the functional properties of the composed system are satisfied. We assume that a behavioral specification of the composed system is available. This specification is provided through *Message Sequence Chart* (MSC) and *High Level MSC* (HMSC) specifications [21,20,22]. Moreover we also assume that a precise definition of the properties to satisfy exists through LTL (*Linear-time Temporal Logic*) formulas definition [2,6,4,5].

The paper is organized as follows. In Section 2 we introduce the problem we want to address and some background. Section 3 presents the technique to allow failures-free connectors synthesis [11] which is then applied in Section 4 on an example. Section 5 concludes.

2 Problem Description

We consider COTS components which are truly black-box components. The properties we want to check are functional properties as deadlock-freeness or general safety and liveness properties which describe expected behaviors of the system. The assembly A depends on the constraints induced by the architectural model the system is based on. The present architectural model, which defines the rules used to build the composed, is a modified version of the C2 architectural style. This modified version of C2 architectural style is called CBA (i.e. *Connector Based Architecture*) style and it is described in detail in [11].

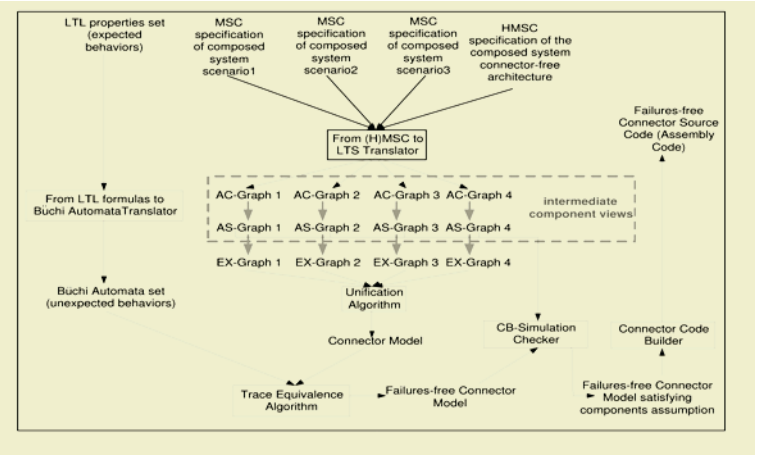


Fig. 1. Tool Architecture for the Automatic Failures-free Connector Synthesis.

It is important to notice that, besides assuming that the system architecture must reflect the rules of a well defined architectural style, we also assume that a system behavioral specification is provided through message sequence chart (MSC) and High level MSC (HMSC) specifications. Then we can derive, in an automatic way, from the MSC and HMSC specification, the behavioral description of each component in the composed system. This behavioral description can be derived by suitable applying the algorithm described in [21,20,22]. Each component behavioral description take a form of graph. Informally our approach is the following. The method starts off a set of components, and builds a connector following the reference style constraints. Components are enriched with additional information on their dynamic behavior which takes the form of graphs. Then property analysis is performed. If the synthesized connector contains property violating behaviors, a recovery policy is applied. Depending on the kind of property, the analysis of only the connector is enough to obtain a property-satisfying version of the system. Otherwise, the property is due to some component internal behavior and cannot be fixed without directly operating on the component code. In a component based setting in which we are assuming black-boxes components this is the best we can expect to do.

3 Connector Synthesis

Our goal is to develop a tool that performs an automatic software connector synthesis. The aim of this synthesis process is to derive the glue code for the components that constitute the composed system. The glue code is automatically synthesized directly from the partial specification of the composed system (MSCs and HMSC). The composed system automatically synthesized must satisfy the behavioral properties expected from the system designers and architects. In Figure 1 we illustrate the automatic synthesis tool architecture.

We represent with labelled boxes the components of the synthesis tool and with the labels the input and output data for each functional component of the tool. We have represented with the labels followed by a gray bold arrow intermediate data that are necessary to perform some transformations on the output data of the *"From (H)MSC to LTS Translator"* component. We perform these transformations in order to obtain the input data for the *"Unification Algorithm"* component. The gray bold arrows represent these transformations.

We model components as labelled transition systems (LTS) where labels represent messages that the components can input and output on the communication channel. We consider the system as a parallel composition of all components. In literature many approaches to build, in an automatic way, an LTS from an MSC specification exist; we are entirely based on the approach described in [21,20,22]. We adapt these algorithms to build the actual behavior graph (AC-Graph) [8,10,9] of the system components directly from the MSC specifications. A formal definition of AC-Graph is in [8]. Informally we can say that an AC-Graph for a component C describes the interaction behavior of C with its external environment. This environment is modelled as the parallel composition of all the others components in the system. We wish to derive from a component behavior the requirements on its environment that guarantee deadlock-freedom. From the AC-Graphs we can automatically derive the corresponding AS (*AS-sumption*) graphs. The AS-Graph is different from the corresponding AC-Graph only in the arcs labels. Actually these labels are symmetric since they model the deadlock-free environment as each component expects it. This is true because we assume synchronous communication between components of the system. Given the CBA style [8], the component environment can only be represented by one or more connectors, thus we refine the definition of AS-Graph into a new graph, the EX-Graph, that represents the behavior that the component expects from the connector. Each component EX-Graph represents a partial view of the connector expected behavior. It is partial since it only reflects the expectations of a single component. Actually in the EX-Graph of a component C we have actions on a communication channel that is unknown for C and actions on a communication channel that links C with the connector. The global connector behavior will be derived by taking into account all the EX-graphs. This will be done through a sort of unification algorithm [8]. The role of the connector is to route every component request to the request receiver component. Then it returns the request response to the component which fired the request. We automatically synthesize a model of the behavior of the connector which contains all the possible request routing policies. Then we perform analysis of properties and recovery. This means that we could allow a designer to assign a precise scheduling policy to the connector. Referring to the usual model checking approach [2,7], we can think of defining the properties that the system must satisfy by using *Linear-time Temporal Logic* (LTL) formulas [6,4,5]. We can specify the set P of properties that describe the expected behaviors of the system. The synthesis tool uses the set of properties P to identify connector graph portions that do not satisfy the requested routing policy. Therefore every property in P is used to check if the connector contains an unexpected behavior. For each property

$p_i \in P$ the synthesis tool derives the Büchi Automaton [2,6,4] BA_i corresponding to the LTL formula $!p_i$, where the symbol ! is the logical negation operator. Then the tool verifies if in the connector graph a (possibly infinite) arc labels sequence t (an execution trace) exists in such a way that an *accepting* execution of BA_i on the word corresponding to t exists. We have explained formally this test in [11]. Informally, an execution trace t on the connector graph represents a connector behavior that satisfies the negation of the property p_i, thus it represents an unexpected behavior of the connector. At this point the tool could apply two possible recovery strategies to guarantee the desired behavior: i) It does not modify the connector semantics or ii) it modifies the connector semantics. In the first case the tool, in the code derivation step, considers the connector graph portions marked as unexpected behavior, as exceptional running traces. Thus it derives for them a code that implements an exception handling block. In the second case the tool simply cuts the connector graph portions marked as unexpected behavior. Thus the code derivation step does not implement these unexpected running traces. Finally the synthesis tool verifies if the connector ensures the expected behavior for all components connected to it. In this last step the tool compares any AS-Graph with a corresponding connector graph portion by using a sort of state-based equivalence (CB-Simulation) [8]. After we have obtained the connector graph that satisfies a particular routing policy, the tool automatically derives the code of a new component, the connector component, to insert in the composed system. This new component routes the requests of the components connected to it in such a way that, by construction, the composed system behaves as required. It is important to notice, in Figure 1, that every LTS corresponding to a component is expressed by using a data structure that takes the form of automata (the AC-Graph). We can give a textual representation for each AC-Graph by using a process algebra specification (e.g. *Calculus of Communicating Systems* (CCS) [16]). The following are the steps of the algorithm used to build the failures-free connector graph and to derive the failures-free assembly code:

1. let K be the connector to build;
2. FOR EACH component C_i build the EX-Graph EX_i for C_i;
3. IF it is impossible to unify all the EX-Graphs EX_i THEN $exit(FAILURE)$ ELSE unify all the EX-Graphs EX_i and put in K the EX-Graphs unification result;
4. FOR EACH property p_i in the set P of properties to be validated, build the Büchi Automaton $BA_{!p_i}$ for the logical negation of p_i;
5. IF a *Connector Graph Execution Trace* t_j exists (in K) in such a way that an accepting execution of $BA_{!p_i}$ on t_j exists, THEN mark the trace t_j for all j;
6. remove from K all the paths that contain a marked *Connector Graph Execution Trace*;
7. FOR EACH component C_i IF $CBSimulation(AS_i, CB_i)$ does not successfully terminate THEN $exit(FAILURE)$ ELSE derive from K the assembly code implementation;
8. $exit(SUCCESS)$;

where:

AS_i is the AS-Graph of the component C_i which is connected to the connector; CB_i is the CB-Graph [11] for C_i. This graph represents the portion of the connector graph that communicates with the component C_i.

$CBSimulation(AS_i, CB_i)$ successfully terminates if the expected behavior of the environment for the component C_i (AS_i) is *CB*-simulated [11] from the portion of the connector behavior regarding the communication with a given component (C_i). $CBSimulation(AS_i, CB_i)$ is performed by not considering the expected environment behaviors (paths of AS_i) corresponding to execution paths in $BA_{!p_i}$. Informally *CB-Simulation* is a notion of simulation based on *stuttering* equivalence [17].

4 Example: The Dining Philosophers

This example is an instance of the well known *Dining Philosophers* problem [19] in which we consider two philosophers and two forks. We present the component structure of the dining philosophers problem in Figure 2.

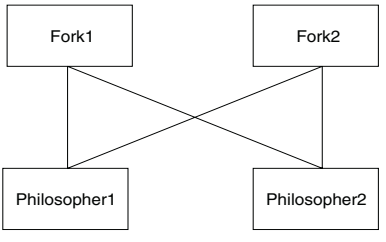


Fig. 2. Architectural View of the Dining Philosophers Problem.

There are 4 components: i) the first fork (**Fork1**), ii) the second fork (**Fork2**), iii) the first philosopher (**Philosopher1**), and iv) the second philosopher (**Philosopher2**). The forks components can iteratively wait for a request, give the fork, and then wait for the fork to be released. The philosophers can nondeterministically choose to ask for a fork, get it, then ask for the other, eat and then release the forks. Since a philosopher to eat needs both the forks it is obvious that in the following scenario a deadlock could arise: 1) component Philosopher1 requests and gets the resource of component Fork1; 2) component Philosopher2 requests and gets the resource of component Fork2; 3) component Philosopher1 requests and waits for the resource of component Fork2; 4) component Philosopher2 requests and waits for the resource of component Fork1. In this scenario Philosopher1 is waiting for Fork2 release. Since Philosopher2 gets the resource of Fork2, this event can be caused only by Philosopher2 who is waiting for Fork1 release. Since Philosopher1 gets the resource of Fork1, this event can be caused only by Philosopher1. Thus each system component is waiting for an event that only another system component can cause. It means a deadlock.

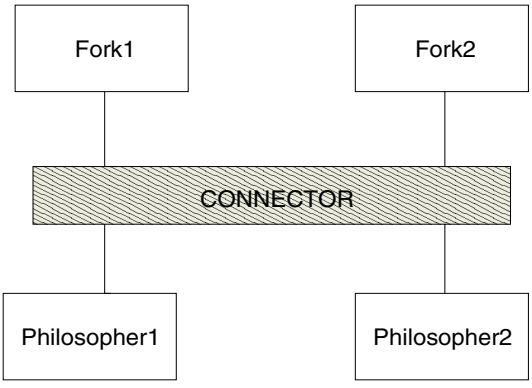


Fig. 3. Architectural View of the Connector-based Dining Philosophers Problem.

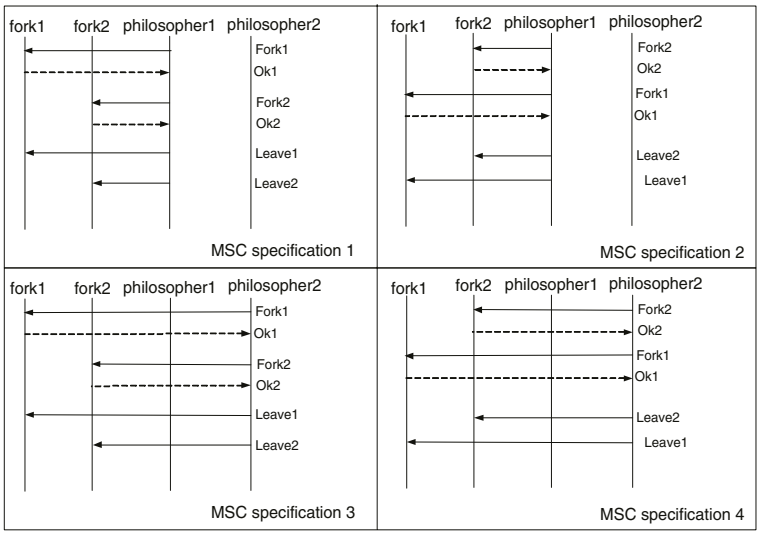


Fig. 4. Basic MSC specification of the composed system.

Now we present the connector based component structure of the dining philosophers problem in Figure 3. The role of the connector is to route every component request to the request receiver component. Then it returns the request response to the component which fired the request. Through the routing policy it implements, the connector can decide to accept or to reject a specific request. Suppose that we can benefit of the behavioral specification of the composed system given in Figures 4 and 5 as MSC and HMSC specification respectively.

By applying the MSC to LTS translation algorithm described in [11], and based on an our purpose adapted version of the algorithm described in [21,20,22], we can obtain the AC-Graphs for each component in the composed system

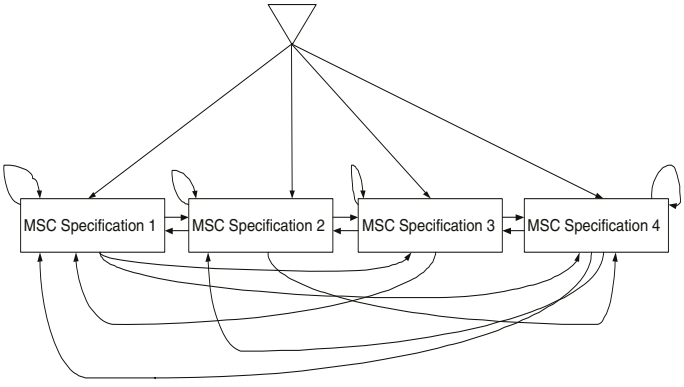


Fig. 5. High Level MSC specification of the composed system.

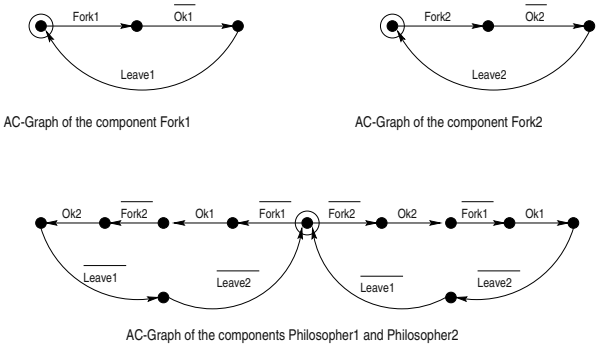


Fig. 6. AC-Graphs of the Dining Philosophers components.

($Fork1$, $Fork2$, $Philosopher1$ and $Philosopher2$). We show these AC-Graphs in Figure 6.

From these graphs we derive the AS-Graphs showed in Figure 7 and from the AS-Graphs we derive the EX-graphs in Figure 8. From the EX-Graphs by applying the unification algorithm described in Section 3 we can obtain the connector graph illustrated in Figure 9.

As showed in Figure 9, we automatically synthesize a model of the behavior of the connector which contains all the possible request routing policies. Then we perform analysis of deadlocks and recovery. The deadlocks analysis step consists of searching for stop nodes in the connector behavioral graph. These nodes represent states in which the system does not perform any action. Thus stop nodes represent deadlock states. The deadlocks recovery step consists of cutting the connector graph branches that lead to stop nodes. In Figure 11 we have showed the deadlock-free connector graph.

It is worthwhile noticing that before the possible deadlocks are fixed the connector contains all possible composed system behaviors. This means that it

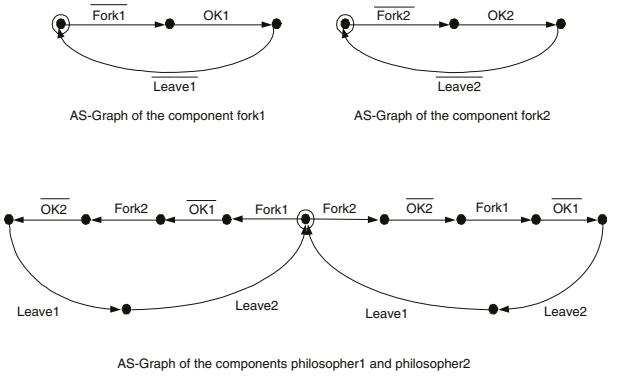


Fig. 7. AS-Graphs of the Dining Philosophers Components.

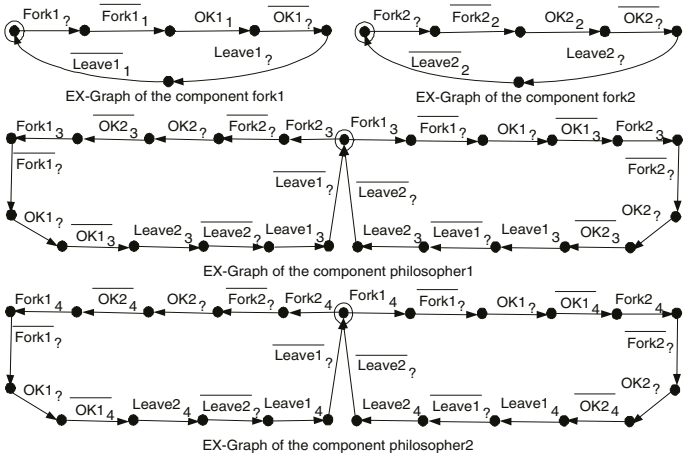


Fig. 8. EX-Graphs of the Dining Philosophers Components.

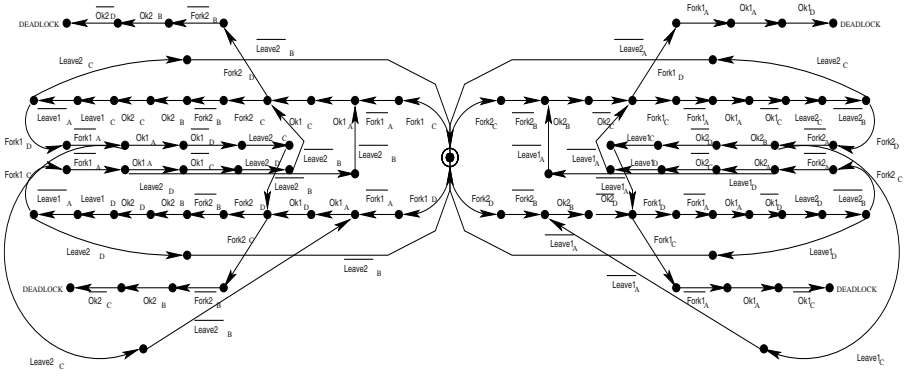


Fig. 9. Automatically Synthesized Connector.

contains all possible routing policies. A designer can now think not only of a deadlock-free routing policy but of a precise scheduling one. For instance he might want the philosophers to eat in turn or that the Philosopher1 always eats twice before Philosopher2. This means that we could allow a designer to assign a precise scheduling policy to the connector. Suppose that the composed system designer has specified the following properties:

- **PROPERTY 1**:
  $LP_1 \equiv \ [](([\overline{Ok1_C} \wedge \overline{Ok2_C}) \longrightarrow \ X([](!([\overline{Ok1_C} \wedge \overline{Ok2_C}))));$

- **PROPERTY 2**:
  $LP_2 \equiv \ [](([\overline{Ok1_D} \wedge \overline{Ok2_D}) \longrightarrow \ X([](!([\overline{Ok1_D} \wedge \overline{Ok2_D})))).$

With these two properties, the system designer specifies two expected system behaviors that have been specified to avoid this two possible scenarios: i) the first philosopher eats and the second philosopher waits for the forks forever and ii) the second philosopher eats and the first philosopher waits for the forks forever. These two conditions are important for the progress of the system because they implies that both the two philosophers eat an equal number of times. More exactly for equal number of times we means the same number of times in quantity order. By satisfying LP_1 and LP_2 the connector can avoid the starvation. For example with LP_1 the user specifies that for all executions (in the connector model that we are considering), in which the first philosopher has requested and obtained both the two forks then it will have to be always true that the first philosopher does not obtain both the two forks again. Analogously for LP_2.

To limit the size of the paper, we describe the behavioral properties analysis step only for the property LP_1. The following approach is completely equivalent for LP_2. We derive, in an automatic way, a suitable form of the Büchi Automaton $BA_{!LP_1}$ corresponding to the property $!LP_1$ in order to search, in the graph of Figure 11, *Connector Graph Execution Traces* t_j in such a way that an *accepting* execution of $BA_{!LP_1}$ on t_j exists. In Figure 10 we have illustrated the Büchi Automaton, corresponding to $!LP_1$.

The reader, by looking the Figure 11, can easily see that there are infinite *Connector Graph Execution Traces* t_j, corresponding to the two paths ***LP1_Failure1*** and ***LP1_Failure2*** in Figure 11 respectively, in such a way that an *accepting* execution of $BA_{!LP_1}$ on t_j exists. This means that the deadlock-free connector model satisfies the LTL formula $!LP_1$ since there is at least one execution trace in which only the first philosopher requests and obtains the two forks and the second one waits for the forks forever. Thus the derived deadlock-free connector model is a really deadlock-free model of the connector but it does not guarantee the progress of the system.

4.1 Behavioral Failures Recovery

After we have performed the behavioral failures analysis step, we can have possible traces or paths in the connector graph in which the properties are not

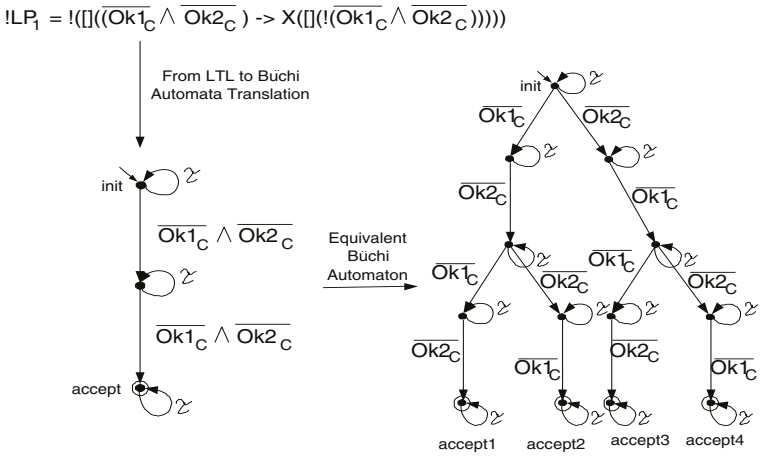


Fig. 10. Equivalent Büchi Automata corresponding to LTL property $!LP_1$.

satisfied (behavioral failures). We propose a strategy based on the elimination of all the paths, in the connector graph, in which we have found a *Connector Graph Execution Trace* accepted by the Büchi Automaton $BA_{!LP_1}$. Then, for every component C_i, we checked if its AS-Graph AS_i is simulated by the CB-Graph CB_i of C_i under the notion of CB-Simulation. We have seen, by performing the analysis step on properties LP_1 and LP_2, that there are two paths, in the deadlock-free connector graph, in which the property LP_1 is not satisfied and there are others two paths in which LP_2 is not satisfied. In Figure 11 we have represented this four paths by coloring gray the nodes in the paths.

We have called the two paths that not satisfy the property LP_1 as ***LP1_Failure1*** and ***LP1_Failure2*** respectively and the two paths that not satisfy the property LP_2 as ***LP2_Failure1*** and ***LP2_Failure2*** respectively. By cutting those paths from the connector graph we obtain the *Deadlock-Free and Progress-Satisfying Connector Graph* of Figure 12.

By the Figure 12, we can see that this model of the connector forces the two philosophers to eat in an alternate way. This is true because every time the first philosopher eats then necessarily the second philosophers will eat too and viceversa. This implies that this model of the connector satisfies the properties LP_1 and LP_2. The reader can easy verifies that for every component C_i the CB-Graph CB_i (of the connector graph of Figure 12) for C_i simulates the AS-Graph AS_i under the notion of CB-Simulation[1].

5 Conclusions and Future Works

In this paper we have presented an example of application of our approach to compose a system out of a set of black-box components in a behavioral failures-free way. A key feature of the approach is that the system architecture follows

[1] Except for the paths of AS_i that are also execution paths of $BA_{!LP_1}$.

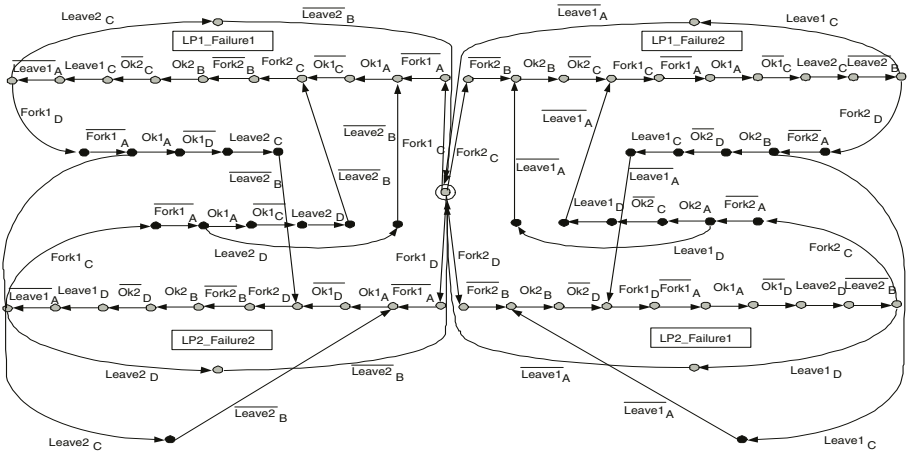


Fig. 11. Deadlock-Free Connector Graph not-satisfying the properties LP_1 and LP_2.

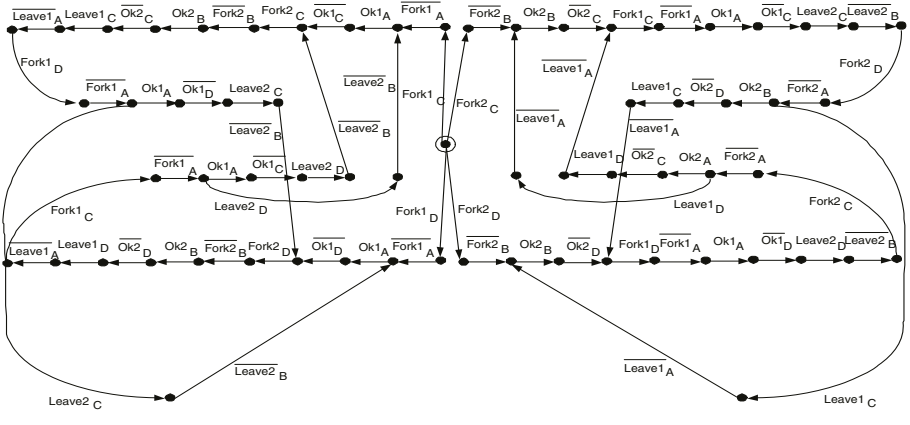


Fig. 12. Deadlock-Free and Progress-Satisfying Connector Graph.

a precise architectural style; this makes the automatic synthesis of connectors possible. Furthermore, the fact that it is known how the components will interact through the connectors makes behavioral failures analysis and/or recovery possible. The approach thus exploits the knowledge of the system architecture in order to improve the quality of the resulting system with respect to architectural mismatches.

As far as complexity is concerned, our approach, contrary to [12], does not improve standard analysis techniques. From this point of view our method shares the same problems of the techniques based on analysis performed on the global system behavioral model. The added-value of our method is the ability to generate a failures-free system, by automatically synthesizing a *safe* connector.

At present we have applied the approach in a real scale context, namely in the context of COM/DCOM applications [9]. At a very high level of description, what has been done is to recast the notions and techniques introduced in the paper in the COM/DCOM context by suitably extending the IDL in order to accommodate our notion of AC-graph in the component interface description. Then the application is build according to the CBA style by letting a COM/DCOM server to act as the synthesized connector. Behavioral failures freedom is then checked by following our definitions. To our respect, it showed the feasibility of our approach and its applicability in commercial component based contexts. As far as components are concerned we only assumed to have a description of the composed system behavior by means of MSCs, which is, in our view, an acceptable hypothesis.

In [10] we mentioned that our method can be applied to multi-layered systems as well. When more than two layers are considered we have to split AS-Graphs according to the two component domains top and bottom. From these we can generate the corresponding pairs of expected graphs which should then be unified. The unification algorithm must be slightly modified to cope with this extension. Our method can have better state complexity results in a layered system, since the connector corresponding to each layer must only consider the subset of components that interact with it.

References

1. B. Boehm and C. Abts. Cots integration: Plug and pray? *IEEE Computer*, 32(1), Jan. 1999.
2. O. G. Edmund M. Clarke, Jr. and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, London, England, 2001.
3. D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6), Nov. 1995.
4. P. Gastin and D. Oddoux. Fast ltl to buchi automata translation. *in Proceedings of CAV'01*, 2001.
5. R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of liner temporal logic. *in Proc. of the 15th IFIP/WG6.1 Symposium on Protocol Specification, Testing and Verification (PSTV'95)*, 1995.
6. D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. *RIACS Technical Report 01.21*, 2001.
7. D. Giannakopoulou, J. Kramer, and S. Cheung. Behaviour analysis of distributed systems using the tracta approach. *Journal of Automated Software Engineering, special issue on Automated Analysis of Software*, 6(1):7–35, January 1999.
8. P. Inverardi and S. Scriboni. Connectors syntesis for deadlock-free component based architectures. *16th ASE, Coronado Island, California*, November 2001.
9. P. Inverardi and M. Tivoli. Automatic synthesis of deadlock free connectors for com/dcom applications. In *ACM Proceedings of the joint 8th ESEC and 9th FSE, ACM Press*, Vienna, Sep 2001.
10. P. Inverardi and M. Tivoli. Deadlock-free software architectures for com/dcom applications. *to appear on Elsevier Journal of Systems and Software Special Issue on Component-based Software Engineering*, Nov. 2001.

11. P. Inverardi and M. Tivoli. Connectors synthesis for failures-free component based architectures. *Technical Report, University of L'Aquila, Department of Computer Science, http://www.di.univaq.it/tivoli/ffsynthesis.ps*, ITALY, August 2002.
12. P. Inverardi and S. Uchitel. Proving deadlock freedom in component-based programming. *Proceed. FASE 2001, LNCS 2029 pp. 60-75*, April 2001.
13. N. Kaveh and W. Emmerich. Deadlock detection in distributed object system. *8th FSE/ESEC, Vienna*, September 2001.
14. D. Mark, R. Vigder, and J. Dean. An architectural approach to building systems from cots software components. *National Research Council Report Number 40221.*
15. N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of off-the-shelf components in c2-style architectures. In *In Proceedings of the 1997 Symposium on Software Reusability and Proceedings of the 1997 International Conference on Software Engineering*, May 1997.
16. R. Milner. *Communication and Concurrency.* Prentice Hall, New York, 1989.
17. R. D. Nicola and F. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM*, 42(2):458–487, 1995.
18. C. Szyperski. *Component Software. Beyond Object Oriented Programming.* Addison Wesley, Harlow, England, 1998.
19. A. S. Tanenbaum. *Modern Operating Systems.* Prentice Hall Inc., 1992.
20. S. Uchitel and J. Kramer. A workbench for synthesising behaviour models from scenarios. In *in proceeding of the 23rd IEEE International Conference on Software Engineering (ICSE'01)*, Toronto, Canada. May 2001.
21. S. Uchitel, J. Kramer, and J. Magee. Detecting implied scenarios in message sequence chart specifications. In *ACM Proceedings of the joint 8th ESEC and 9th FSE, ACM Press*, Vienna, Sep 2001.
22. S. Uchitel, J. Kramer, and J. Magee. From sequence diagrams to behaviour models. In *In WTUML: Workshop on Transformations in UML. Satellite event of the European Joint Conferences on Theory and and Practice of Software (ETAPS'01)*, Genova, Italy. April 2001.

Module Dependences in Software Design

Daniel Jackson

MIT Lab for Computer Science
200 Technology Square
Cambridge, Massachusetts 02139
dnj@mit.edu

Abstract. A new model of inter-module dependences is proposed. The key idea is that dependences are mediated by *specifications*, so that not only the existence of a dependence is recorded, but also its quality. A single module does not necessarily offer only a single specification; each dependent module may use it through a different specification. This notion of dependence seems to explain some common programming idioms more readily than the conventional notion, and offers new opportunities for analysis and design critique.

1 Introduction

Software designers have long recognized the significance of dependences between modules in evaluating a design. When modules have many interdependences, the system is harder to understand; there is less flexibility to divide the labour of coding; and local changes have wide repercussions. But despite the seminal role of dependences, most designers have only a fuzzy sense of what they are and how to express them.

Although various kinds of code dependences have been heavily investigated because of their role in separate compilation and program analysis, much less attention has been paid to design dependences. As a result, dependence diagrams are much less useful than they might be in program design, and cannot be used as a basis for analysis.

This paper outlines a new dependence model. Work on the model is in its preliminary stages, but it does seem to resolve some of the problems of the standard model, and in particular accounts better for some common idioms in object-oriented designs.

2 The Standard Model and Its Deficiencies

The standard notion of module dependence, first articulated by Parnas [8], is familiar to most developers. The system is described as a graph whose nodes represent modules and whose edges represent syntactic dependences. A dependence of module `A` on module `B`, drawn as an arrow from `A` to `B`, says that `A`, in providing services to the modules that depend on it, makes use of `B`. More precisely, `A` *depends on* `B` or *uses* `B` if "correct execution of `B` may be necessary for `A` to complete the task described in its specification" [8].

M. Wirsing et al. (Eds.): RISSEF 2002, LNCS 2941, pp. 198–203, 2004.

In teaching software engineering to undergraduates over the last few years, we have used these 'module dependence diagrams' to explain software designs, and have required students to produce them as documentation [6]. But we have come to realize that these diagrams are inadequate, and often do not capture the insights that motivate a design:

- *Transitivity.* The dependency relation is, by definition, transitive, when intuitively it should not be. If `A` uses `B` and `B` uses `C`, then, in the absence of additional information, we must assume that any change to `C` affects `B`, and thus indirectly `A`, and thus `A` uses `C`. But perhaps `B` was inserted between `A` and `C` precisely to shield `A` from changes to `C`, and most changes to `C` will not affect `A`.
- *Polymorphism.* Crucial elements of modern languages, such as polymorphism and interfaces, are not accommodated. Consider a typical Java program in which a client class `C` makes use of a hashtable class `H` and a key class `K`. Does `H` depend on `K`? On the one hand, the code of `H` makes no reference to the code of `K`; `H` likely belongs to a library that was written before `K`. On the other hand, changes to `K` can certainly cause `H` to fail. If the `equals` and `hashCode` methods of `K` are incompatible, for example, a crucial representation invariant of the hashtable will be violated (that equal keys are kept in the same bucket chains), and a lookup with a given key may not return the value last inserted under that key.
- *Preconditions.* Presenting a module with a bad procedure argument can cause it to fail. If a module's specification imposes a precondition, the designer's intent was that blame for a bad input should be attributed to the calling module. But a module cannot easily be said to 'use' the modules that call it.
- *Replacement.* Since a dependence of `A` on `B` does not indicate which properties of `B` are actually used by `A`, one cannot tell what properties of `B` should be preserved in a module that replaces it. The dependence diagram cannot therefore be used to reason about compatibility of components.

3 A New Model

Our new model is based on two simple ideas. First, a dependence represents an *assumption*: a module `A` depends on a module `B` if the designer or implementer of `A` makes an assumption about the environment in which `A` is to be inserted that is justified in the constructed system by the presence of `B`. The assumption may be that `A` is executed with a certain frequency; or that values passed to it satisfy certain conditions; or that values returned to it by procedures it calls are related to the values it passed them in certain ways.

Second, dependences are *mediated by specifications*: one module does not depend directly on another, but rather on a specification which the other module must satisfy. A dependence of module `A` on module `B` mediated by specification `S` means that `A` assumes the conditions guaranteed by `S`, which the system is configured to provide through `B`.

These notions address some but not all of the deficiencies of the standard model:

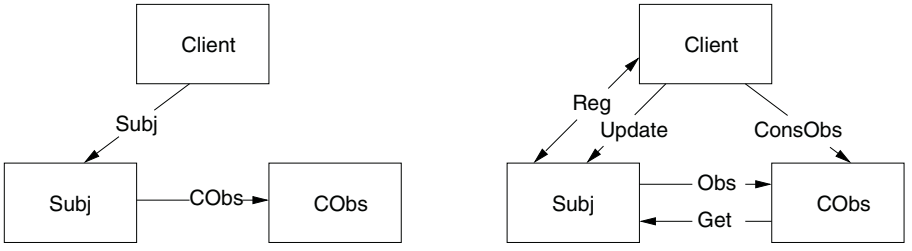
- There is no transitivity problem, because of the interposed specifications. If `A` depends on `B` via specification `S`, and `B` depends on `C` via specification `T`, it is clear that changes to `C` that preserve the behaviour required by `T` will not affect `B`, and of the changes that affect `B`, only those that cause `B` no longer to satisfy `S` will affect `A`.
- The use of polymorphic modules is clarified. A hashtable `H` depends on its key class `K` via the specification `Object` — what in Java is referred to as 'the object contract'. The dependence of `H` on `K` is an artifact of the configuration: in this case due to some client class `C` passing objects of type `K` to the hashtable class `H`. The designer of `H` is not expected to predict the use of `K`, but can assume that any class whose objects are used as keys will satisfy `Object`.
- A precondition is represented by a dependence in the reverse direction; the pre- and postconditions of a procecure specification can be regarded as distinct specifications associated with data flows in different directions.
- Reasoning about replacement comes easily as a byproduct of the use of specifications. A dependence of a module that makes the assumption described in specification `S` will be preserved appropriately if the module it depends on is replaced by one that also satisfies `S`. If specifications are treated as uninterpreted names (which will often be convenient), an explicit ordering on specifications can be used to justify replacements that do not involve exact matches.

Specifications appear, at least implicitly, in the traditional model, but they play a different role. In Parnas's formulation, the specification of the *using* module, and not the *used* module is relevant. Furthermore, in our model, specifications are associated with dependences rather than modules. A single module may have many specifications, each representing a view of the module presented to another module that depends on it. In the hashtable example, key class `K` appears to the hashtable class `H` under the specification `Object`, but will likely appear to the client class `C` under a stronger specification that provides domain-specific properties.

4 Example: Observer

Many of the 'Gang of Four' design patterns [3] are motivated by dependence considerations. We illustrate our dependence model on the *Observer* pattern.

The figure shows dependence diagrams for a fragment of a program that has been refactored with *Observer*. The original program on the left has a subject class `Subj` and a concrete observer class `CObs`. Updates to `Subj` from a client class `Client` are propagated to `CObs` by calls to procedures specific to `CObs`. We have shown this by labelling the dependence with a specification `CObs`: using the name of the module for the specification is intended to suggest that the specification promises all the properties the module is designed to provide.



The refactored program on the right differs primarily in the specifications that mediate the dependences. The key difference is that `Subj` no longer interacts with `CObs` via the specification `CObs`, but rather through a far weaker specification `Obs`, a generic observer specification that could be used for other observer classes. This is the central motivation of *Observer*: it allows new observer classes to be added without changing the code of `Subj`. There is a back dependence from `CObs` to `Subj` mediated by the specification `Get` representing the calls `CObs` must now make to `Subj` to obtain state updates.

The relationship between the client class `Client` and the subject `Subj` has been refined to show two different interactions. The dependence mediated by the specification `Update` stands for the service `Subj` provides in updating its internal state in response to requests from `Client`. The label `Reg` stands for the registration service that `Subj` provides in which an object is added to its internal record of observers. There are two dependences associated with the `Reg`, shown as a double-headed arrow, corresponding to the pre- and postconditions of the registration operation. The precondition, shown as an arrow from `Subj` to `Client`, represents the obligation of `Client` to avoid registering observers that might create observation cycles.

Finally, there is a new dependence of `Client` on `CObs`, since the client class is now required to handle the observer object when it registers it. Assuming that `Client` creates the observer object, we can label the dependence with a specification `ConsObs` to represent the use of a constructor, but probably no other operations.

It is typical of a design pattern that the code is in some respects made more complicated. There are more dependences after the pattern has been applied, but they are more systematically organized. Most importantly, some aspects of the behaviour are factored out into dependences mediated by generic specifications (`Reg` and `Obs`). These specifications are the hallmark of the pattern.

5 Discussion

The dependence structure of a program is not trivially extractable from its code. For one thing, it may not be easy to determine which module in a system discharges the obligations incurred by another module's assumptions. In the hashtable example, finding the dependence of the hashtable module `H` on the key module `K` would require at least a type-based analysis to determine what types of object are inserted into the table.

Moreover, dependences represent the designer's subjective view of how responsibilities are partitioned amongst modules. In the *Observer* example, we have taken the traditional abstract data type viewpoint that the subject class, and the observer indirectly, provide a service to the client. But an equally tenable viewpoint is that the observer class depends on the services of the other classes: its specification requires it to display certain state changes, and it therefore requires notification of when those changes occur. In this viewpoint, the notification dependence from `Subj` to `CObs` is reversed. Similarly, in the hashtable example, the concern about keys might have been expressed as a precondition dependence from the hashtable class `H` to the client class `C` that assigns blame to `C` for passing a bad key to `H` on a call to `add`.

Dependence diagrams should not be confused with class diagrams or object models. A class diagram is just a graphical representation of the syntactic structure of an object-oriented program, showing the inheritance hierarchy and the source and target classes of instance variables. An object model is a graphical representation of a heap invariant [4]: it is thus *semantic* where the module dependence diagram is *syntactic*.

Since a module can be viewed as not only requiring multiple services, but also providing multiple services, it may be useful to track the relationship between the two. An enriched dependence diagram might include a relation for each module between its incoming and outgoing specifications. One could then perform a kind of 'module slice', following dependences between modules and within modules. The set of modules on which a module depends indirectly might be smaller than the set that would be obtained simply by following dependence edges between modules, especially for object-oriented programs in which the 'roles' a class plays are often largely independent. In the *Observer* example, such an analysis would show that the `Reg` dependence on `Subj` is not propagated further; the `Update` dependence, in contrast, is propagated to the `Obs` and `Get` dependences.

Some important forms of coupling are not captured in our model or in the standard model, most notably those due to *sharing*. Suppose module `M` uses module `W` to write a file and module `R` to read it; the file may be used to store state (such as a browser's bookmarks) across executions of the program, for example. The format with which `W` writes the file must be the same format with which `R` reads it. A change to `W` may therefore break `R`. But neither provides a service to the other, so the standard notion of dependence finds no coupling between them.

Our new model seems to offer some new insight into software designs and their rationale, but it is far from complete. Plans to develop it further include:

- Sorting out the relationship between dependences and code, and developing analysis algorithms for extracting at least approximate dependences. In Java, it should be possible to use rudimentary type inference to find dependences (in the style of Womble [5]) and to synthesize specification labels from explicit use of interfaces or from the use of subsets of a class's methods,

- Investigating the use of dependences in reasoning about which parts of a system compromise critical modules. Precondition dependences seem to offer some promise in representing couplings that are missed by the traditional model.
- Finding a way to represent couplings due to sharing, perhaps using the sharing constraints of the ML programming language [7] or the parameter binding constraints of Units [2].
- Establishing a better understanding of the role of dependences in software engineering, perhaps along the lines of Eppinger's work [1] in conventional engineering on the design structure matrix and its applications.

Since this paper was written, an analysis of the code of a radiotherapy machine was conducted using the dependence model [9]. It exposed a number of serious issues.

Acknowledgments

This paper grew out of discussions amongst the teaching staff of MIT's undergraduate software engineering class, 6.170: my co-lecturers John Chapin, Srinivas Devadas, Michael Ernst, and Barbara Liskov, and many teaching assistants, especially Felix Klock, Robert Lee, Jeremy Nimmer, and Jon Whitney. The ideas were developed in part with Allison Waingold.

This research was supported by grant 0086154 from the ITR program of the National Science Foundation, and through the High Dependability Computing Program from NASA Ames (Cooperative Agreement NCC-2-1298).

References

1. Steven D. Eppinger. Innovation at the Speed of Innovation. *Harvard Business Review*, January 2001.
2. M. Flatt, and M. Felleisen. Units: Cool modules for HOT languages. *Proc. Sigplan 1998 Conference on Programming Language Design and Implementation*, pp. 236–248.
3. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.
4. Daniel Jackson. Object Models as Heap Invariants. A chapter in: *Programming Methodology*, eds. Carroll Morgan and Annabelle McIver. Springer Verlag, 2002.
5. Daniel Jackson and Allison Waingold. Lightweight Extraction of Object Models from Bytecode. *IEEE Transactions on Software Engineering*, February 2001.
6. Barbara Liskov with John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison Wesley, 2001.
7. Robin Milner, Mads Tofte, Robert Harper and David MacQueen. *The Definition of Standard ML* (Revised), MIT Press, 1997.
8. David L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Eng*, SE-5, 2 (1979).
9. Andrew Rae, Prasad Ramanan, Daniel Jackson and Jay Flanz. Critical Feature Analysis of a Radiotherapy Machine. *International Conference of Computer Safety, Reliability and Security* (SAFECOMP 2003), Edinburgh, September 2003.

Towards Fully Automatic Execution Monitoring

Clinton Jeffery, Mikhail Auguston, and Scott Underwood

Department of Computer Science, New Mexico State University
{jeffery,mikau,sunderwo}@cs.nmsu.edu

Abstract. UFO is a new application framework in which programs written in FORMAN, a declarative assertion language, are compiled into execution monitors that run on a virtual machine with extensive monitoring capabilities provided by the Alamo monitor architecture. FORMAN provides an event trace model in which precedence and inclusion relations define a DAG structure that abstracts execution behavior. Compiling FORMAN assertions into hybrid runtime/post-mortem monitors allows substantial speed and size improvements over post-mortem analyzers. The UFO compiler generates code that computes the minimal projection of the DAG necessary for a given set of assertions. UFO enables fully automatic execution monitoring of real programs. The approach is non-intrusive with respect to program source code and provides a high level of abstraction for monitoring and debugging activities. The ability to compile suites of debugging rules into efficient monitors, and apply them generically to different programs, enables long-overdue breakthroughs in program debugging.

1 Motivation

Debugging is one of the most challenging and least developed areas of software engineering. A special issue of Communications of the ACM characterized the current state of debugging tools as a "Debugging scandal" [1]. According to the classic "Brook's rule" [2] more than 50% of all time and effort in a software project is spent in testing and debugging activities. Typical activities include detection and removal of errors, profiling, and performance tuning.

Debugging activities include queries regarding many aspects of target program behavior: sequences of steps performed, histories of variable values, function call hierarchies, checking of pre- and post-conditions at specific points, and validating other assertions about program execution. Performance testing and debugging involves a variety of profiles and time measurements. Visualization is another common debugging activity that may help locate logic or performance problems.

There is an urgent need for tools that automate the primary, labor-intensive tasks of debugging, but progress has been slow. Debugging automation has its own system of ideas and domain-specific programming activities. Support for these concepts and activities is essential in order to move debugging automation forward.

We are building automatic debugging tools based on precise program execution behavior models that enable us to employ a systematic approach. Our program behavior models are based on events and event traces [3][4][5]. Debugging automation refers to a computation over an event trace. *Program execution monitors* are programs

M. Wirsing et al. (Eds.): RISSEF 2002, LNCS 2941, pp. 204–218, 2004.

that load and execute a target program, obtain events at run-time, and perform computations over the event trace. Computations are performed during execution, post-mortem, or in any mixture of both times.

Any detectable action performed during a target program's run time is an *event*. For instance, expression evaluations, statement executions, and procedure calls are all examples of events. An event has a beginning, an end, and some duration; it occupies a time interval during program execution. This leads to the introduction of two basic binary relations on events: partial ordering and inclusion. Those relations are determined by target language syntax and semantics, e.g. two statement execution events may be ordered, or an expression evaluation event may occur inside a statement execution event. The set of events produced at run time, together with ordering and inclusion relations, is called an *event trace* and represents a model of program behavior. An event trace forms an acyclic directed graph (DAG) with two types of edges corresponding to the basic relations.

Our previous work included the FORMAN assertion language [3] and the Alamo program execution monitoring architecture [6]. FORMAN takes a top-down approach, introducing a domain-specific syntax for expressing bug manifestations and other behavior of interest, while Alamo takes a more bottom-up, implementation-driven approach, providing runtime system support for the development of monitors in which efficiency and scalability to real programs are primary concerns. Alamo's efficient source-level access and control over monitored programs has been integrated into a production virtual machine; in the absence of such support, monitoring would require extensive low-level instrumentation and control mechanisms.

The language UFO (Unicon-FORMAN) integrates the experience accumulated in these previous projects to provide a complete solution for development of an extensive suite of automatic debugging tools. UFO is an implementation of FORMAN for debugging programs written in the Unicon and Icon programming languages [7][8]. Previous FORMAN implementations worked on subsets of Pascal, and C languages and used post-mortem event trace processing methods that limited their applicability. In contrast, UFO uses the Alamo monitoring architecture that pervades the Unicon virtual machine to support debugging real programs at run time.

2 Unicon and Alamo

The Unicon language and the Alamo monitoring architecture provide the underlying research framework for the implementation of UFO. Unicon is an imperative, goal-directed, object-oriented superset of the Icon programming language. Unicon's syntax is similar to Pascal or Java, while its semantics are higher level, featuring built-in backtracking and heterogeneous data structures and string scanning facilities. Icon has influenced many scripting languages such as Python. Unicon is Icon's direct descendant, derived from Icon's implementation. It runs regular Icon programs and extends Icon's reach with object-orientation and packages, as well as a much richer system interface with high level graphics, networking, and database facilities.

The reference implementation of Unicon is a virtual machine. Virtual machines (VM) are attractive to language implementers, enhancing portability and allowing simpler implementation of very high level language features such as backtracking.

VMs are also ideal for developing debugging tools. VMs provide an appropriate level of abstraction for developing behavior models to describe program executions in a processor independent manner, as illustrated by the JPAX tool [9]. VMs also provide easy access to program state and control flow, the information most needed for debugging activities. Automatic instrumentation on multiple semantic levels is greatly simplified via the use of a VM. This potential was exploited in the Unicon VM by a framework that implements the Alamo monitoring architecture. Event instrumentation and processing support are an integral part of the VM.

The Alamo Unicon framework is summarized in Figure 1. Execution monitors (EM) and the target program (TP) execute as (sets of) coroutines with separate stacks and heaps inside a common VM. The VM is instrumented with approximately 150 kinds of atomic events, each one reporting a <code,value> pair. EMs specify categories of events by supplying an event mask when they activate the TP by coroutine switch. The TP executes up to an event of interest.

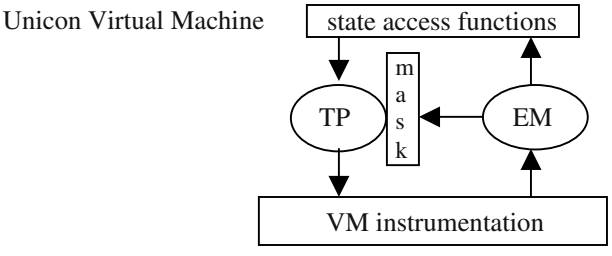


Fig. 1. Alamo architecture within the Unicon VM.

The event mask is used by the VM for instrumentation selection and control. Event reports during TP execution are coroutine context switches from the VM runtime system back to the execution monitor. In addition to the <code,value> reported for the event, the EM can directly access arbitrary variable values and state information from the TP via state access functions. Monitors are written independently from the target program, and can be applied to any target program without recompiling the monitor or target program. Monitors dynamically load target programs, and can easily query the state of arbitrary variables at each event report. Multiple monitors can monitor a program execution, under the direction of a monitor coordinator.

Alamo's goal was to reduce the difficulty of writing execution monitors to be just as easy as writing other types of application programs. UFO moves beyond Alamo to efficiently support FORMAN's more ambitious goal of reducing the difficulty of writing automatic debuggers to the task of specifying generic assertions about program behavior. UFO's FORMAN language is described in Section 4 below, but first it is necessary to present the underlying behavior model.

3 An Event Grammar for Unicon

Event grammars provide a model of program run time behavior. Monitors do not have to parse events using this grammar, since event detection is part of VM and UFO

runtime system functionality. Monitors implement computations over event traces supplied by the VM. An event is an abstraction of a detectable action performed at run time and has an event type and various attributes associated with it. The following description in fact provides a "lightweight" semantics of the Unicon programming language tailored for specification of debugging activities. An event corresponds to some specific action of interest performed during program execution. Event type is an important part of the behavior model.

Universal attributes are found in every event. They frequently are used to narrow assertions down to a particular domain (function, variable, value) of interest. Some of these attributes are much easier to obtain than others, and affect the optimizations that can be performed when generating monitor code; see Section 5 for details.

source_text:	in a canonical form
line_num, col_num:	source text locations
time_at_end, time_at_begin, duration:	timing attributes
eval_at_begin (Unicon-expr),	
eval_at_end (Unicon-expr):	runtime access to the program states
prev_path, following_path:	set of events before/after this event

Event types and their type-specific attributes are summarized in the table below.

Event Type	Description	Type Specific Attributes
prog_ex	Whole program execution	
expr_eval	expression evaluation	value, operator, type, failure_p
func_call	function call	func_name, paramlist
input, output	I/O	file
variable	variable reference	
literal	reference to a constant value	
lhp	lefthand part, assignment	address
rhp	righthand part, assignment	
clause	then-, else-, or case branch execution	
test	test evaluation	
iteration	loop iteration	

Event types form a hierarchy, shown in Figure 2. Subtypes inherit attributes from the parent type. Expression evaluation is the central action during Unicon program execution, this explains why the expr_eval event is on the top of the hierarchy.

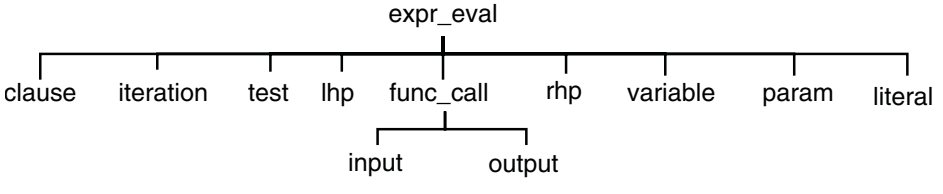


Fig 2. Event Type Inheritance Hierarchy

The UFO *event grammar* for Unicon is a set of axioms describing the structure of event traces with respect to the two basic relations: inclusion and precedence. The grammar is one possible abstraction of Unicon semantics; other event grammars with far more (or less) detail might be used. The event grammar limits what kinds of bugs can be detected, so some detail is useful. The grammar uses the following notation:

Notation	Meaning
A :: (B C)	B precedes A, A includes B and C
A*	Zero or more A's under precedence
A+	One or more A's under precedence
A | B	Either A or B; alternative
A?	A is optional
{ A , B }	Set; A and B have no precedence
x : A	Let x denote event A

```
prog_ex::    ( expr_eval *)
expr_eval::  (  ( expr_eval ) |            unary op
             ( expr_eval  expr_eval ) |    binary op
             ( expr_eval+ ) |
             ( test clause ) |             conditional / case expressions
             ( iteration * ) |             loops
             ( { lhp,  rhp} )              assignment
                                           lhp and rhp are not ordered,  beginning of
                                           lhp precedes rhp, and end of lhp follows rhp
           )
iteration::  ( test expr_eval*) | ( expr_eval* test ) | ( expr_eval * )
```

Execution of a Unicon program produces a set of events (an *event trace*) organized by precedence and inclusion into a DAG. The structure of the event trace (event types, precedence and inclusion of events) is constrained by the event grammar axioms above. The event trace models Unicon program behavior and provides a basis to define different kinds of debugging activities (assertion checking, debugging queries, profiles, debugging rules, behavior visualization) as appropriate computations over the event traces.

4 FORMAN

Alamo allows efficient monitors to be constructed in Unicon, but using a special-purpose language such as FORMAN, with the rich behavior model described in the preceding section, has compelling advantages. On a basic level, for example, it is convenient to refer to target program variables directly instead of through a library call. For example, in FORMAN we may refer to target program variable x, while in the Unicon monitor it is referenced as variable("x", &eventsource). UFO rules are up to an order of magnitude smaller (in terms of lines of source code) than the equivalent imperative monitors written in Unicon, depending on the type of quantifiers and aggregate operations used in the FORMAN rule.

More important than such conveniences are FORMAN's control structures that directly support dynamic analysis. FORMAN supports computations over event traces centered around event patterns and aggregate operations over events. The simplest event pattern consists of a single event type and matches successfully an event of this type or an event of a subtype of this type. Event patterns may include event attributes and other event patterns to specify the context of an event under consideration. For example, the event pattern

```
E: expr_eval  & E.operator == ":="
```

matches an event of assignment. Temporary variable E provides an access to the events under consideration within the pattern.

The following example demonstrates the use of an aggregate operation.

```
CARD[A: func_call & A.func_name == "read" FROM prog_ex]
```

yields a number of events satisfying the given event pattern, collected from the whole execution history. Expression […] is a list constructor and CARD is an abbreviation for a reduction of '+' operation over the more general list constructor:

```
+/[A:func_call & A.func_name=="read" FROM prog_ex APPLY 1]
```

Quantifiers are introduced as abbreviations for reductions of Boolean operations OR and AND. For instance,

```
FOREACH Pattern FROM event_set Boolean_expr
```

is an abbreviation for

```
AND/[Pattern FROM event_set APPLY Boolean_expr ]
```

Debugging rules in FORMAN usually have the form:

```
Quantified_expression
     WHEN SUCCEEDS SAY-clauses
      WHEN FAILS SAY-clauses
```

The Quantified_expression is optional and defaults to TRUE. The execution of FORMAN programs relies on the Unicon monitors embedded in a VM environment. Section 5 below describes the architecture of the UFO compiler and runtime system, which translates FORMAN to Unicon VM monitor code.

The following examples illustrate additional features of FORMAN as needed.

Application-Specific Analyses

This section presents formalizations of typical debugging rules. UFO supports and improves upon the most common application-specific debugging techniques. For example, UFO supports traditional precondition checking, or print statement insertion, without any modification of the target program source code. This is especially valuable when the precondition check or print statement is needed in not just one location, but instead in many locations scattered throughout the code.

Example #1: Tracing. Probably the most common debugging method is to insert output statements to generate trace files, log files, and so forth. This allows for subsequent human analysis, and while it has its limitations, it will remain a common technique. It is possible to request evaluation of arbitrary Unicon expressions at the be-

ginning or at the end of events. The VM evaluates these expressions at the indicated time moments, allowing dynamic instrumentation of the Unicon program, whether to print some values, or to call a visualization library subroutine.

```
FOREACH A: func_call & A.func_name == "my_func" FROM prog_ex
  A.value_at_begin(write("entering my_func, value of X is:",X))
AND
  A.value_at_end(write("leaving my_func, value of X is:", X))
```

This debugging rule causes calls to `write()` to be evaluated at selected points at run time, just before and after each occurrence of event A.

Example #2: Profiling. A myriad of tools are based on a premise of accumulating the number of times a behavior occurs, or the amount of time spent in a particular activity or section of code. The following debugging rule illustrates such computations over the event trace.

```
SAY("Total number of read() statements: "
    CARD[ r: input & r.filename == "xx.in" FROM prog_ex ]
    "Elapsed time for read operations is: "
    SUM [ r: input & r.filename == "xx.in" FROM prog_ex
          APPLY r.duration] )
```

Example #3: Pre- and Post-conditions. Typical use of assertions includes checking pre- and post-conditions of function calls.

```
FOREACH A:func_call & A.func_name=="sqrt" FROM prog_ex
  A.paramlist[1] >=0 AND
  abs(A.value*A.value-A.paramlist[1]) < epsilon
WHEN FAILS SAY("bad sqrt(" A.paramlist[1] ") yields " A.value)
```

Generic Bug Descriptions

Another interesting prospect is the development of a suite of generic automated debugging tools that can be used on any Unicon program. UFO provides a level of abstraction sufficient for specifying typical bugs and debugging rules.

Example #4: Detecting Use of Un-initialized Variables. Although reading an uninitialized variable is permissible in Unicon, this practice often leads to errors. Therefore, in this debugging rule all variables within the target program are checked to ensure that they are initialized before they are used.

```
FOREACH V: variable  FROM prog_ex
  FIND D: lhp FROM V.prev_path   D.source_text == V.source_text
WHEN FAILS SAY( " uninitialized variable " V.source_text)
```

Example #5: Empty Pops. Removing an element from an empty list is representative of many expressions that fail silently in Unicon. While this can be convenient, it can also be a source of difficult to detect logic errors. This assertion assures that items are not removed from empty lists.

```
FOREACH a:func_call & a.func_name=="pop" &
                      a.value_at_begin(*a.paramlist[1]==0)
  SAY("Popping from empty list at event " a)
```

5 Implementation Issues

The most important of implementation issues is the translation model by which FORMAN rules are compiled into Unicon monitors. Rules are written as if they have the complete post-mortem event trace available for processing. This generality is powerful; however the majority of assertions can be compiled into monitors that execute entirely at runtime. Runtime monitoring is the key to practical implementation. For assertions that require post-mortem analysis, the UFO runtime system computes a projection of the execution DAG needed to perform the analysis.

The UFO translation model categorizes each rule as either "runtime", "post-mortem", or "hybrid", denoting the amount of computations that can be performed at runtime. Runtime and hybrid categories are determined by constraints on FORMAN quantifier prefixes and result in more efficient code. Nested quantifiers and aggregate operations generally require post-mortem operation.

Translation Examples

Each FORMAN statement is translated into a combination of initialization, run-time, and post-mortem code. Monitors are executed as coroutines with the Unicon target program, as explained in Section 2. The following examples give a flavor of the run time architecture of monitors generated from the UFO high level rules.

Implementation of Example #1. A lone FOREACH quantifier is typical of many UFO debugging actions and allows computation to be performed entirely at runtime. The events being counted and values being accumulated determine an *event mask* in the initialization code that defines the Alamo events that will be monitored. The monitor's event processing loop implements a filter based on procedure name within an if-expression. The Unicon code blocks containing write() expressions are inserted directly into the event loop for the relevant events. The complete monitor is:

```
$include "evdefs.icn"
link evinit
procedure main(av)
  EvInit(av) | stop("can't monitor ", av[1])
  mask := E_Pcall ++ E_Pret ++ E_Pfail ++ E_Prem
  while EvGet(mask) do {
    if &eventcode == E_Pcall & &eventvalue === my_func then
      write("entering my_func, value of X is:", X)      # BEFORE
    if &eventcode == (E_Pret | E_Pfail | E_Prem) &
       &eventvalue=== my_func then
      write("leaving my_func, value of X is:", X)       # AFTER
    }
end
```

Implementation of Example #2. Another typical situation involves an aggregate operation and selection of events according to a given pattern. The SAY expression is implemented by a call to write(); it must be performed post-mortem since it uses parameters whose values are constructed during the entire program execution. CARD denotes a counter, while SUM denotes an accumulator +/; both require a variable that is initialized to zero. The event subtypes and constraints are used to generate addi-

tional conditional code in the body of the event processing loop. Lastly, some attributes such as r.duration require additional events and measurements besides the initial triggering event. In the case of r.duration, a time measurement between the function call and its return is needed.

```
$include "evdefs.icn"
link evinit
procedure main(av)
  EvInit(av) | stop("can't monitor ", av[1])
  cardreads := sumreadtime := 0
  mask := cset(E_Fcall)
  while EvGet(mask) do {
    ### count CARD of r:input...
    if &eventcode == E_Fcall & &eventvalue === (read|reads) then
      cardreads +:= 1
    ### add SUM of r.duration for r:input
    if &eventcode == E_Fcall & &eventvalue===(read|reads) then {
      thiscall := &time
      EvGet(E_Ffail++E_Fret)
      sumreadtime +:= &time - thiscall
      }
    }
  ### Translation of SAY
  write("Total number of read() statements: ", cardreads, "\n",
        "Elapsed time for read operations is: ",  sumreadtime)
end
```

Basic Generation Templates

The preceding handwritten example monitors use a single main loop that implements traditional event-driven processing. Monitors generated by the UFO compiler reduce complex assertions to this same single event loop. Keeping event detection in a single loop allows uniform processing of multiple event types used by multiple monitors. The code generated by the UFO compiler integrates event detection, attribute collection, and aggregate operation accumulation in the main event loop.

Assertions in UFO that use nested quantifiers entail two nested loops. Code generation flattens this loop structure, and postpones assertion processing until required information is available. A hybrid code generation strategy performs runtime processing whenever possible, delaying analyses until post-mortem time when necessary. Different assertions require different degrees of trace projection storage; code responsible for trace projection collection is also arranged within the main loop.

Each UFO rule falls in one of the following categories which determines its code generation template in the current implementation. We have not found a use for assertions requiring more than two nested quantifiers.

Compiler-Based Optimizations

The advantage of the UFO approach is the combination of an optimizing compiler for monitoring code with efficient run-time event detection and reporting. Since we know at compile time all necessary event types and attributes required for a given UFO program, the generated monitor is very selective about the behavior that it observes.

Type	FORMAN template	Pseudocode
I	Single quantifier. Rule applies to whole trace(prog_ex); evaluates at runtime.	See examples in Section 4.1.
II	Nested quantifiers of the form Quantifier A: Pattern_A Quantifier B: Pattern_B FROM A Body This requires accumulation of a trace projection for B-events and causes a mild overhead at runtime.	Main Loop Maintain stack of nested A events Accumulate events B in a B-list If end of event A Loop over B-list Do Body Endif If stack of A is empty Destroy B-list End of Main Loop
III	Nested quantifiers of the form Quantifier A: Pattern_A Quantifier B: Pattern_B FROM A .prev_path Body Accumulates a trace projection for B-events and may cause a heavy overhead at runtime. The B-list can not be deleted till the end of session.	Main Loop Maintain stack of nested A events Accumulate events B in a B-list If end of event A Loop over B-list If B precedes A Do Body Endif End of Main Loop
IV	Nested quantifiers of the form Quantifier A: Pattern_A Quantifier B: Pattern_B FROM A .following_path (or FROM prog_ex) Body Accumulates trace projections for both A and B events and causes a very heavy overhead at runtime.	Main Loop Accumulate events A in A-list Accumulate events B in B-list End of Main Loop # Postmortem Loop Loop over A-list Loop over B-list Do Body End of Postmortem Loop

For certain UFO constructs, such as nested quantifiers, monitors accumulate a sizable projection of the complete event trace and postpone corresponding computations until required information is available. The use of the previous_path and following_path attributes in UFO assertions facilitates this kind of optimization.

For further optimization, especially in the case of programs containing a significant number of modules, the following FORMAN construct limits event processing to events generated within the bodies of functions F1, F2, … , Fn.

```
WITHIN F1, F2, … , Fn DO
    Rules
END_WITHIN
```

This provides for monitoring only selected segments of the event trace.

Unicon expressions included in the value_at_begin and value_at_end attributes are evaluated at run time. Some other optimizations implemented in this version are:

- only attributes used in the UFO rule are collected in the generated monitor;
- an efficient mechanism for event trace projection management, which disposes from the stored trace projection those events that will not be used after a certain time (for example, see Category II);
- event types *and* context conditions are used to filter events for the processing.

UFO's goal of practical application to real-sized programs has motivated several improvements to the already-carefully-tuned Alamo instrumentation of the Unicon VM. We are working on additional optimizations.

6 Results of Sample Assertion Execution

Table 1 gives results from executing rules written in UFO on a sample target program, a 1,100 line version of egrep. Tests were run on a 700 MHz Solaris machine with 512MB of RAM. The results reported are number of events generated by the VM and execution time averaged over several runs. Execution time is reported as minutes:seconds.tenths. The second row contains the time for program execution without monitoring. Each program/input file combination was monitored by 8 different assertions corresponding to the basic generation templates.

Cases 1-4 are examples of a Category I template. Case 5 is a Category II rule. Case 6 is a Category III rule. It uses PREV_PATH and accumulates a trace projection over part of the program execution. Cases 7 and 8 contain nested quantifiers that belong to Category IV. These assertions require the accumulation of two trace projections over the entire program execution, and complete post-mortem processing. Case 9 is composed of all the previous assertions to yield a monitor that combines multiple assertions on a single execution of the target program.

Table 1. Results for igrep.icn.

Input Size (lines)	4000		16000		64000	
No monitoring	0.5		1.6		6.4	
	Events	**Time**	**Events**	**Time**	**Events**	**Time**
Case 1	184208	4.1	736208	16.2	2944208	1:04.9
Case 2	284123	4.6	1136123	18.1	4544123	1:12.9
Case 3	184208	3.4	736208	13.5	2944208	54.0
Case 4	184208	3.5	736208	13.6	2944208	54.0
Case 5	276306	6.3	1104306	28.0	4416306	2:09.3
Case 6	276306	6.5	1104306	28.4	4416306	2:11.8
Case 7	276306	6.5	1104306	29.1	4416306	2:11.3
Case 8	276306	6.5	1104306	29.4	4416306	2:12.6
Case 9	340306	45.9	1360306	3:57.8	5440306	20:38.6

The results depicted in this table allow several observations. Average monitoring speeds on simple assertions in the test environment were in the range of 2-3 million events per minute. Monitoring realistic assertions on real-size programs with real-size input data is feasible with this system. Most assertions impose about one order of magnitude execution slowdown compared with the unmonitored program execution.

The execution time required by the combination of all assertions (Case 9) is longer than the sums of separate monitor executions. Combined assertion executions have greater memory requirements in the current implementation, because separately collected trace projections compete for available cache and virtual memory resources. Multi-assertion optimizations are not yet implemented in the current UFO compiler.

7 Related Work

What follows is a very brief survey of basic ideas known in Debugging Automation to provide the background for the approach advocated in this paper.

The Event Based Behavioral Abstraction (EBBA) [10] characterizes the behavior of programs in terms of primitive and composite events. Context conditions involving event attributes are used to distinguish events. EBBA defines two higher-level means for modeling system behavior -- clustering and filtering. Clustering is used to express behavior as composite events, i.e. aggregates of previously defined events. Filtering serves to eliminate from consideration events, which are not relevant to the model being investigated. Both event recognition and filtering can be performed at run-time.

Event-based debuggers for the C programming language built on top of GDB include Dalek [11] and COCA [12]. Dalek supports user-defined events that typically are points within a program execution trace. A target program has to be manually instrumented in order to collect values of event attributes. Composite events can be recognized at run-time as collections of primitive events. COCA uses GDB for tracing and PROLOG for the execution of debugging queries. It provides an event grammar for C and event patterns based on attributes for event search. The query language is designed around special primitives built into the PROLOG query evaluator.

Assertion languages provide another approach to debugging automation. Boolean expressions are attached to points in the target program, like the assert() macro in C. [13] advocates a practical approach to programming with assertions for the C language, and demonstrates that even local assertions associated with particular points within the program may be extremely useful for program debugging.

The ANNA [14] annotation language for the Ada language supports assertions on variable and type declarations. The TSL [15], [16] annotation language for Ada uses events to describe the behavior of Tasks. Patterns can be written which involve parameter values of Task entry calls. Assertions are written in Ada using a number of special pre-defined predicates. Assertion-checking is performed at run-time. RAPIDE [17] provides an event-based assertion language for software architecture description. Temporal Rover is a commercial tool for dynamic analysis based on temporal logics [18]. The DUEL [19] debugging language introduces expressions for C aggregate data exploration, for both assertions and queries.

Algorithmic debugging was introduced in [20] for the Prolog language. In [21] and [22] this paradigm is applied to a subset of PASCAL. The debugger executes the program and builds a trace execution tree at the procedure level while saving some

useful trace information such as procedure names and input/output parameter values. The debugger traverses the execution tree, asking the user about the intended behavior of each procedure. The search finally ends and a bug is localized within a procedure p when one of the following holds: procedure p contains no procedure calls, or all procedure calls performed from the body of procedure p fulfill the user's expectations. The notion of computation over execution trace introduced in FORMAN is a generalization of Algorithmic Debugging and is a convenient basis for describing such generic debugging strategies.

PMMS [23] is a high level program monitoring and measuring system. This system works by receiving queries from the user about target programs written in the AP5 high level programming language. PMMS instruments the source code of the target program in order to gather data necessary to answer the posed questions. This data is collected during run time by the monitoring facilities of PMMS and stored in a database for subsequent analysis. Their domain specific query language is similar to FORMAN but tailored for database-style query processing.

JPAX [9], the Java Path Explorer, provides a means to check execution events within a program based on a user provided specification written in Maude, a high level logic language. Like UFO, JPAX supports monitoring based on a VM (JVM). JPAX supports both black box (based on automatic byte-code instrumentation) and white box (based on hand instrumentation) runtime verification.

Dynascope [24] is a system for directing programs written in vanilla C. A director monitors and controls the actions of the program, while an interpreter controls the flow of event streams to and from the director in addition to interpreting the program. Dynascope can test and debug programs without altering their source code.

YODA [25] uses a preprocessor to attach statements to a target Ada program. These statements activate a monitor creates a trace database and a symbol table to aid in debugging. The trace database will contain the program's history regarding variable declaration and use, task synchronization, and change in task status. Prolog queries can be issued by the user in order to confirm or reject hypotheses about program behavior. YODA represents a classical post-mortem trace processing paradigm.

8 Conclusions and Future Work

The rising popularity of virtual machine architectures enables dramatic improvements in automatic debugging. These improvements will only occur if debugging is one of the objectives of the VM design, e.g. as in the case of .net [26].

The architecture employed in UFO could be adapted for a broad class of languages such as those supported by the Java VM or the .net VM. Our approach to debugging automation uniformly represents many types of debugging-related activities as computations over traces, including assertion checking, profiling and performance measurements, and the detection of typical errors. We have integrated event trace computations into a monitoring architecture based on a VM. Preliminary experiments demonstrate that this architecture is scalable to real-world programs.

One of our next steps is to build a repository of formalized knowledge about typical bugs in the form of UFO rules, and gather experience by applying this collection of assertions to additional real-world applications. There remain many optimizations that will improve the monitor code generated by the UFO compiler, for example,

merging common code used by multiple assertions in a single monitor, and generating specialized VMs adjusted to the generated monitor.

Acknowledgements

This work has been supported in part by U.S. Office of Naval Research Grant # N00014-01-1-0746, by U.S. Army Research Office Grant # 40473--MA-SP, by the NSF Grant # EIA 02-20590, and by the National Library of Medicine.

References

1. Communications of the ACM, Vol.4, 1997.
2. F. Brooks, The Mythical Man-Month. Addison-Wesley, Reading, MA, 1975.
3. Mikhail Auguston, Program Behavior Model Based on Event Grammar and its Application for Debugging Automation, Proceedings of the 2nd Int'l Workshop on Automated and Algorithmic Debugging, Saint-Malo, France, May 1995, pp. 277-291.
4. M. Auguston, A. Gates, M. Lujan, "Defining a Program Behavior Model for Dynamic Analyzers", in Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering, SEKE'97, Madrid, Spain, June 1997, pp. 257-262.
5. M. Auguston, "Lightweight semantics models for program testing and debugging automation", Proceedings of the 7th Monterey Workshop, June 2000, pp.23-31.
6. Clinton L. Jeffery, Program Monitoring and Visualization: an Exploratory Approach. Springer, New York, 1999.
7. Clinton Jeffery, Shamim Mohamed, Ray Pereda, and Robert Parlett, "Programming with Unicon", http://unicon.sourceforge.net.
8. Ralph E. Griswold and Madge T. Griswold, The Icon Programming Language, 3rd edition. Peer to Peer Communications, San Jose, 1997.
9. K. Havelund, S. Johnson, G. Rosu. "Specification and Error Pattern Based Program Monitoring", ESA Workshop on On-Board Autonomy, Noordwijk, Holland, Oct. 2001.
10. P. C. Bates, J. C. Wileden, "High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach", Journal of Systems and Software 3, 1983, pp. 255-264.
11. R. Olsson, R. Crawford, W. Wilson, "A Dataflow Approach to Event-based Debugging", Software -- Practice and Experience, Vol.21(2), February 1991, pp. 19-31.
12. M. Ducasse, "COCA: An automated debugger for C", in Proceedings of the 1999 International Conference on Software Engineering, Los Angeles, 1999, pp. 504-513.
13. D. Rosenblum, "A Practical Approach to Programming with Assertions", IEEE Transactions on Software Engineering, Vol. 21, No 1, January 1995, pp. 19-31.
14. D. C. Luckham, S. Sankar, S. Takahashi, "Two-Dimensional Pinpointing: Debugging with Formal Specifications", IEEE Software, Vol. 8, No 1, January 1991, pp.74-84.
15. D. C. Luckham, D. Bryan, W. Mann, S. Meldal, D. P. Helmbold, "An Introduction to Task Sequencing Language, TSL version 1.5", Stanford University, Feb. 1990, pp. 1-68.
16. D. Rosenblum, "Specifying Concurrent Systems with TSL", IEEE Software, Vol. 8, No 3, May 1991, 52-61.
17. D. Luckham, J. Vera, "An Event-Based Architecture Definition Language", IEEE Transactions on Software Engineering, Vol.21, No. 9, 1995, pp. 717-734.
18. D. Drusinsky, The Temporal Rover and the ATG Rover, LNCS #1885, pp.323-330, Springer, 2000.

19. M. Golan, D. Hanson, “DUEL - A Very High-Level Debugging Language”, in Proceedings of the Winter USENIX Technical Conference, San Diego, Jan. 1993.
20. E. Shapiro, “Algorithmic Program Debugging”, MIT Press, May 1982.
21. P. Fritzson, N. Shahmehri, M. Kamkar, T. Gyimothy, “Generalized Algorithmic Debugging and Testing”, ACM LOPLAS, Vol 1 (4), Dec 1992.
22. N. Shahmehri, “Generalized Algorithmic Debugging”, Ph.D. Thesis No. 260, Dept. of Computer and Information Science, Linköping University, S-581 83 Linköping, 1991.
23. Y. Liao, D. Cohen, “A Specificational Approach to High Level Program Monitoring and Measuring”, IEEE Transactions on Software Engineering, Vol 18, No 11, Nov 1992, pp.969 – 978.
24. R. Sosic, “Dynascope: a Tool for Program Directing”, Sigplan Notices 27(7), pp.12-21, 1992.
25. LeDoux, Carol H. and Parker, D., “Saving Traces for Ada Debugging. Ada in Use”, Proc. of the Ada International Conference, ACM Ada Letters, 5(2), pp.97-108, Sep 1985.
26. http://www.microsoft.com/net/

Appendix. Syntax for UFO Rules

```
Rules::=   ( (  Rule | Within_group  ) ';') +
Within_group::= 'WITHIN'  Procedure_name  ( ',' Procedure_name ) *
       'DO' ( Rule ';' ) + 'END_WITHIN'
Rule::= [ Label ':' ]
      [ ('FOREACH' | 'FIND') Pattern  [ 'FROM' 'PROG_EX' ] ]
      [ ('FOREACH' | 'FIND') Pattern  [ 'FROM'   ('PROG_EX'   |
                    Metavariable [ '.' ( 'PREV_PATH' | 'FOLLOWING_PATH' )] )  ]
         [ 'SUCH' 'THAT' ]   Bool_expr
         [['WHEN' 'SUCCEEDS'] Say_clause + ]   ['WHEN' 'FAILS'    Say_clause + ]
Say_clause ::= 'SAY' '('  (  Expression | Metavariable | Aggregate_op )  *  ')'
Bool_expr::= Bool_expr1 ( 'OR'  Bool_expr1 )*
Bool_expr1::= Bool_expr2 ( 'AND'  Bool_expr2 )*
Bool_expr2::=   Expr   [ ( '=' | '==' | '>' | '<' | '>=' | '<=' | '|=' )  Expr ] | 'NOT' Bool_expr2  |
                    '('  Bool_expr ')'
Pattern::=    Metavariable ':' Event_type [ '&' Bool_expr ]
Aggregate_op::=   [ ( 'CARD' | 'SUM' ) ] '['   Pattern
   ['FROM' ('PROG_EX' | Metavariable [ '.' ( 'PREV_PATH'|'FOLLOWING_PATH' )]  )]
   [ 'APPLY' ( Bool_expr | Expression ) ]  ']'
Expression::=    Expr1  (* ( '+' | '-') Expr1 *)
Expr1::=  Simple_expr  ( ( '*' | 'DIV' | 'MOD' ) Simple_expr )*
Simple_expr::=  '-' Simple_expr | integer | Aggregate_op |
                   Metavariable '.'  Attribute  | string | '('   Expr  ')'
Attribute::= (SOURCE_TEXT | LINE_NUM | COL_NUM | TIME_AT_END |
           TIME_AT_BEGIN | COUNTER_AT_END | COUNTER_AT_BEGIN |
           DURATION | VALUE | OPERATOR | TYPE | FAILURE | FUNC_NAME |
           ( PARAM_NAMES '[' integer ']' ) | FILE_NAME | ADDRESS |
           ( VALUE_AT_BEGIN | VALUE_AT_END)  '(' Unicon_expr ')'  )
Event_type::=  (func_call | expr_eval | input | output | variable | literal |
           lhp | rhp | clause | iteration | test )
```

Automation of Software System Development Using Natural Language Processing and Two-Level Grammar

Beum-Seuk Lee and Barrett R. Bryant

Department of Computer and Information Sciences
The University of Alabama at Birmingham
Birmingham, AL 35294-1170 USA
{leebs,bryant}@cis.uab.edu

Abstract. In software engineering, even with recent active research on formal methods and automated tools, users' involvement is inevitable and crucial throughout the software development lifecycle. Automation of these manual tasks would assist the developers throughout the development. Our project goal is to help the engineers to resolve ambiguity in natural language (NL) using Natural Language Processing and to overcome different levels of abstraction between requirements documents and formal specifications using Two-Level Grammar (TLG). The result is a system that assists developers to build a formal representation from the informal requirements for rapid prototyping and complete system implementation.

Keywords: Natural Language Processing, Formal Specification, Automated Software Engineering, Two-Level Grammar (TLG).

1 Introduction

Even the rigorous development of formal specifications and automated tool kits in recent years hasn't eliminated the practical importance of requirements documents written in natural language and the necessity of users' involvement throughout the software development life cycle.

Even though natural language is inherently object-oriented and descriptive with strong representation power, its syntax and semantics are not formal enough to be used directly as a programming language. Therefore the requirements documentation written in NL has to be reinterpreted into a formal specification language by software engineers. Pohl rightly stated regarding this process that improving an opaque system comprehension into a complete system specification and transforming informal knowledge into formal representations are the major tasks in the requirements engineering [1]. When the system is very complicated, which is mostly the case when one chooses to use formal specification, this conversion, if manually done, is both non-trivial and error-prone, if not implausible.

Many similar tasks of manual involvement occur and are repeated to translate the requirements documents into a formal specification or into final executable

M. Wirsing et al. (Eds.): RISSEF 2002, LNCS 2941, pp. 219–233, 2004.

code regardless if the type of the system under development. Some examples of these tasks are domain-specific knowledge collection, correct interpretation of requirements, specification update, and maintenance of consistency, to name a few.

It is well known that as much as 60 percent of the errors that appear during a system's life cycle have their origin in the requirements phase [2]. It is also well known that the cost to correct an error found in the implementation and later stages of system development is orders of magnitude higher than to correct the same error found during the requirements stage [3]. Therefore ensuring the correctness of the requirements as well as their interpretation and translation cannot be overemphasized.

The challenge of formalizing a natural language requirements document, which takes up major portion of human involvement in the system development, results from many factors such as miscommunication between domain experts and engineers. However the major bottleneck of this conversion is from the inborn characteristic of ambiguity of NL and the different levels of formalism between the two domains of NL and formal specification.

To handle this ambiguity problem, some have argued that the requirements document has to be written in a particular way to reduce ambiguity in the document [4]. Others have proposed controlled natural languages (e.g., Attempto Controlled English (ACE) [5]) which limit the syntax and semantics of NL to avoid the ambiguity problem. Another approach to NL requirements analysis is to search each line of the requirements document for specific words and phrases for the purpose of quality analysis [6]. A similar project [7] focuses mainly on the automatic indexing and reuse of the software components in the requirement documents. However there has been no attempt to automate the conversion from requirements documentation into a formal specification language for prototyping as well as implementation.

In our research, Natural Language Processing (NLP) [8] is used to handle the ambiguity problem in NL and Two Level Grammar (TLG) [9] is used to deal with the different formalism level between NL and formal specification languages to achieve the automated conversion from NL requirements documentation into a formal specification (in our case VDM++ - an object-oriented extension of the Vienna Development Method [10]) and to reduce and reuse the developers involvement.

2 Approach

To achieve the conversion from requirements documents to a formal specification several levels of conversions are required. First the original requirements written in natural language is refined as a preprocessing of the actual conversion. This refinement task involves checking spellings, grammatical errors, consistent use of vocabularies, organizing the sentences into the appropriate sections, etc. This process is carried out manually by the requirements engineer(s) and it should be noted that a well-written requirements document will need little if any prepro-

cessing. Next the refined requirements document is automatically translated into XML format. By using XML to specify the requirements, XML attributes (metadata) can be added to the requirements to interpret the role of each group of the sentences during future conversions. The information of the domain-specific knowledge is specified in XML. The domain-specific knowledge describes the relationship between components and other constraints that are presumed in requirements documents or too implicit to be extracted directly from the original documents, and it is constructed by the domain engineer(s).

Then a knowledge base is automatically constructed from the requirements document in XML using NLP to parse the documentation and to store the syntax, semantics, and pragmatics information. In this phase, the ambiguity is detected and resolved, if possible. If there is difficulty resolving the ambiguity, the system may seek assistance from the software engineer(s). Once the knowledge base is constructed, its content can be queried in NL. Next the knowledge base is automatically converted, with the information of the domain specific knowledge, into Two Level Grammar (TLG) by removing the contextual dependency in the knowledge base. TLG, the most NL-like specification language which is a unification of functional, logic, and object-oriented programming styles, is used as an intermediate representation to build a bridge between the informal knowledge base and the formal VDM++ representation.

Finally the TLG code is translated into VDM++ by data and function mappings. VDM++ is chosen as the target specification language because VDM++ has many similarities in structure to TLG and also has a good collection of tools for analysis and code generation. Once the VDM++ representation of the specification is acquired we can do prototyping of the specification using the VDM++ interpreter. Also we can convert this into a high level language such as Java^{TM} or C++ or into a model in the Unified Modeling Language (UML) [11] using the VDM++ Toolkit [12]. The entire system structure is shown in Figure 1.

The translation of our system is incremental and iterative reflecting the changes made throughout the system development. The user interaction is likely to happen any stage of the translation to supervise and assist the automation. By keeping track of user's preferences and configurations for each iteration and automating the translations accordingly, the user's involvement can be reasonably reduced.

In the sections which follow, we will present the following simplified (thus incomplete) Computer Assisted Resuscitation Algorithm (CARA) [13] to illustrate our approach and describe the various system components. CARA is a closed loop infusion pump control software system that drives a high output infusion pump used for fluid resuscitation of patients suffering from conditions that lead to hypotension.

```
HOST is powered up and all software subsystems are available.
The pump software system is now in the wait operating state. Patient
with IV/pump running is placed onto the HOST. Pump cable is connected
to the HOST. HOST now provides power for pump. Pump software system
detects pump connection and monitors occlusion and airlock logic levels.
Pump subsystem display is automatically brought forward to the secondary
```

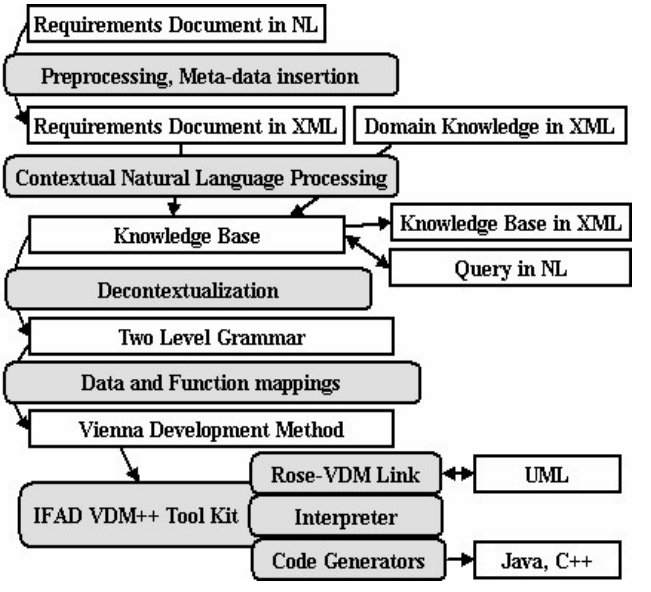


Fig. 1. System Structure.

```
display. Pump software subsystem detects back EMF and fluid impedance
and begins to log infusion rate. Pump continues to operate on it's
hardware setting. Pump software system is now in manual operating state.
One of the blood pressure sensors is connected to the patient.
Pump software system detects clean blood pressure signal and activates
automatic servo-control start button. When the start button is pressed
the MAC controls the pump and begins resuscitation to the prescribed
blood pressure setpoint. The system is now in the automatic
servo-control on operating state when the pump is infusing fluid into
a patient using the hardware(HW) flow setting on the pump. If for any
reason (change IV bags, change or fix blood pressure sensor, etc.) it
becomes necessary to pause the MAC, the pause button on the display may
be pressed. This causes the infusion pumping to cease. The system is
now in the automatic servo-control paused operating state. The system
maybe restarted at any time. When the patient is to be removed from
the HOST, the pump software system should be returned to the manual
operating state. The blood pressure sensor should be removed from the
patient and then the pump cable can be removed from the HOST.
This allows the pump to continue operating in standalone mode or the
IV infusion to be discontinued.
```

3 Requirements in XML

Rearranging related information together in the requirements will ease the conversion. Specially because we are assuming that the requirements can contain

different aspects of information (functional, non-functional or even a mixture of both) about the system. Even functional requirements can have different types of functionality. For example, they can be object-oriented, procedural, real time-based, event-based, etc. Rearranging related information together will ease the conversion. This can be achieved by specifying the role of each paragraph using XML data structure and notations. This will help the knowledge base to maintain the correct structure.

The CARA specification in XML is shown as follows.

```
<document>
<c title = "Mode" meta = "mode">
 <c title = "wait state" meta = "submode">
  <p meta = "pre_cond">
   <s>HOST is powered up and all software subsystems are available</s>
  </p>
  <p meta = "pre_exec">
   <s>Patient with IV/pump running is placed onto the HOST</s>
   <s>Pump cable is connected to the HOST</s>
  </p>
  <p meta = "exec">
   <s>HOST now provides power for pump</s>
  </p>
  <p meta = "break_cond">
   <s>When the pump is infusing fluid into a patient using
      the hardware (HW) flow setting on the pump the system is no
      longer in the wait state</s>
  </p>
 </c>
 <c title = "manual state" meta = "submode">
  <p meta = "pre_exec">
   <s>Pump software system detects pump connection and monitors occlusion
      and airlock logic levels </s>
   <s>Pump subsystem display is automatically brought forward to the
      secondary display</s>
   <s>Pump software subsystem detects back EMF and fluid impedance and
      begins to log infusion rate</s>
   <s>Pump continues to operate on it's hardware setting</s>
   <s>One of the blood pressure sensors is connected to the patient</s>
   <s>Pump software system detects clean blood pressure signal and
      activates automatic servo-control start button</s>
  </p>
 </c>
 <c title = "autocontrol on state" meta = "submode">
  <p meta = "pre_exec">
   <s>When the start button is pressed the MAC controls the pump and
      begins resuscitation to the prescribed blood pressure
      setpoint</s>
  </p>
 </c>
```

```
 <c title = "autocontrol paused state" meta = "submode">
  <p meta = "pre_exec">
   <s>If for any reason (change IV bags, change or fix blood pressure
      sensor, etc.) it becomes    necessary to pause the MAC, the
      pause button on the display may be pressed</s>
   <s>This causes the infusion pumping to cease</s>
  </p>
  <p meta = "break_cond">
   <s>The system maybe restarted at any time</s>
  </p>
  <p meta = "break_exec">
   <s>When the patient is to be removed from the HOST, the pump software
      system should be returned to the manual operating state</s>
   <s>The blood pressure sensor should be removed from the patient and
      then the pump cable can be removed from the HOST</s>
   <s>This allows the pump to continue operating in standalone mode or
      the IV infusion to be discontinued</s>
  </p>
 </c>
</c>
</document>
```

The meta attribute in XML indicates the role of each paragraph. Namely it shows if the group of the sentences describes state types (`mode`), execution types (`_exec`), various conditions (`_cond`), etc. `submode` indicates the state. In the CARA example, there are four distinctive states; wait state, manual state, autocontrol on state, and autocontrol paused state. In a state, preconditions (`pre_cond`) have to be satisfied to enter the state. Some statements (`pre_exec`) will be executed when the system enters into a state. Other statements (`exec`) will be executed while the system is in the state. If any break conditions (`break_cond`) are satisfied in the state, the system will leave the state. There may be some cases where break conditions will execute some statements (`break_exec`) before breaking out of the state. Also some default statements (`post_exec`) are executed before leaving the state. We have specified these meta attributes for various types of functionality in requirements to cover a wide range of different requirements documents. Using a tree-like structure in XML the specifications become more descriptive as the tree expends further. Organizing and representing the requirements document in XML according to the roles of the specifications of the system not only enhances understanding of specifications but also helps to standardize requirements composition.

4 Domain-Specific Knowledge in XML

A requirements document usually contains specific information about how the system should work whereas the domain knowledge describes how the system is composed by its components and the constraints imposed on the components or on the relations among them. The domain-specific knowledge is a world knowledge specific to a certain domain in which the system is defined. This is well

tied into the concept of the family or the ontology of systems. Depending on the level of abstraction (or the details described) of the domain knowledge, the effort to construct it can vary. By limiting the level of abstraction, the body of the knowledge can be reduced into a reasonable size and so can the effort to build it. Usually the domain-specific knowledge is defined informally or only for a specific project, not reusable or extensible for similar systems (the systems in the same family). By using XML to specify the domain knowledge with a minimum semantics, not only can the specification be formally defined but also it can be extensible, gradually building up an ontology of systems.

In our research the domain knowledge specified in XML shares many similarities with DARPA Agent Markup Language (DAML) [14] which is a frame-based language with semantics to describe ontology. Because domain knowledge is more than just an ontology, DAML is not expressive enough to describe the whole aspect of the domain knowledge [15]. However using the XML syntax a domain knowledge can be specified in various ways leaving the interpretation of its semantics totally up to the system that uses it [16]. Therefore when a specification for domain-specific knowledge in XML is to be developed, its formal semantics as well as its expressiveness has to be considered at the same time.

The following describes an example of the domain knowledge of `Car` to illustrate the use of domain-specific knowledge expressed in XML in our project.

```
<system name = "Car">
 <component name = "Engine">
  <amount type ="exactly" value = "1"/>
  <unit type = "volume" value = "liter"/>
  <subcomponent name="Cylinder" type = "integer">
   <amount type ="one_of" value = "4,6,8"/>
  </subcomponent>
  <relation with = "Starter" type ="pass_to" value ="signal"/>
 </component>
 <component name = "Wheel"/>
 <component name = "Body">
  <relation with = "Frame" type ="synonym"/>
 </component>
 <relation with = "Vehicle" type ="inheritance" value ="parent"/>
 <relation with = "Van" type ="inheritance" value ="child"/>
</system>
```

According to the above domain specification, `Car` is composed of `Engine`, `Wheel`, `Control`, and `Body`. `Vehicle` is a parent of `car` whereas `Van` is a type of `Car`. `Car` can have exactly one `Engine` and the unit of `engine` is a volume expressed in liters. `Engine` has `Cylinder` as its subcomponent. The number of `Cylinder`s, which is an integer and is a representative part of the subcomponent, can be either 2, 6, or 8. `Starter` passes a signal to `Engine` (to turn the motor on). `Body` of `Car` also can be referred to as `Frame`.

The following is a Document Type Definition (DTD) for the domain knowledge in XML, which defines the formal semantics of the domain-specific knowledge while retaining proper expressive power.

```
<!ELEMENT system (component|relation)*>
<!ELEMENT component (amount?, unit?, (subcomponent|relation)*)>
<!ELEMENT subcomponent amount?>
<!ELEMENT amount EMPTY>
<!ELEMENT unit EMPTY>
<!ELEMENT relation EMPTY>

<!ATTLIST system name CDATA #REQUIRED>
<!ATTLIST component name CDATA #REQUIRED>
<!ATTLIST subcomponent name CDATA #REQUIRED type CDATA #IMPLIED>
<!ATTLIST amount type CDATA "exactly" value CDATA #REQUIRED>
<!ATTLIST unit name CDATA #IMPLIED type CDATA #REQUIRED>
<!ATTLIST relation with CDATA #IMPLIED type CDATA #REQUIRED value
 CDATA #IMPLIED>
```

Note that the domain-specific knowledge in XML for the translation doesn't have to describe the domain exhaustively. Namely most of the elements and attributes are optional and attribute values can be any character strings. For example, the relationship element can represent inheritance, acronyms, message passing, etc. The minimum information required to guide the translation would be sufficient with the possibility of adding on more information later when necessary.

The domain knowledge for our CARA example is shown as follows.

```
<system name = "CARA system">
 <component name = "Computer Assisted Resuscitation Algorithm">
  <subcomponent name = "Display"/>
  <subcomponent name = "Button"/>
  <subcomponent name = "Pump Software System"/>
  <subcomponent name = "MAC"/>
  <relation with = "System" type = "hypernym"/>
  <relation with = "Software" type = "hypernym"/>
  <relation with = "Algorithm" type = "hypernym"/>
  <relation with = "CARA" type = "acronym"/>
 </component>
 <component name = "Patient"/>
 <component name = "HOST">
   <subcomponent name = "Pump"/>
 </component>
</system>
```

The above specification describes that the whole system is composed by `Computer Assisted Resuscitation Algorithm`, `Patient`, and `HOST`. `Computer Assisted Resuscitation Algorithm` is a type of `Algorithm, Software,` or `System` that can be abbreviated as `CARA`.

In the natural language documents one concept can be represented by many different ways making the translation hard to cluster similar information together. These can be acronym, synonym, and hypernym. From the CARA example, the word Computer Assisted Resuscitation Algorithm' is interchangeable

with 'Algorithm' or 'CARA'. By using a minimum set of representative words that describes the entire components in the domain-specific knowledge, one-to-many relations between words and their various representations can be obtained and thus provides a simpler source to translate. The full set of words in the requirements documents are mapped into the minimum set of representative words by measuring similarity among words. The hypernym and the location of the common words are used for this estimation.

In summary, by specifying domain-specific knowledge in XML and limiting the scope of the knowledge, the effort needed to build up the domain knowledge for the translation can be greatly reduced.

5 Conversion from XML to Knowledge Base

The raw information of the requirements document in natural language is not in the proper form to be used directly because of the ambiguity and implicit semantics in the document. Therefore an explicit and declarative representation (knowledge base) is needed to represent, maintain, and manipulate knowledge about a system domain [17]. Not only does the knowledge base have to be expressive enough to capture all the critical information but also it has to be precise enough to clarify the meaning of each knowledge entity (sentence). In addition, the knowledge base has to reflect the structure of TLG into which the knowledge base is translated later.

The knowledge base isn't a simple list of sentences in the requirements document. The linguistic information of each sentence such as lexical, syntactic, semantic, and most importantly discourse level information has to be stored with proper systematic structure.

Each sentence of the requirements documents has to be represented in a way that eases the interpretation of the sentence. In computational linguistics this is done by constructing a parse tree of the sentence, which contains the syntactic information of the sentence. By using this semantic information we can tell what type of operation a certain object executes on other objects.

To build a parse tree, each sentence in the requirements document is read by the system and tokenized into words. At the syntactical level, the part of speech (e.g. noun, verb, adjective) and the part of sentence (e.g. subject and object) of each word are determined by standard parsing techniques [8]. The corpora of statistically ordered parts of speech (frequently used ones being listed first) of about 85,000 words from Moby Part-of-Speech II [18] are used to resolve the syntactic ambiguity when there is more than one valid parsing tree. The system is able to handle elliptical compound phrases, comparative phrases, compound nouns, and relative phrases to allow the natural language in the requirements documents to be less controlled thus more natural.

Also the anaphoric references (pronouns) in a sentence are identified according to the current context history. A pronoun can represent a word, sentence, or even context. It is worthwhile to mention here that the requirements documents are easier to process than other types of textual documents in the sense that usu-

ally requirements documents have well defined structures with less ambiguities and infrequent use or narrow reference scope of pronouns.

Once the references of pronouns are determined, each sentence is stored into the proper context in the knowledge base. The structure of knowledge base reflects the structure of the requirements in XML. The meta attribute information from XML is also stored in the knowledge base to be used for the translation from knowledge base into TLG. If no meta attribute or data structure is specified in the requirements in XML, the system totally relies on the linguistic information in the document to build the knowledge base according to the context. For more information on this process, we refer the readers to [19]. A part of the CARA knowledge base is shown in Figure 2. The knowledge base of CARA system con-

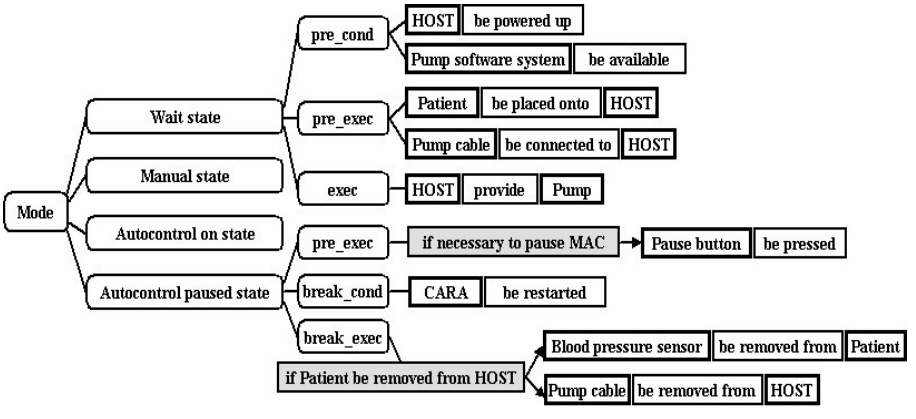


Fig. 2. Knowledge base for CARA.

tains the meta information from the XML requirements in its tree-like structure as well as the linguistic knowledge.

In summary, the knowledge base stores not only the linguistic information of each sentence but also the data structure and meta information of related sentences as specified in the requirements in XML. Along with this process, linguistic ambiguity is detected and resolved in parsing and construction of the knowledge base.

6 Transition from Knowledge Base to TLG

Two-Level Grammar (TLG) may be used to achieve translation from an informal NL specification into a formal specification. Even though TLG has NL-like syntax its notation is formal enough to allow formal specifications to be constructed using the notation. It is able not only to capture the abstraction of the requirements but also to preserve the detailed information for implementation. The term “two level” comes from the fact that a set of domains may be defined

using context-free grammar, which may then be used as arguments in predicate functions defined using another grammar. TLG may be used to model any type of software specification. The basic functional/logic programming model of TLG is extended to include object-oriented programming features suitable for modern software specification [20]. The syntax of the object-oriented TLG is:

```
class Class_Name.
  Data_Name {, Data_Name}::Data_Type {, Data_Type}.
  Rule_Name : Rule_Body {, Rule_Body}.
end class [Class_Name].
```

where the term that is enclosed in the curly brackets is optional and can be repeated many times, as in Extended Backus-Naur Form (EBNF). The data types of TLG are fairly standard, including both scalar and structured types, as well as types defined by other class definitions. The rules are expressed in NL with the data types used as variables. Further details about the TLG language may be found in [9].

The conversion from the knowledge base to TLG flows very nicely because the knowledge base is built with the structure taking this translation into consideration. The root of each context tree becomes a class. And then the body of each class is built up with the sentence information in the sub-contexts of the root. Combined with the specification in the domain-specific knowledge, the knowledge base of the CARA example would be translated into the following TLG specification.

```
class Mode.

 HOST, MAC, Pump :: Component.
 Pump_Software_System :: Software.
 Blood_Pressure_Sensor :: Sensor.
 Patient :: Person.
 Secondary_Display, Pause_Button :: Interface.
 Automatic_Servo-control_Start_Button :: Interface.
 Hardware_Setting :: Setting.
 Occlusion, Airlock_Logic_Levels :: Float.
 Back_EMF, Fluid_Impedance, Infusion_Rate :: Float.
 Clean_Blood_Pressure_Signal :: Float.
 Resuscitation, Infusion_Pumping :: Undefined.
 Prescribed_Blood_Pressure_Setpoint :: Undefined.
 IV_Infusion, Standalone_Mode :: Undefined.

 main :
  wait state;
  manual state;
  autocontrol on state;
  autocontrol paused state.
```

```
 wait state:
  HOST be powered up,
  Pump_Software_System be available,
  Patient be placed onto HOST,
  Pump Cable be connected to HOST,
  while true then
    if Pump be infusing Fluid into Patient then
      break,
    HOST provide Power for Pump
  endwhile.

 manual state:
  Pump_Software_System detect Pump_Connection,
  Pump_Software_System monitor Occlusion and Airlock_Logic_Levels,
  Pump Display be brought to Secondary_Display,
  Pump_Software_System detect Back_EMF and Fluid_Impedance,
  Pump_Software_System begin to log Infusion_Rate,
  Pump continue to operate on Hardware_Setting,
  Blood_Pressure_Sensor be connected to Patient,
  Pump_Software_System detect Clean_Blood_Pressure_Signal,
  Pump_Software_System activate Automatic_Servo-control_Start_Button.

 autocontrol on state:
  if Start_Button be pressed then
   MAC control Pump,
   MAC begin Resuscitation to Prescribed_Blood_Pressure_Setpoint
  endif.

 autocontrol paused state:
  if necessary to pause MAC then
   Pause_Button be pressed,
   cause Infusion_Pumping to cease
  endif,
  while true then
   if Patient be removed from HOST then
    Blood_Pressure_Sensor be removed from Patient,
    Pump Cable be removed from HOST,
    allow Pump to continue operating in Standalone_Mode,
    allow IV_Infusion to be discontinued,
    break
   endif
  endwhile.

end class.
```

The `main` function will execute all 4 state functions (`wait state`, `manual state`, `autocontrol on state`, `autocontrol paused state`) nondeterministically, indicated by separating these by semicolons (;) whereas a comma (,) implies sequential composition. However preconditions (`pre_cond`) in each state will be used as guarded statements to determine which state the system is currently in.

For each state function, first the preconditions will be checked. If all the preconditions are met, `pre_exec` statements are executed once. If there is a while loop, `exec` statements are executed. `break_exec` and `break_cond` statements are used for the system to break out this loop. If there are any `post_exec` statements, they are executed before returning from the function.

The TLG code is translated into VDM++ by data and function mappings (for more details on this translation we refer the readers to [9]). Once we have translated the TLG specification into a VDM++ specification we can convert this into a high level language such as Java^{TM} or C++, using the code generator that the VDM++ Toolkit^{TM} provides. Not only is this code quite efficient, but it may be executed, thereby allowing a proxy execution of the requirements. This allows for a rapid prototyping of the original requirements so that these may be refined further in future iterations. Namely the inconsistencies, contradictions, and ambiguities hidden in the informal description can be discovered in the formal representation using the VDM++ Toolkit. Another advantage of this approach is that the VDM++ Toolkit also provides for a translation into a model in the Unified Modeling Language (UML) using a link with Rational Rose^{TM}.

7 Contribution and Conclusion

This research project is developed as an application of formal specification and computational linguistic techniques to automate the conversion from a requirements document written in NL to a formal specification language while assisting the developers with repetitive tasks. The knowledge base is built up from a NL requirements document in XML in order to capture the contextual information from the document while handling the ambiguity problem and to optimize the process of its translation into a TLG specification with aid of domain-specific knowledge in XML. Due to its NL-like flexible syntax without losing its formalism, TLG is chosen as a formal specification to fill the gap between the different level of formalisms of NL and formal specification language.

The system is working for some small examples such as the requirements for an Automatic Teller Machine (ATM), with associated banking system domain knowledge. For various, more complex, requirements documents, such as the CARA Infusion Pump Controller, the system has been useful in identifying problems and ambiguities with such specifications and in identifying additional information necessary to complete the implementation. It is expected that the technology we are developing will be able to produce implementation for these requirements documents as well, once the problems identified are corrected.

This system is a very useful tool to assist software engineers in moving from the requirements document to the formal specification. Our future work is to continue developing the system to improve system usability and robustness with respect to its coverage of requirements documents. When finalized, it is expected that by using the formalized context in NLP and TLG as a bridge between the requirements document and a formal specification language, we can achieve an

executable and reusable NL specification for a rapid prototyping of requirements, as well as development of a final implementation assisting the developers throughout the software development life cycle.

Acknowledgements

This material is based upon work supported by, or in part by, the U.S. Army Research Laboratory and the U. S. Army Research Office under contract/grant number DAAD19-00-1-0350 and by the U. S. Office of Naval Research under award number N00014-01-1-0746. The authors would like to thank IFAD for providing an academic license to the IFAD VDM Toolbox in order to conduct this research.

References

1. Pohl, K.: The Three Dimensions of Requirements Engineering. Conference on Advanced Information Systems Engineering (1993) 275–292
2. Davis, A.: Software Requirements Analysis and Specification. Prentice-Hall (1990)
3. Boehm, B.W.: Software Engineering Economics. IEEE Transactions on Software Engineering **10** (1984) 4–21
4. Wilson, W.M.: Writing Effective Natural Language Requirements Specifications. Technical report, Naval Research Laboratory (1999)
5. Fuchs, N.E., Schwitter, R.: Attempto Controlled English (ACE). Proc. CLAW 96, 1st Int. Workshop Controlled Language Applications (1996)
6. Wilson, W.M., Rosenberg, L.H., Hyatt, L.E.: Automated Quality Analysis Of Natural Language Requirement Specifications. Technical report, Naval Research Laboratory (1996)
7. Girardi, M.R.: Classification and Retrieval of Software through their Description in Natural Language. PhD thesis, Computer Science Department University of Geneva, Switzerland (1996)
8. Jurafsky, D., Martin, J.: Speech and Language Processing. Prentice-Hall (2000)
9. Bryant, B.R., Lee, B.S.: Two-Level Grammar as an Object-Oriented Requirements Specification Language. Proc. 35th Hawaii Int. Conf. System Sciences (2002)
10. Bjørner, D., Jones, C.B.: The Vienna Development Method: The Meta-Language. Springer-Verlag (1978)
11. Quatrani, T.: Visual Modeling with Rational Rose 2000 and UML. Addison-Wesley (2000)
12. IFAD: The VDM++ Toolbox User Manual. Technical report, IFAD (`www.ifad.dk`) (2000)
13. Walter Reed Army Institute for Research (WRAIR): CARA Specification: Roprietary Document. Technical report, WRAIR, Dept. of Resuscitative Medicine (2001)
14. Decker, S., Melnik, S., van Harmelen, F., Fensel, D., Klein, M.C.A., Broekstra, J., Erdmann, M., Horrocks, I.: The semantic web: The roles of XML and RDF. IEEE Internet Computing **4** (2000) 63–74
15. Lee, B.S., Bryant, B.R.: Contextual Natural Language Processing and DAML for Understanding Software Requirements Specifications. Proc. 19th International Conference on Computational Linguistics (2002) 516–522

16. Cleaveland, J.C.: Program Generators with XML and Java. Prentice-Hall (2001)
17. Lakemeyer, G., Nebel, B.: Foundations of knowledge representation and reasoning. Volume 810. Springer-Verlag Inc. (1994)
18. Grady, W.: Moby Part-of-Speech II (data file) (1994)
19. Lee, B.S., Bryant, B.R.: Contextual Knowledge Representation for Requirements Documents in Natural Language. Proc. 15th International FLAIRS Conference (2002) 370–374
20. Bryant, B.R.: Object-Oriented Natural Language Requirements Specification. Proc. ACSC 2000, 23rd Australasian Comp. Sci. Conf. (2000) 24–30

A General Resource Framework for Real-Time Systems*

Insup Lee[1], Anna Philippou[2], and Oleg Sokolsky[1]

[1] Department of Computer and Information Science
University of Pennsylvania, USA
{lee,sokolsky}@cis.upenn.edu
[2] Department of Computer Science, University of Cyprus, Cyprus
annap@ucy.ac.cy

Abstract. The paper describes a formal framework for designing and reasoning about resource-constrained systems. The framework is based on a series of process algebraic formalisms which have been previously developed to describe and analyze various aspects of real-time communicating, concurrent systems. We develop a uniform framework for formal treatment of resources and demonstrate how previous work fits into the new framework.

1 Introduction

An embedded system consists of a collection of components that interact with each other and with their environment through sensors and actuators. Embedded software is used to control these sensors and actuators and to provide application-dependent functionality. Two important distinguishing characteristics of embedded applications are limited resources (processing power, memory, network bandwidth, power consumption, *etc.*) and the hybrid (discrete and continuous) nature of behaviors. Many embedded systems are part of safety-critical applications, *e.g.*, avionic systems, manufacturing, automotive controllers, and medical devices.

There are two major factors that complicate the design and implementation of embedded systems. First, the software complexity of embedded systems has been increasing steadily as microprocessors become more powerful. To mitigate the development cost of software, embedded systems are being designed to flexibly adapt to different environments. The requirements for increased functionality and adaptability make the development of embedded software complex and error-prone. Second, embedded systems are increasingly networked to improve functionality, reliability and maintainability. Networking makes embedded software even more difficult to develop, since composition and abstraction principles are poorly understood.

* This research was supported in part by NSF CCR-9988409, NSF CCR-0086147, NSF CCR-0209024, ARO DAAD19-01-1-0473, and by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

M. Wirsing et al. (Eds.): RISSEF 2002, LNCS 2941, pp. 234–248, 2004.

A natural response to the increasing complexity of embedded systems development is an increased emphasis on model-based development of embedded software. Models can be constructed for embedded systems and their properties can be analyzed through simulation and model checking. Models can be used to generate code skeleton and task structures and then platform dependent code can be added to work on specific environment. Since it may not be possible to completely automate code generation, models can be used to validate implementation. One way to do this is to use a design model for generating test suites, and then, use them to check the conformance of an implementation to design specifications. Another way is to extract models from legacy code and use them to validate the code with respect to specifications. The third way is to ensure that the implementation is correct at runtime through monitoring and checking of the behavior of the running system. For safety critical embedded systems, it is important to have assurance that such systems are reliable. It is well-known that activities related to certification of such systems (e.g., avionics, medical devices) are extremely time consuming and costly. Model based development can be tailored to facilitate certification processes adopted by various regulatory agencies such as FAA and FDA.

Figure 1 provides an overview of the model-based development framework being developed at the University of Pennsylvania. From the model of an embedded system specified as hybrid system, the code generator produces a set of tasks as well as code for the tasks. Given the end-to-end timing requirements of the embedded system and the description of the target hardware and operating systems platform, the timing estimator identifies the periods and deadlines of the tasks. These timing parameters are chosen to guarantee the end-to-end constraints, but the execution times of the tasks are not yet determined. The resource modeler takes the communication and synchronization structure of the generated tasks and tradeoffs between code size and execution as input and generates possible resource-aware models. From the models, the schedulability evaluator estimates the worst-case execution times and then identifies which models can be executed with their timing parameters under the available resource limits. If no such solution exists, the timing parameters of the tasks are readjusted and the design process is repeated.

In this paper, we limit our discussion to the general resource framework for embedded systems, which provides a formal semantical foundation for understanding resource-constrained behaviors subject to real-time constraints, memory limitations, power consumption, etc. The notion of a resource plays a central role in the specification and design of embedded systems, where execution is subject to a large number of resource constraints, such as timing, power consumption, size and weight, etc. We feel that, in order to properly specify and analyze such systems, a modeling formalism should incorporate the notion of a resource as a first-class entity.

Related work in the area of resource handling in embedded real-time systems falls into two categories. On the one hand, the importance of the issue has been long realized by practitioners and a number of model-based, albeit informal, ap-

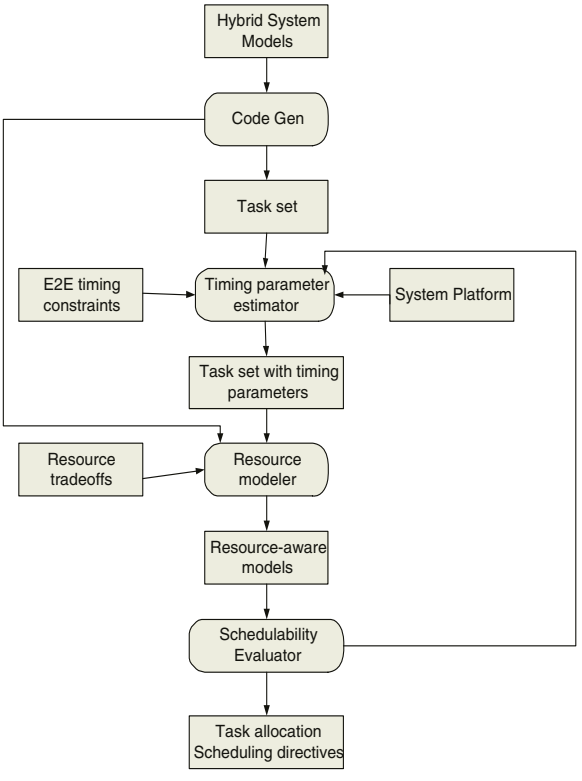


Fig. 1. Overall structure of model-based development framework.

proaches have been published. We mention [13,16,9,3,2] among many others. On the other hand, several formal approaches have emerged that aim at scheduling of sets of tasks under constraints. For the most part, these approaches consider only timing constraints and do not introduce the notion of resource, implicitly considering the processor as the only shared resource in the system. For example, the authors of [8] propose a formalism that allows us to model preemption in asynchronous real-time systems. In [4], the authors limit themselves to fixed-priority scheduling approaches, which allow them not to consider preemption directly, accounting for it in the worst-case computation time. The formalism of [6] provides a general scheme for handling preemption of processes due to resource contention, but the scheduling rules have to be encoded by modifying transition rules in the formalism (effectively creating a custom formalism for each scheduling policy). A different approach is taken in [1], where the authors view the scheduling activity as control and use controller synthesis techniques to model scheduled real-time systems.

In our previous work, we have proposed a family of process-algebraic formalisms for modeling and reasoning about resource-constrained systems (see [11] for an overview). The family is built around ACSR, a discrete-time process

algebra. Extensions and variations include Dense-time ACSR that includes a more general notion of time; ACSR-VP that includes a value-passing capability; PACSR, a probabilistic formalism for quantitative reasoning about fault-tolerant properties of resource-bound systems; P^2ACSR [12] that allows to specify power-constrained systems. The PARAGON toolset [18] provides tool support for modeling and analysis using these formalisms.

The family of process algebras all share the modeling approach, where a system is modeled as a collection of communicating concurrent processes. Operators to express structure of a process are also very similar in all of the formalisms. The difference lies in the way resources are used and the attributes they carry. For example, in PACSR resources have an attribute that captures the probability of failure for the resources. In ACSR-VP we can have tuples of data values associated with a resource that may be manipulated during execution, and P^2ACSR introduces power consumption attributes. In each formalism, the set of resource attributes and the way the attributes are manipulated are slightly different. This makes it difficult to systematically present the formalisms. More importantly, each extension required changes to the PARAGON toolset that have to be implemented in an *ad hoc* every time a new extension is made.

The main aim of this paper is the development of a general process algebraic framework to facilitate the construction of system models that allow us to capture faithfully all of their relevant functional and non-functional requirements. The framework allows to represent all the formalisms in the ACSR family and provide for easy incorporation of new features both into the formalism and the supporting toolset in a uniform manner. The paper presents the formal definition of the framework and demonstrate how to capture existing formalisms within the framework as well as create new ones.

2 The Framework

We define a process-algebraic formal framework for reasoning about real-time systems. The basic entity of the framework is that of a resource. We assume that a system contains a finite set of reusable resources drawn from a countably infinite set of resources $\mathcal{R}$. Resources can correspond to physical entities, such as processor units and communication channels, or to abstract notions such as message arrival.

A resource is characterized by a set of attributes that let us capture aspects of the resource's behavior depending on the needs of the application, such as timing, probabilistic, or communication behavior, or, priority and power consumption during resource usage. Resources are partitioned into classes $\mathcal{R}_1, \ldots,$ with all resources in a class having the same attributes. In turn, an attribute may have one or more elements; an attribute, a, with n elements is specified as a tuple

$$a : \langle T_1 : \mathsf{kind}_1, \ldots, T_n : \mathsf{kind}_n \rangle ,$$

where $T_1, \ldots, T_n$ are basic types such as integers, characters, tuples, etc, and $\mathsf{kind}_1, \ldots, \mathsf{kind}_n \in \{\mathsf{static}, \mathsf{dynamic}\}$ define whether the value of the attribute's

element remains constant throughout computation (static) or is associated to every resource use (dynamic).

Example 1. As an example, consider the class of resources $\mathcal{R}_f$ that may experience failures, consume power, and whose use is regulated by priorities. We may characterize this class by three attributes as follows:

$$\mathcal{R}_f : [\pi : \langle [0,1] : \mathsf{static} \rangle, pc : \langle int : \mathsf{dynamic} \rangle, pr : \langle int : \mathsf{dynamic} \rangle].$$

Attribute π captures the possibility of resource failure. It has one element, that of the probability of failure, which is assumed to be constant throughout all executions of the resource. Attribute pc is the power consumption, which may be different in each resource use depending on the level of power required on each occasion, and pr is the priority of a resource access, which may also be different depending on which process uses r.

When writing a model in the framework, given a resource r, we specify the values of all of the static elements of its attributes, and then, whenever r is used in the model, it is accompanied by values for all of the dynamic elements of its attributes. Furthermore, we write $a_r(i)$ for the ith element of attribute a of resource r. Given resources *cpu* and *chan* in the resource class $\mathcal{R}_f$, we specify once initially that, for example, $\pi_r(cpu) = 0.01$ and $\pi_{chan}(1) = 0.1$. Then, whenever each of the resources is used in the model, we give the values of its dynamic attributes in a *resource access*, for example, $(cpu, 2.5, 1)$. We will always assume positional correspondence between the dynamic attribute names in the attribute tuple of a resource class and the values of attributes in the resource use. Therefore, in $(cpu, 2.5, 1)$, $pc = 2.5$ and $pr = 1$. Resource accesses are specified in *actions*. An action A is a collection of resource accesses. By $\rho(A)$ we denote the multiset of names of resources used in A. Actions are building blocks for processes.

Syntax. We let P, Q range over processes, A ranges over actions, and I ranges over sets of resources. The following grammar describes the syntax of processes and actions.

$$\begin{aligned} P &::= \text{ NIL} \mid A : P \mid P + Q \mid P \| Q \mid [P]_I \mid P \backslash\backslash I \\ A &::= \{(r_1, a_{11}, \ldots, a_{1n_1}), \ldots, (r_m, a_{m1}, \ldots, a_{mn_m})\} \end{aligned}$$

Process NIL represents the inactive process. Process $A : P$, is the prefix operator: it executes action A and proceeds to P. Process $P+Q$ represents a nondeterministic choice between the two summands. Process $P \| Q$ describes the concurrent composition of P and Q: the component processes may proceed independently or interact with one another while executing actions. The construct $[P]_I$, $I \subseteq \mathcal{R}$, referred to as *resource closure*, produces a process that reserves the use of resources in I for itself, extending every action A in P with resources in $I - \rho(A)$. Finally, $P \backslash\backslash I$, referred to as *resource hiding*, allows the process to *hide* or *restrict* the identity of resources in I so that they are not visible on the interface with the environment of process P.

Example 2. As an example of a process, consider

$$P \stackrel{\text{def}}{=} \{(cpu, 3, 2), (chan, 1, 0)\} : P_1 + \{(cpu, 1, 1)\} : P_2 \ .$$

where resources *cpu* and *chan* are drawn from the resource class $\mathcal{R}$ of Example 1. Process P represents a processor that can accept messages from a channel. We assume that reading the message from the channel requires additional power than remaining idle. Depending on whether the message arrives or not, P has two alternative behaviors. If the message arrives, as described in resource access $(chan, 1, 0)$, the processor may receive the message, consuming 3 units of power, and proceed to process it as P_1. Otherwise, the processor consumes only 1 unit of power and continues as P_2.

Semantics. The semantics of the framework is given operationally by a transition system that captures the behavior of processes. It is based on the notion of a configuration which comprises of a process and a world/state that can be used to keep useful information about the resources of the process. We write $\mathcal{C}$ for the set of all configurations and we write $S(P) \in \mathcal{C}$, for a configuration containing process P in state S. Finally, we write Act for the set of all actions a configuration can engage in. The semantics is based on a function

$$F(\mathcal{C}, Act) \longrightarrow 2^{\mathcal{C}}$$

which, given a process configuration $S(P)$ and an action A, returns the set of configurations that can be reached from $S(P)$ by performing action A. Thus, the semantics is based on the following rule:

$$\frac{S'(P') \in F(S(P), A)}{S(P) \stackrel{A}{\longrightarrow} S'(P')}$$

Domain Specialization. In order to adjust the general framework for the needs of a specific application domain, we must give meaning for resources and their attributes and establish the semantics for the processes. The following steps are needed to perform the specialization for a particular domain.

- *Resource classes.* A finite set of resource classes need to be established for the domain along with the attributes of the class.
- *Syntactic consistency.* A predicate *valid* must be provided. For a given action A, $valid(A)$ denotes that the action can be legitimately used in a model within this domain. Furthermore, we may restrict the domain for the set of resources I appearing in process constructs $[P]_I$ and $P \backslash\backslash I$.
- *Semantic interpretation.* Finally, the set $\mathcal{C}$ of configurations has to be defined and the function F has to be given.

3 Framework Instantiations

In this section, we will show how to instantiate the general framework to several progressively more complex domains.

3.1 CCS

The first domain we consider is the CCS domain [14]. CCS processes consider only communication constraints between concurrent processes. The actions that processes can engage in are *send* or *receive* messages on named channels. In addition, there is a *silent* action denoted τ. A send action and a receive action on the same channel can synchronize to produce a silent action, whereas the silent action cannot synchronize with any other action. We introduce two resource classes, $\mathcal{R}_1$ and $\mathcal{R}_2$. The class $\mathcal{R}_1$ has one attribute *polarity* of type $\langle\{!, ?\} : \mathsf{dynamic}\rangle$. The class $\mathcal{R}_2$ does not have attributes and contains the single resource τ. As a shorthand, and to coincide with CCS style, we write $r!$ for $\{(r, !)\}$, the send action on resource (channel r), $r?$ for $\{(r, ?)\}$ the receive action on channel r and τ for $\{(\tau)\}$ the silent action.

The consistency predicate for an action A stipulates that A is valid if and only if $\rho(A)$ is a singleton. Finally, we require that in $[P]_I$, I ranges from the empty set of resources, i.e. the process construct is disabled, whereas in $P\backslash\backslash I$, I can be any subset of the resource class $\mathcal{R}_1$, that is, we can only hide named channels.

The process does not need any additional state information ($S(P) = P$), and the semantic function is defined recursively on the process structure, following the standard CCS approach. We begin by considering how actions interact with the process constructs. Let A and B be well-formed actions and $I \subseteq \mathcal{R}_1$, then we define:

$$A \| B = \begin{cases} \tau, & \text{if } \{A, B\} = \{a?, a!\} \\ \bot, & \text{otherwise} \end{cases}$$

$$A \backslash\backslash I = \begin{cases} A, & \text{if } \rho(A) \notin I \\ \bot, & \text{otherwise} \end{cases}$$

Consequently we have that two actions may be composed in parallel to produce the silent action if they are send and receive actions on the same channel, and that an action can survive the hiding operator only if does not involve a resource from set I.

We may now define the semantic function F as follows, where $+$ stands for summation $\mathsf{mod}\,2$.

$$F(A{:}P, A) = \{P\}$$

$$R \in F(P_1 + P_2, A) \quad \textit{iff} \quad \exists i \in \{1, 2\} \text{ such that } R \in F(P_i, A)$$

$$R \in F(P_1 \| P_2, A) \quad \textit{iff} \quad (P_1 \xrightarrow{A_1} P_1',\ P_2 \xrightarrow{A_2} P_2',\ A = A_1 \| A_2,\ R = P_1' \| P_2') \text{ or } (\exists i \in \{1, 2\} \text{ such that } P_i' \in F(P_i, A),\ R = P_i' \| P_{i+1})$$

$$R \in F(P \backslash\backslash I, A) \quad \textit{iff} \quad R' \in F(P, B),\ A = B \backslash\backslash I,\ R = R' \backslash\backslash I$$

3.2 ACSR

ACSR can be viewed as an extension of CCS with time-consuming steps and priorities. We add a new resource class $\mathcal{R}_3$ with the attribute *time* of type

$\langle int : \mathsf{static} \rangle$ and specify that for any resource $r \in \mathcal{R}_3$, $time_r = 1$. That is all time-consuming actions take one unit of time. In addition, all three classes have the attribute *priority* of type $\langle int : \mathsf{dynamic} \rangle$.

The consistency predicate states than an action A is well-formed if the resources occurring in A, $\rho(A)$, are pairwise distinct and, satisfy either $\rho(A) \subseteq \mathcal{R}_3$, in which case they are referred to as *timed actions*, or $\rho(A) = \{r\}$, $r \in \mathcal{R}_1 \cup \mathcal{R}_2$, in which they are referred to as *instantaneous events*. We write $\mathcal{D}_R$ for the set of timed actions and $\mathcal{D}_E$ for the set of instantaneous events. As before, we omit the set brackets from actions involving resources in $\mathcal{R}_1 \cup \mathcal{R}_2$, and simply write $(a!, p)$ and $(a?, p)$ for send and receive actions along channel a. Further, we specify that, in $[P]_I$, $I \in 2^{\mathcal{R}_3}$, and that, in $P \backslash\backslash I$, $I \in 2^{\mathcal{R}_1} \cup 2^{\mathcal{R}_3}$.

We now proceed to give the semantic function for ACSR. This is similar to the one in the CCS domain, except that it handles timed actions and, further, applies the preemption relation $\prec$, which specifies when two actions are comparable with respect to priorities. For example, $\emptyset \prec A$ for all $A \in \mathcal{D}_R$, that is, the idle action $\emptyset$ is preemptable by all other timed actions, and $(a, p) \prec (a, p')$, whenever $p < p'$. For the precise definition of $\prec$ we refer to [10].

There is no state information and the definition of the semantic function is similar to that of CCS. We begin by describing the composability of actions with the various operators. Let A and B be well-formed actions, $I \in 2^{\mathcal{R}_3}$, and $J \in 2^{\mathcal{R}_1} \cup 2^{\mathcal{R}_3}$, then

$$A \| B = \begin{cases} (\tau, p + p'), & \text{if } \{A, B\} = \{(a?, p), (a!, p')\} \\ A \cup B & \text{if } A, B \in \mathcal{D}_R,\ \rho(A) \cap \rho(B) = \emptyset \\ \bot, & \text{otherwise} \end{cases}$$

$$[A]_I = \begin{cases} A \cup \{(r, 0) \mid r \in I - \rho(A)\}, & \text{if } A \in \mathcal{D}_R \\ A, & \text{otherwise} \end{cases}$$

$$A \backslash\backslash J = \begin{cases} \{(r, p) \in A \mid r \notin J\}, & \text{if } A \in \mathcal{D}_R \\ A, & \text{if } A \in \mathcal{D}_E,\ \rho(A) \notin J \\ \bot, & \text{otherwise} \end{cases}$$

We point out that two timed actions may be composed together only if the resources they access are independent from each other, that is, they do not compete for the use of any common resources. In case of the contrary, $A \| B = \bot$, signifying that a deadlock arises. $[A]_I$ and $A \backslash\backslash I$ capture the informal explanation of the process constructs $[P]_I$ and $P \backslash\backslash I$, respectively. The former ensures that access to resources I is reserved for process P by employing all of the resources $r \in I - \rho(A)$ at priority level 0. As a result any further sharing of the resources in I, with any parallel process of P, is prohibited (see the definition of $A \| B$). The latter disables any instantaneous events involving a channel in I and hides the use of I-resources from timed actions.

We proceed to define function F. We let A, B range over the set of actions Act of ACSR, consisting of timed actions and instantaneous events.

$$\begin{array}{lll} F(A{:}P, A) = \{P\} & & \\ R \in F(P_1 + P_2, A) & \textit{iff} & \exists i \in \{1, 2\} \text{ such that } R \in F(P_i, A) \\ & & \text{and, if } P_{i+1} \xrightarrow{B}, A \not\prec B \end{array}$$

$$
\begin{array}{lll}
R \in F(P_1 \| P_2, A) & \textit{iff} & [(P_1' \in F(P_1, A_1),\ P_2' \in F(P_2, A_2), \\
 & & A = A_1 \| A_2,\ R = P_1' \| P_2') \\
 & & \text{or } (A \in \mathcal{D}_E, \text{ and } \exists i \in \{1,2\} \cdot \\
 & & P_i' \in F(P_i, A) \text{ and } R = P_i' \| P_{i+1})] \\
 & & \text{and, if } \exists i \in \{1,2\} \cdot P_i \xrightarrow{B}, B \in \mathcal{D}_E \\
 & & \text{or } P_1 \xrightarrow{B_1}, P_2 \xrightarrow{B_2}, B = B_1 \| B_2, \\
 & & \text{then } A \not\prec B \\
R \in F([P]_I, A) & \textit{iff} & R' \in F(P, A'),\ R = [R']_I,\ A = [A']_I \\
 & & \text{and, if } P \xrightarrow{B} \text{ then } A \not\prec [B]_I \\
R \in F(P \backslash\backslash I, A) & \textit{iff} & R' \in F(P, A'),\ R = R' \backslash\backslash I,\ A = A' \backslash\backslash I, \\
 & & \text{and, if } P \xrightarrow{B} \text{ then } A \not\prec B \backslash\backslash I
\end{array}
$$

The first two rules define the semantics of the prefix and summation operators. The third rule describes the behavior of the parallel composition operator. This allows component processes to proceed independently or synchronize with one another with respect to instantaneous actions and forces processes to synchronize on timed actions, making timed transitions truly synchronous, in that a process only advances if both of its subprocesses take a step. By the definition of $A_1 \| A_2$ we have that only one process may use a resource during any time step. The next rule describes the behavior of the close operator: When a process is embedded in a closed context, such as $[P]_I$, we ensure that there is no further sharing of the resources $r \in I - \rho(A)$ by employing all of these resources at priority level 0 (see definition of $[A]_I$). Instantaneous events are not affected by the close operator. Finally, the rule for resource hiding establishes that the set of resources I is restricted from the interface with the environment. Note, that in all but the first rule, a side condition checks that the action in question cannot be preempted by any other enabled action of the process.

3.3 PACSR

The PACSR domain is aimed at fault-tolerance analysis of real-time systems. Resources are allowed to fail with a fixed probability during an execution. This is captured by extending $\mathcal{R}_3$ with an additional attribute π of type $\langle [0,1] : \mathsf{static} \rangle$, representing a probability. This probability captures the rate at which the resource may fail. To be able to reason about failed as well as non-failed resources, we also have the attribute *status* of type $\langle \{up, down\} : \mathsf{dynamic} \rangle$. Intuitively, the process $\{(r, 1, up)\} : P$ will succeed in performing action $\{(r, 1, up)\}$ with probability π_r and fail, becoming NIL with probability $1 - \pi_r$. On the other hand, the process $\{(r, 1, down)\} : P$ will fail with probability π_r, exactly when the first process succeeds, and succeeds with probability $1 - \pi_r$. The use of failed resources is useful when we need to specify recovery from failures. We adopt the following notation: for all $r \in \mathcal{R}_3$, we write (r, p), for $(r, p, up))$, and $(\overline{r}, p)$, for $(r, p, down)$.

The consistency condition for the PACSR domain is the same as for the ACSR domain, both in case of the validity predicate and in the case of the syntactic conditions for resource hiding and resource closure.

We continue to define the semantics function F for PACSR. As already mentioned, resources are associated with a probability of failure. Thus, the behavior of a system has certain probabilistic aspects to it which must be reflected in the operational semantics of the domain. For example consider action $\{(cpu, 2), (chan, 1)\}$, where resources *cpu* and *chan* have probabilities of failure 0 and 1/3, respectively, that is $\pi_{cpu} = 1$ and $\pi_{chan} = 2/3$. Then the action takes place with probability $\pi_{cpu} \cdot \pi_{chan} = 2/3$ if both resources are up, and fails with probability 1/3 if either of the resources fails. Therefore, behavior of a given process P depends on the status of resources which are relevant to P. To capture information about resource status in the configuration we write $S(P)$ for process P in state S, where $S \in 2^{\mathcal{R}_3} \times \{up, down\}$ records the status of resources of P. Configurations are partitioned into probabilistic configurations, from which only probabilistic steps are possible that update resource failure information, and non-deterministic configurations, where the state of all relevant resources is known and transitions can be computed.

The intuition for the semantics is as follows: for a process P, we begin with the configuration $\emptyset(P)$. As computation proceeds, probabilistic transitions are performed from configurations to determine the status of probabilistic resources immediately relevant for execution (denoted $\mathsf{imr}(P)$) but for which there is no knowledge in the configuration's world. Once the status of a resource is determined by some probabilistic transition, it cannot change until the next timed action occurs. Once a timed action occurs, the state of resources has to be determined anew, since in each time unit resources can fail independently from any previous failures. Nondeterministic transitions (which can involve events or actions) are performed from nondeterministic configurations. Precise definition for $\mathsf{imr}(P)$ is given in [15].

Let $S = \{(r_1, s_1), \ldots, (r_n, s_n)\} \subseteq \mathcal{R}_3 \times \{(up, down)\}$ and $I = \{r_1, \ldots, r_n\} \subseteq \mathcal{R}_3$. We write

- $\mathsf{p}(S) = \Pi_{1 \leq i \leq n, s_i = up}\ \pi_{r_i} \ \cdot\ \Pi_{1 \leq i \leq n, s_i = down}\ (1 - \pi_{r_i})$,
- $\mathcal{W}(I) = \{\{r_1, s_1), \ldots, (r_n, s_n)\} \mid s_i \in \{up, down\}\}$, and
- $res(S) = I$.

We partition the set of configurations into the sets of nondeterministic configurations, $\mathcal{C}_N$, and probabilistic configurations, $\mathcal{C}_P$. We have that $S(P) \in \mathcal{C}_N$ *iff* $\mathsf{imr}(P) - res(S) = \emptyset$, that is, there is no immediate resource of P whose status is not already recorded in S, and $S(P) \in \mathcal{C}_P$, otherwise. We proceed to define function F.

$S'(P) \in F(S(P), \ell)$	*iff*	$P \in \mathcal{C}_P$, $S'' \in \mathcal{W}(\mathsf{imr}(P) - res(S))$, $S' = S \cup S''$, and $\ell = \mathsf{p}(S'')$
$F(S(A{:}P), A) = \{S(P)\}$	*iff*	$S(A{:}P) \in \mathcal{C}_N$ and $A \in \mathcal{D}_E$
$F(S(A{:}P), A) = \{\emptyset(P)\}$	*iff*	$S(A{:}P) \in \mathcal{C}_N$, $A \subseteq S$ and $A \in \mathcal{D}_R$
$R \in F(S(P_1 + P_2), A)$	*iff*	$S(P_1 + P_2) \in \mathcal{C}_N$, $\exists i \in \{1, 2\}$ such that $R \in F(S(P_i), A)$ and if $S(P_{i+1}) \xrightarrow{B}$, $A \not\prec B$
$R \in F(S(P_1 \| P_2), A)$	*iff*	$S(P_1 \| P_2) \in \mathcal{C}_N$, and [$S'(P_1') \in F(S(P_1), A_1)$, $S'(P_2') \in F(S(P_2), A_2)$, $A = A_1 \| A_2$, $R = S'(P_1' \| P_2')$ or $A \in \mathcal{D}_E$, and

$$\exists i \in \{1,2\}\cdot\ S'(P_i') \in F(S(P_i), A),\ R = S'(P_i' \| P_{i+1})\ ]$$
$$\text{and, if } \exists i \in \{1,2\} \cdot S(P_i) \xrightarrow{B}, B \in \mathcal{D}_E$$
$$\text{or } S(P_1) \xrightarrow{B_1}, S(P_2) \xrightarrow{B_2}, B = B_1 \| B_2, \text{ then } A \not\prec B$$

$R \in F(S([P]_I), A)$ *iff* $S([P]_I) \in \mathcal{C}_N$, $S'(Q) \in F(S(P), A')$, $A = [A]_I$, $R = S'([Q]_I)$, and, if $S(P) \xrightarrow{B}$ then $A \not\prec [B]_I$

$R \in F(S(P \backslash\backslash I), A)$ *iff* $S(P \backslash\backslash I) \in \mathcal{C}_N$, $S'(Q) \in F(S(P), A')$, $A = A' \backslash\backslash I$, $R = S'(Q \backslash\backslash I)$, and, if $S(P) \xrightarrow{B}$ then $A \not\prec B \backslash\backslash I$

Thus, given a probabilistic configuration $S(P)$, with I the immediate resources of P for which the state is not yet determined in S, and $S'' \in \mathcal{W}(I)$, P enters the state extended by S'' with probability $\mathsf{p}(S'')$. Note that configuration $S(P)$ evolves into $S'(P)$ which is, by definition, a nondeterministic configuration.

Example 3. To illustrate the probabilistic transition relation, consider process

$$P \stackrel{\text{def}}{=} \{(r_1, 1), (r_2, 2)\} : P_1 + (e, 1).P_2$$

in the initial configuration $\emptyset(P)$. The immediate resources of P are $\{r_1, r_2\}$. Since there is no knowledge in the configuration's world regarding these resources, the configuration belongs to the set of probabilistic configurations $\mathcal{C}_p$, from where we have four probabilistic transitions that determine the states of r_1 and r_2:

$$\emptyset(P) \xrightarrow{\pi_{r_1} \cdot \pi_{r_2}} \{r_1, r_2\}(P), \qquad \emptyset(P) \xrightarrow{\pi_{r_1} \cdot (1-\pi_{r_2})} \{r_1, \overline{r_2}\}(P),$$
$$\emptyset(P) \xrightarrow{(1-\pi_{r_1}) \cdot \pi_{r_2}} \{\overline{r_1}, r_2\}(P), \quad \text{and} \quad \emptyset(P) \xrightarrow{(1-\pi_{r_1}) \cdot (1-\pi_{r_2})} \{\overline{r_1}, \overline{r_2}\}(P).$$

All of the resulting configurations are nondeterministic since they contain full information about P's immediate resources.

The remaining of the rules concerning the nondeterministic configurations follow along the same lines as the ACSR rules. The only point to note concerns the prefix operator: for the timed action A to be performed by configuration $S(A : P)$, it must be that all resources in A are available in the configuration's state.

Example 4. Returning to the previous example, the nondeterministic configuration $\{r_1, r_2\}(P)$, where $P \stackrel{\text{def}}{=} \{(r_1, 2), (r_2, 2)\} : P_1 + (e, 1).P_2$ has two nondeterministic transitions:

$$\{r_1, r_2\}(P) \xrightarrow{\{(r_1,2),(r_2,2)\}} \emptyset(P_1) \qquad \text{and} \qquad \{r_1, r_2\}(P) \xrightarrow{(e,1)} \{r_1, r_2\}(P_2).$$

The other configurations $\{r_1, \overline{r_2}\}(P)$, $\{\overline{r_1}, r_2\}(P)$, and $\{\overline{r_1}, \overline{r_2}\}(P)$, allow only the e-labeled transition since either r_1 or r_2 is failed.

4 Multi-capacity Resources and Memory Constraints

In this section we use the resource framework to construct a formalism MCSR that captures memory use as a different kind of resource. Memory is a critical

resource in size-constrained embedded systems such as mobile phones. In the design of an embedded system, we need to consider tradeoffs between memory use and the speed of the tasks in the system. A task can be made to use less memory at the cost of executing longer. But this increased execution can violate timing constraints in the system. The proposed formalism will allow us to capture such tradeoffs and reason about their effects.

We see that the nature of memory as a resource is different from other serially reusable resources considered so far in this paper. Two processes can use the same memory, as long as the total use does not exceed the memory capacity. Therefore, we introduce the new class of resources called *multi-capacity* resources.

We develop MCSR as an extension of ACSR (see Section 3.2) by adding a new resource class, $\mathcal{R}_4$, of multi-capacity resources. We will use resources in this class to represent memories, however, other kinds of resources may be modeled in the same way. Each resource has two attributes. The first attribute, *capacity* of type $\langle int : static \rangle$, represents the capacity of the resource. The second attribute, *use* of type $\langle int : dynamic \rangle$, captures the memory used in a step of a process.

The consistency predicate in this framework, extends that of ACSR by allowing multi-capacity resources to be used in timed actions of MCSR processes, so that for a timed action A, $\rho(A) \subseteq \mathcal{R}_3 \cup \mathcal{R}_4$. The semantic function for MCSR is the same as for ACSR, except that the definition of action composition is modified as follows:

$$A \| B = \begin{cases} (\tau, p + p'), & \text{if } \{A, B\} = \{(a?, p), (a!, p')\} \\ A \uplus B, & \text{if } A, B \in \mathcal{D}_R,\ (\rho(A) \cap \rho(B)) \cap \mathcal{R}_3 = \emptyset \text{ and} \\ & \forall r \in \mathcal{R}_4, (r, u_1, c) \in A, (r, u_2, c) \in B \Rightarrow u_1 + u_2 \leq c \\ \bot, & \text{otherwise} \end{cases}$$

The operator $A \uplus B$, defined on compatible timed actions (meaning that $A \| B \neq \bot$), is defined as follows:

1. $(r, a_1, \ldots, a_n) \in A \uplus B$ if $r \in \mathcal{R}_3 \cup \mathcal{R}_4, (r, a_1, \ldots, a_n) \in A$ and $r \notin \rho(B)$ or $(r, a_1, \ldots, a_n) \in B$ and $r \notin \rho(A)$, or
2. $(r, u_1 + u_2, c) \in A \uplus B$ if $r \in \mathcal{R}_4, (r, u_1, c) \in A$ and $(r, u_2, c) \in B$.

Memory Aware Scheduling. We illustrate the use of MCSR by showing a collection of periodic tasks that execute within the same system, sharing the processor and memory. Each task T_i is characterized by its period p_i and execution time c_i. Each task is released for execution at the beginning of every period and its deadline is equal to the period. That is, a task has to use the processor for c_i time units in each interval $[k * p_i, (k + 1) * p_i]$, for each integer k. Tasks are assigned a priority for accessing the processor, and an executing task can be preempted by a task with a higher priority. Each task has a fixed amount $m_{i,c}$ of memory allocated to store the code and static data structures. In addition, when the task is released for execution, an additional amount of memory, $m_{i,d}$, is allocated to the task for keeping its dynamic data. Once the task completes its execution in the current period, the dynamically allocated memory is released.

To model such a task in MCSR, we adapt the approach of [7], extending it with additional resources representing memory consumption. To simplify presentation, we assume that priorities of tasks are fixed, for example, according to rate-monotonic scheduling discipline. More complex dynamic-priority scheduling approaches, such as EDF, can be accommodated as well. An instance of the scheduling problem is modeled as a collection of processes $T_1, \ldots, T_n$. Process T_i is shown below. A task is represented as a parallel composition of two processes: Job_i and $Activator_i$.

$$
\begin{array}{ll}
T_i & = (Job_i \| Activator_i) \backslash\backslash \{start_i\} \\
Activator_i & = (\overline{start_i}, i).\emptyset^{p_i} : Activator_i \\
Job_i & = \{(mem, m_{i,c})\} : Job_i + (start_i, 0).Exec_{i,0,0} \\
Exec_{i,e,t} & = e < c_i \rightarrow \{(cpu, i), (mem, m_{i,c} + m_{i,d})\} : Exec_{i,e+1,t+1} \\
 & \quad + \{(mem, m_{i,c} + m_{i,d})\} : Exec_{i,e,t+1} \\
 & \quad + e = c_i \rightarrow Job_i \qquad\qquad e \in \{0..c_i\}, t \in \{0..p_i\}
\end{array}
$$

The description of task T_i involves the use of three resources: $start_i \in \mathcal{R}_1$, $mem \in \mathcal{R}_4$ and $cpu \in \mathcal{R}_3$, the last two corresponding to the system's processor and available memory. We assume a value M for the capacity of multi-capacity resource mem. The task consists of two processes running in parallel. The role of the activator is to keep track of the timing constraint of the task. At the beginning of every period, $Activator_i$ sends the signal $start_i$ to Job_i, releasing the task for execution, and then idles until the end of the period. If, by the end of the period, the task has not finished its execution, it will not be able accept the next $start_i$ signal, resulting in a deadlock alerting us to the scheduling failure.

The other process, Job_i, upon receiving the $start_i$ signal, Job_i begins its execution. At each time step, the task has a fixed priority that is equal to the task index. When the task receives the processor resource, it executes for one time unit and its accumulated execution time e is increased together with the elapsed time t. At any time step, the task can be interrupted by another task that has a higher priority. In this case, the interrupted task executes a timed action that does not use the *cpu* resource, but retains its memory use. In this case, its accumulated execution time stays the same while the elapsed time is increased.

Note that each job uses some amount of the memory resource *mem* in each step. However, this amount is different in different states of the task. While the job is waiting for the $start_i$ signal in Job_i, it uses $m_{i,c}$ units of memory, and after the task is started, it uses $m_{i,c} + m_{i,d}$ units of memory, regardless of whether it is running or preempted.

Code Size vs. Execution Time Tradeoff Modeling. In a recent paper [17], Shin *et al.* study a different problem that considers a tradeoff between code size (static memory use) and execution time. By choosing different encodings of the instructions sets, modern embedded processors offer the possibility to reduce the code size at the expense of a longer execution time. Each task can have its own encoding. However, increased execution time may render the task set

unschedulable, while increased code size may exceed the memory capacity. Thus we have to find an encoding for each task that will satisfy all the constraints. For a task i, the tradeoff is represented as a list of pairs $(m_{i,j}, c_{i,j})$, where the first element is the code size and the second element is the respective execution time. The encoding is chosen statically before the task begins executing.

We can address this problem with a modified task model shown below. Each task initially makes a non-deterministic choice from J possibilities, choosing its memory use $m_{i,j}$ and execution time $c_{i,j}$. Since the memory use remains constant once the choice has been made, only the first step of the process $JobStart_i$ uses the memory resource. The rest of the job process consists of J disjoint copies of the process Job_i from the previous example, for the different values of $c_{i,j}$. When the task processes are combined in parallel to represent the complete system, some of the initial choices become infeasible, exceeding the memory constraints. Of the initial choices that satisfy the memory constraints, some will violate the timing constraints during execution, resulting in a deadlock in the state space. Therefore, analysis will find initial steps that lead to deadlock-free subsystems to identify parameter values that satisfy both timing and memory constraints.

$$
\begin{array}{lll}
T_i & = (JobStart_i \| Activator_i) \backslash \{start_i\} & \\
Activator_i & = (\overline{start_i}, i).\emptyset^{p_i} : Activator_i & \\
JobStart_i & = \Sigma_{j \in J}\{(mem, m_{i,j})\} : Job_{i,j} & \\
Job_{i,j} & = \emptyset : Job_{i,j} + (start_i, 0).Exec_{i,j,0,0} & \\
Exec_{i,j,e,t} & = e < c_{i,j} \rightarrow \{(cpu, i)\} : Exec_{i,j,e+1,t+1} & \\
 & \quad + \emptyset : Exec_{i,j,e,t+1} & \\
 & + e = c_{i,j} \rightarrow Job_{i,j} & e \in \{0..c_{i,j}\}, t \in \{0..p_i\}
\end{array}
$$

5 Conclusions and Future Work

We have presented a formal approach to the design of real-time embedded systems, which explicitly captures resource constraints that affect the system behavior. The approach includes an extension mechanism that allows us to easily incorporate new kinds of resources and resource constraints. We have shown how to model memory capacity constraints using multi-capacity resources. The resource-modeling formalism is incorporated into a larger model-based development framework for embedded systems.

We are working to identify additional classes of resources and develop means of incorporating them into the formalism, as well as providing flexible tool support for the model development in the formalism. Two interesting extensions to the current work are being considered. One is to go beyond serially reusable resources to *consumable* resources, which can be used only a fixed amount of times during a computation and possibly replenished after a certain amount of time passes. Such an extension will allow us to have a more natural treatment of battery-operated devices. The other direction is to explore whether stochastic process algebras (e.g., [5]) can be captured within our framework.

References

1. K. Altisen, G. Goessler, and J. Sifakis. Scheduler modeling based on the controller synthesis paradigm. *Journal of Real-Time Systems*, 23:55–84, 2002.
2. H. Ammar, V. Cortellessa, and A. Ibrahim. Modeling resources in a UML-based simulative environment. In *ACS/IEEE International Conference on Computer Systems and Applications (AICCSA '01)*, June 2001.
3. L. Baum and T. Kramp. Towards a uniform modeling technique for resource-usage scenarios. In *Proceedings of PDPTA '99*, June-July 1999.
4. V.A. Braberman and M. Felder. Verification of real-time designs: combining scheduling theory with automatic formal verification. In *Proceedings of the 7th European Engineering Conference*, pages 494–510, 1999.
5. E. Brinksma, J-P. Katoen, D. Latella, and R. Langerak. A stochastic causality-based process algebra. *The Computer Journal*, 38(7):552–565, 1995.
6. M. Buchholtz, J. Andersen, and H.H. Loevengreen. Towards a process algebra for shared processors. In *Workshop on Models for Time-Critical Systems*, BRICS Notes Series NS-01-5, pages 87–99, August 2001.
7. J.-Y. Choi, I. Lee, and H.-L. Xie. The specification and schedulability analysis of real-time systems using ACSR. In *Real-Time Systems Symposium*. IEEE Computer Society Press, December 1995.
8. J. Ermont and F. Boniol. TPAP: an algebra of preemptive processes for verifying real-time systems with shared resources. In *Workshop on Theory and Practice of Timed Systems*, April 2002.
9. E. Huh, L. Welch, B. Shirazi, and C. Cavanaugh. Heterogeneous resource management for dynamic real-time systems. In *Heterogeneous Computing Workshop*, pages 287–296, May 2000.
10. I. Lee, P. Brémond-Grégoire, and R. Gerber. A process algebraic approach to the specification and analysis of resource-bound real-time systems. *Proceedings of the IEEE*, pages 158–171, Jan 1994.
11. I. Lee, J.-Y. Choi, H.-H. Kwak, A. Philippou, and O. Sokolsky. A family of resource-bound real-time process algebras. In *Formal Techniques for Networked and Distributed Systems (FORTE'01)*, August 2001.
12. I. Lee, A. Philippou, and O. Sokolsky. Process algebraic modelling and analysis of power-aware real-time systems. *Computing and Control Engineering Journal*, 13(4):180–188, August 2002.
13. A. Mehra, A. Indiresan, and K.G. Shin. Resource management for real-time communication: Making theory meet practice. In *Proceedings of IEEE Real-Time Technology and Applications Symposium*, pages 130–138, June 1996.
14. R. Milner. *Communication and Concurrency*. Prentice Hall Intl., 1989.
15. A. Philippou, O. Sokolsky, R. Cleaveland, I. Lee, and S. Smolka. Probabilistic resource failure in real-time process algebra. In *Proceedings of CONCUR '98*, pages 389–404. Springer-Verlag, 1998.
16. S. Saewong and R. Rajkumar. Cooperative scheduling of multiple resources. In *IEEE Real-Time Systems Symposium*, pages 90–101, 1999.
17. I. Shin, I. Lee, and S.L. Min. Embedded system design framework for minimizing code size and guaranteeing real-time requirements. In *Proceedings of IEEE Real-Time Systems Symposium*, December 2002.
18. O. Sokolsky, I. Lee, and H. Ben-Abdallah. Specification and analysis of real-time systems with PARAGON. *Annals of Software Engineering*, 7:211–234, 1999.

Architecture Based Model Driven Software and System Development for Real-Time Embedded Systems

Bruce Lewis

US Army Avionics and Missile Systems Command
Bruce.Lewis@sed.redstone.army.mil

Abstract. Architecture Description Languages provide significant opportunity for the incorporation of formal methods and engineering models into the analysis of software and system architectures. A standard is being developed for embedded real-time safety critical systems which will support the use of various formal approaches to analyze the impact of the composition of systems from hardware and software and which will allow the generation of system glue code with the performance qualities predicted. The standard, the Avionics Architecture Description Language (AADL), is based on the MetaH language developed under DARPA and US Army funding and on the model driven architectural based approach demonstrated with this technology over the last 8 years. The AADL standard will include a UML profile useful for avionics, space, automotive, robotics and other real-time concurrent processing domains including safety critical applications. The paper provides an overview of the concepts supported in MetaH and the AADL as examples of the architecture based model driven paradigm and notes several new model based approaches becoming available.

The Need for an Industry Solution

Complex embedded real-time systems are very expensive to develop, to maintain and to evolve. When system lifecycles are long compared with the commercial lifecycle of its components and when system capabilities must change to keep up with advances, evolvability becomes a key component in reducing the cost of keeping systems up to date. Families of systems also need a simple approach for evolving new members of the family from a baseline configuration. Component changes impact system performance in many areas such as speed, schedulability, reliability, and safety and must be made quickly and predictably against these cross cutting dimentions of system performance to keep costs reasonable. Predictive models in the past have been very difficult to apply due to the complexity and size of the systems and a lack of an approach which integrates the models with system development so that the models are updated with the system, accurately reflecting the implementation.

A Solution with Potential for Radical Impact

An architecture based, model driven approach provides the foundation to make computer based system changes and predict the impact. It provides analysis at the natural

M. Wirsing et al. (Eds.): RISSEF 2002, LNCS 2941, pp. 249–260, 2004.

level for change, the software and hardware component. Component level changes that affect system performance are represented through component attributes necessary to perform the system level analysis. Component level analysis is handed with separate tools, component level development is by conventional methods, hand coded, reused or generated. The challenge is to develop sound engineering models for a class of behaviors that are at the heart of large complex real-time systems. Some very helpful models exist now that can be applied to system development. The benefit is the rapid construction to specification with predictable behavior for the development and evolution of trusted systems. An additional challenge is the extension of these concepts to new domains of system architecture.

The capture of the architectural specification and the component attributes is accomplished with an Architecture Description Language (ADL). The language must be capable of expressing the architectural requirements for the domain of application in a way that engineers working in the domain can easily specify and modify systems. The ADL and tools must permit partial specification to allow for early analysis and prototyping, with incremental completion as additional information or analysis become available. Analysis must be efficient, adequate and automated so that it will always be performed. The analysis and system building capabilities must provide support in major areas of concern in the domain. The ADL must support multiple forms of analysis so that a change in a component or a change in the architecture can check multiple areas of possible impact without changing to alternative specification languages. For instance, moving a component from one processor to another can impact scheduability on the bus, scheduability on the processor, system optimization, multi-level safety relationships and system reliability.

The MetaH ADL and toolset provides such an environment. Our experience using MetaH over the last 8 years has convinced us that architecture based model driven approaches can revolutionize the development of real-time systems.

Defining the Needed Models

An engineering model is defined for the purposes of this paper as a body of knowledge that provides a clear and concise notation for describing significant system behaviors; that applies to a broad class of systems within the domain; that provides repeatable methods without custom tuning for producing good implementations; and that is supported by analytic methods to predict and verify behavior. These criteria were used in the development of the MetaH language and code generation to select the methods that would be applied to system architectural analysis. Examples of such methods are preemptive real-time scheduling theory, fault tree and Markov chain models, feed-back control theory, and real time message passing and timing models. These are widely used in successful engineering practice within the domain. Each was used in MetaH to define the language, construct the toolsets to analyze the specification and construct the code generator used to build the executive and to integrate the system. Additional key characteristics of these models are that the modeling approach can be supported by efficient code generation, necessary attributes can be determined or estimated to form profitable analysis, and that the models provide a bound for the designed system (schedulable processors and busses, mean time to failure, can affect higher level safety or security components when failing) that can be checked each

time the system is updated. Good engineering models are typically produced by a combination of theoretical and empirical work.

Additional models, beyond foundational models that are reflected within the ADL itself, can be added. The emerging AADL standard (based on MetaH) permits extension to the set of standard component attributes to enable the user or vendor to add new modeling approaches and analysis tools. For example, attributes can be extended to incorporate constraint analysis so that mathematical functions can be applied to insure that high level attributes are not exceeded by the composition of lower level component specifications. The European Space Agency in its experiments has used such constraints to add modeling of the weight of a satellite but they could be used as well for power and memory consumption. Attribute extensions could also be used to add additional types of dependency analysis, record or model the results of various types of safety analysis, provide for the assessment of various qualities of service and support new scheduling methods. Attribute sets developed by organizations may be contributed to the standardization committee for possible inclusion in future versions of the standard.

Engineering models are essential for many of the "ilites" we desire in system development: scalability (adding processors, processes, devices, then understanding quickly the effect and generating the compliant glue code to integrate the system), evolvability (incrementally changing the architecture during development and over the lifecycle, reviewing the impact via checking the models, provides strong support for evolutionary development methods such as the spiral development approach), dependability (understanding the impact of composition of components with different fault models and the effect of fault tolerant approaches), composability, and reusability (providing a precise specification of component attributes so its fit may be analyzed). Engineering models are an essential basis for requirement analysis, specification languages, and for tools to automate various development activities such as design analysis and code generation.

MetaH as an Example

Since our insight into model based real-time development is through MetaH, the following sections provide some history, a high level overview and a comparison of the current typical development process to the model based process using MetaH. The paper also includes some data on the success of the approach.

History

MetaH was developed over two DARPA programs and has greatly benefited from additional funding from the Open Systems - Joint Task Force. The first, Domain Specific Software Architectures (DSSA), was concerned with using domain specific system engineering knowledge to build Architecture Description Languages (ADLs) that could specify software architectures and analyze architectural properties. MetaH leverages the rapid construction of systems further by adding the automatic integration of hardware and software in accord with modeling used to analyze the system. The sec-

ond, the Evolutionary Design of Complex Systems (EDCS), was focused on our ability to build systems that could be rapidly evolved with predictable impact. A third DARPA program, Dynamic Assembly for System Adaptability, Dependability and Assurance (DASADA) is leveraging MetaH as an embedded systems ADL. The Open Systems - Joint Task Force (OS-JTF) has supported analysis of open standards using MetaH as well as the standardization of the Avionics Architecture Description Language (AADL) based on MetaH.

Dr. Steve Vestal of the Honeywell Technology Center has been the principal investigator. Bruce Lewis has served as technical POC on both DARPA programs and has led the US Army Aviation and Missile Command, Research Development and Engineering Center, Software Engineering Directorate (SED) laboratory demonstrations and technology integration with MetaH since 1993. The US Navy, US Air Force, the Ada Joint Program Office, and the US Army Space and Missile Defense Command have also funded MetaH related projects.

A High Level Overview

The MetaH language provides a system designer a simple but precise language for specification of architectural requirements. From the specification, the toolsets extract the formal modeling parameters for multiple analyses. MetaH was designed to integrate the multiple domains of application software in avionics on a generated architectural backplane based on formal scheduling and implementation methods while guaranteeing a hard real-time schedule. It comprehends both hardware and software components. It will generate the architecture integrating the hardware and software components into a system complaint with the modeled behavior. It provides a means to port hard real-time systems across execution environments rapidly. Current architectural analyses include schedulability, reliability and safety/security.

The language and toolset were developed to meet the requirements for building state-of-the-art modular multiprocessor systems with multilevel safety, security, and fault management. It provides for the specification and generation of dynamic multimode behavior across multiple processors under the real-time constraints of flight control. It also can build from the specification advanced space and time partitioned systems enabling very significant reductions (estimated at 80%) in re-validation and re-testing costs when changes are made to an avionics or mission critical system. The approach of using a combination of time and space partitioning is an emerging commercial avionics technology, now part of several fielded commercial avionics systems. Space and time partitioning is also valuable in military systems for the same reasons, especially with the recent requirements on some military avionics systems for FAA certification.

Model Based Development Process

Model based development has many advantages over conventional software development and hardware / software integration. To illustrate this, we compare the current

software development approach to the approach we recommend based on MetaH's capabilities for integrated modeling, architecture description, and system integration.

Figure 1 shows the current typical software development paradigm with its specification of requirements and design in various media, case tools, prototypes, paper, disconnected models, spreadsheets etc. Integrated Project Teams alleviate some of the communication problems in this "Over-The-Wall" approach, but this approach is characterized by specification for human interpretation and system construction. Evaluations of architecture may occur with requirements modeling tools and simulations but the results are reduced again to paper for impact on the final system software. Modeling results tend to be disconnected from the next phase and from each other. Multiple complex modeling languages are required, one for each system analysis area requiring high levels of expertise. Traceability of the models to the implementation is usually rapidly lost during development. Integration of components into a system is manual and yet must be traceable to the models if they are used. Integration is often difficult, complex and very expensive. Code generation for system or component analysis is typically for prototyping and requirements are again specified for human development of a traceable, testable integrated system.

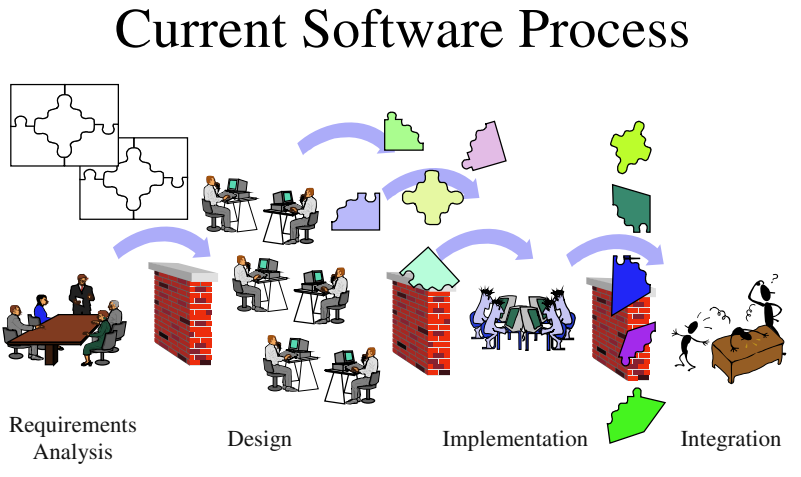


Fig. 1. Manual, paper intensive, error prone, resistant to change

Figure 2 shows the architecture based model driven approach [1][8][9] which provides the ability to specify an architecture (consisting of software and hardware components, their interfaces and the system execution behavior), analyze its properties, populate it with source components, integrate hardware components, and then automatically build the system to the specification. First, using specified attributes for the hardware and software components and connections within and across processors, the architectural specification is used to model and analyze schedulability, reliability (fault handling), and safety/security dependencies. Modeling is incremental; the user fills out the attributes required for the modeling desired. For instance if safety level is not filled out for the components, the multilevel safety analysis is not performed. The analysis for each model is provided each time the architecture is modified and regenerated so the impact can be noted in multiple areas of concern. The system is checked

against these models, warning the user if any used analysis is out of limits. Different versions of software or hardware components can be stored on the "bookshelf" and easily substituted in the architecture. Architectural changes are easily made and analyzed if the models for the components (as expressed as attributes of system level interest) are provided.

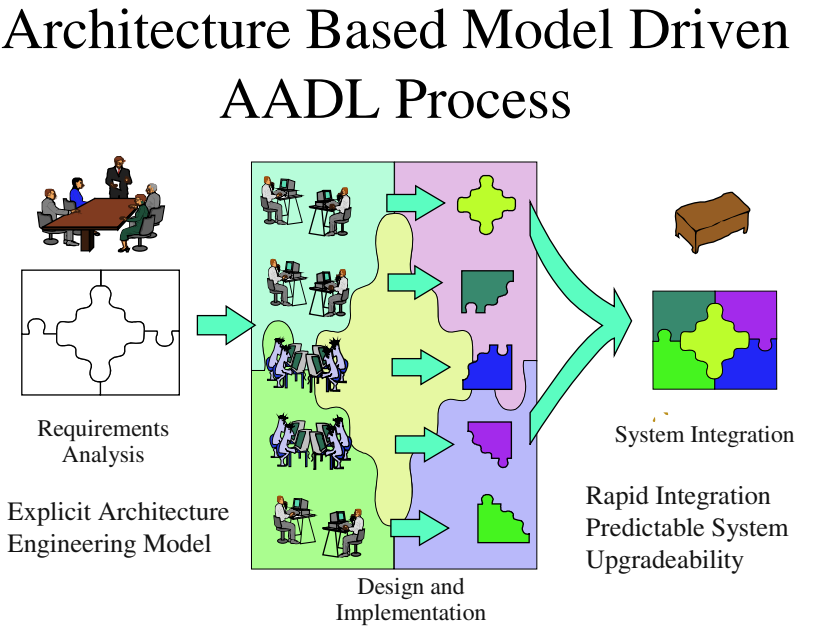


Fig. 2.

Additional model analyzers can be added as they are defined or improved to provide modeling approaches which allow specification of more complex or larger systems. We are currently working with the University of Illinois to increase the size of the reliability models we can analyze by two orders of magnitude. The AADL draft standard provides for attribute extension allowing users to add attributes and analyzers.

For real-time architectures, the issues of schedulability, reliability and partitioning need to be understood early in safety and time critical systems [2][4][5][8]. Once the systems engineer is satisfied with the architecture, the components can be developed, reused from another project, or generated in parallel with incremental automated integration of the system with MetaH. The system is easily re-integrated through regeneration from the specification. Typically, early integrations may be on a workstation where behavior and system output can be validated. The final system would be automatically integrated from the specification and components, hardware and software, on the target platform where execution behavior and results can again be validated.

A major benefit is that the architecture and execution behavior specified is captured, not in paper or the heads of the designers, or in scattered databases but in one

specification that integrates the final system and generates the executive that drives its execution. Also a single architectural specification is used for multiple formal analyses and therefore the system is generated compliant with each of the models used for analysis.

If, in the process of development, the system needs to be changed, the specification is updated and the system is rebuilt. Specification changes allow scaling across multiple processors, adding new modes, changing the execution environment (new processor(s), compilers, O/S), moving processes across processors, tuning execution rates, message passing, event propagation, substituting new components, changing process interfaces, changing error models, safety or security levels, etc. Since the processor, buses, or other hardware devices are part of the architecture they can quickly be changed to any of a predefined set. The defined set is user expandable. Execution environment dependencies reside in the toolset rather than the application code allowing rapid ports to new environments based on the toolset porting cost, not the size or complexity of the application.

The systems engineer benefits from the power of the modeling approach without having to be an expert in it or having to implement the system in conformance to it manually. The timing and data movement semantics required to implement control applications are part of the semantics of the language. So are the elements required for execution and data movement for minimal latency. The engineer has to understand how and where to specify what he needs. He constructs the architecture by specification and analyzes the impact using the models provided.

Evidence of Success

The SED developed a generic missile architecture and used it to re-engineer the onboard software of an Army missile system. MetaH was used to specify the architecture interfaces and timing and to integrate re-engineered components and the dual 80960 embedded hardware together to create an executable system [3]. In addition, a 6 Degree of Freedom (6DOF) Software-In-The-Loop missile flight environment simulation was developed to allow us to evaluate flight characteristics. It was also re-engineered into MetaH so it could be executed in real-time.

During this first development, the MetaH model based approach reduced the total effort to re-engineer the system and fly software-in-the-loop by 40 %. This estimate was made by our system and our hardware/software engineer, both of whom have extensive experience in the design and construction of real-time systems. Given the possible differences in skill level in dual implementation approaches, we believe this approach gave us the best estimate. Their estimates are based on what it would have taken if they did the same task without the aid of the ADL and the associated toolset. We purposely made the estimate conservative. To further verify our estimates we presented the work to several prime contractors, including the one who developed the missile system. They put the estimated savings at 66%. We later ported the application to several new execution environments. A single processor Aonix, Parr Lap OS, Pentium, dual Pentiums on VME, and a port using Green Hills, PowerPC and VxWorks.

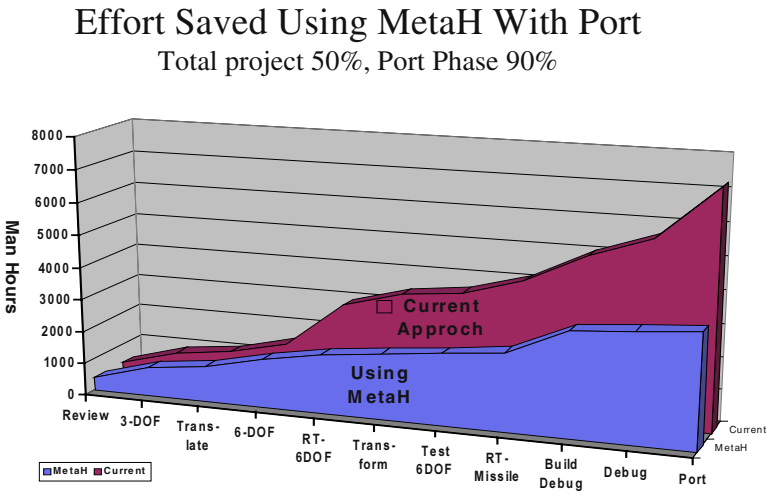


Fig. 3.

Our Software Engineering Directorate lab accomplished the MetaH toolset port in less than a week and the missile flew correctly in a simulation the first time it executed on the new embedded environment. Our estimated savings moved to 50% [9].

While the ADL significantly reduced the work in porting the application by capturing the real time specification and accurately reproducing the real time performance, it should be recognized that hardware oriented ports from a cost perspective are difficult to predict. The risk of problems with hardware drivers, operating system interfaces, compiler and O/S incompatibilities and hardware bugs can always extend the cost. A low risk approach is to use the provided hardware targets in the toolset or have the tool vendor provide new targets. We did our own retargeting to demonstrate that it could be done.

Since the missile architecture was not safety critical we did not experiment with the reliability and safety analysis. We expect that the savings provided by early analysis of reliability and safety partitioning will also provide significant benefit. Several studies of helicopter avionics architectures have been preformed by Honeywell. A significant avionics experiment using the additional modeling capabilities is needed, along with other complementary technologies, to demonstrate more convincingly the benefits.

Rising Support for Architecture Based Model Driven Approaches

Based on experience from a number of experiments and applications of the technology on various DARPA programs since 1993 and based on avionics and space requirements from industry, MetaH is being used as the baseline definition for the specification of the AADL under the Society of Automotive Engineers, Avionics Division. See figure 4.

MetaH History

1991 DARPA DSSA program begins
1992 First partitioned target operational (Tartan MAR/i960MC)
1994 First multi-processor target operational (VME i960MC)
1998 Portable Ada 95, POSIX executive configurations
Example evaluation and demonstration projects
- Missile G&C reference architecture (AMCOM SED)
- Missile Re-engineering demonstration (AMCOM SED)
- Space Vehicle Attitude Control System (AMCOM SED)
- Reconfigurable Flight Control (AMCOM SED)
- Hybrid automata formal verification (AFOSR, Honeywell)
- Missile defense (Boeing)
- Fighter guidance SW fault tolerance (DARPA, CMU, Lockheed-Martin)
- Incremental Upgrade of Legacy Systems (AFRL, Boeing, Honeywell)
- Comanche study (AMCOM, Comanche PO, Boeing, Honeywell)
- Tactical Mobile Robotics (DARPA, Honeywell, Georgia Tech)
- Advanced Intercept Technology CWE (BMDO, MaxTech)
- Adaptive Computer Systems (DARPA, Honeywell)
- Avionics System Performance Management (AFRL, Honeywell)
- Ada Software Integrated Development/Verification (AFRL, Honeywell)
- FMS reference architecture (Honeywell)
- JSF vehicle control (Honeywell)
- IFMU reengineering (Honeywell)

Fig. 4.

Both European and US companies/agencies are participating in the standardization effort. The standardization committee also has a liaison relationship with NATO's Aviation Standard's Committee and we informally coordinate with the Open Group and with UML committees as we have opportunity to attend the meetings. See figure 5 below.

AADL Standard Participants

- Bruce Lewis: Chair, technology user
- Ed Colbert: AADL & UML Mapping
- Peter Feiler: Secretary, Co-author, Editor, technology user
- Steve Vestal: MetaH Originator, Co-author

Members

- Boeing, Rockwell, Honeywell, Lockheed Martin, Raytheon, Smith Industries
- NIST, NAVAir, OSJTF, British MOD
- Dassault Aviation, Airbus, European Space Agency, COTRE

Fig. 5.

The same concepts that provide an Architecture Based Model Driven approach in MetaH will be part of the AADL. The standard provides a means for the commercial production of tools with a common AADL language interface. The UML profile, a

specialization providing AADL semantics, will allow the application of formal analysis and code generation tools through a UML graphical specification, enabling the use of currently available UML tools for specification. The UML profile is being developed in parallel with the AADL standard and will be provided as an appendix. We also plan to provide an XML specification for the AADL language once the first version of the language standard is completed. These capabilities will provide an early interface for developing new analysis approaches and integrated use of toolsets. We will ballot the first version of the standard in December of 2003.

The AADL Standardization Subcommittee also has a liaison relationship with a French research consortium, COTRE, headed by Airbus. COTRE has adopted the AADL for research into new tools, development and analysis methods to support aviation system development requirements. The AADL plays a significant role in a future software and systems development approach described by Airbus and COTRE in a recent paper[10]. Other US and European companies and agencies are evaluating and experimenting with MetaH or the AADL, including Future Combat Systems.

Architecture based, model driven approaches are also beginning to appear in the general software engineering domain. UML 2.0, the Model Driven Architectures Initative [11], will provide a new layer to UML to directly support a generalized Model driven architecture based approach. See figure 6. It is expected that multiple profiles for different domains will be defined as specializations of UML 2.0. UML 2.0 is expected to be released in mid 2003. The AADL UML profile will incorporate new architecture description capabilities from UML 2.0 when its released. The AADL UML profile will be based on UML 2.0.

The Model Driven Architecture (MDA) Initiative

- Based on the success of UML, the OMG has formulated a vision of a method of software development based on the use of models
- Key characteristic of MDA:
  - The focus and principal products of software development are models rather than programs
  - "The design is the implementation" (i.e. UML as both a modeling and an implementation language)
- UML plays a crucial role in MDA
  - Automatic code generation from UML models
  - Executable UML models
  - Requires a more precise definition of the semantics of UML (UML 2.0)

Source: Bran Selic, Rational

Fig. 6.

The University of Southern California, Center for Software Engineering, lead by Barry Boehm, has announced the development of Model-Based Architecting and

Software Engineering (MBASE) approach [12]. This approach currently is being developed to be compatible with several Architecture Description Languages, one being the AADL.

Conclusions

Initial demonstrations of the architecture based model driven approach with MetaH has shown significant promise for real-time avionics and related domains. Larger demonstrations and technology extension, including research and development of new modeling approaches will add to the paradigm's power for rapid, low cost development and evolution of computer based systems. The power of the approach is strongly related the types of components and communications that can be described, to the models supported and to the automation of the system building process. The AADL Standard will significantly increase the power of MetaH to express architectures, extendable attributes will support many new modeling approaches and the formalization of the language will still support an automated system build process. The UML profile will leverage existing UML toolsets and XML will ease toolset integration. The release of UML 2.0 will focus significant industry interest in the model driven approach. Significant research into modeling focused on predicting system level properties from component attributes is needed and can be significantly leveraged by Architecture Based Model Driven methods to reduce the cost of system development and change.

References

1. Pam Binns, Matt Englehart, Mike Jackson and Steve Vestal, "Domain Specific Software Architectures for Guidance, Navigation and Control," Honeywell Technology Center, Minneapolis, MN, International Journal of Software Engineering and Knowledge Engineering, Vol6, No. 2, 1996, pages 201-227.
2. Mark H. Klein, John P. Lehoczky and Ragunathan Rajkumar, "Rate-Monotonic Analysis for Real-Time Industrial Computing," IEEE Computer, January 1994.
3. David J. McConnell, Bruce Lewis and Lisa Gray, "Reengineering a Single Threaded Embedded Missile application onto a Parallel Processing Platform using MetaH," 5th Workshop on Parallel and Distributed Real Time Systems, 1996.
4. Farnam Jahanian and Aloysius K. Mok, "Modechart:A Specification Language for Real-Time Systems," IEEE Transactions on Software Engineering, V20, N12, December, 1994.
5. Andrew L. Reibman and Malathi Veeraraghavan, "Reliability Modeling: An Overview for Systems Engineers, "IEEE Computer, April 1991.
6. Software Considerations in Airborne Systems and Equipment Certification, RTCA/DO-178B, RTCA, Inc., Washington D.C., December 1992.
7. Pam Binns, "Scheduling slack in MetaH," Real-Time Systems Symposium, December, 1994.
8. Jonathan W. Kruger, Steve Vestal, Bruce Lewis, "Fitting the Pieces Together: System/Software Analysis and Code Integration Using MetaH, " Digital Avionics Systems Conference, 1998.

9. Peter H. Feiler, Bruce Lewis, Steve Vestal, "Improving Predictability in Embedded Real-time Systems," Carnegie Mellon Software Engineering Institute, CMU/SEI-2000-SR-011, October 2000.
10. Patrick Farail, Pierre Dissaux, "COTRE a Software Design Workshop", DASIA 2002, May 2002.
11. Bran Selic, "Performance Oriented UML", Tutorial, 3rd International Workshop On Software and Performance, July 2002.
12. Barry Boehm, Overview, Mini Tutorial, http://sunset.usc.edu/research/MBASE/

A Computational Model for Complex Systems of Embedded Systems

Luqi, Ying Qiao, and Lin Zhang

Naval Postgraduate School, Software Engineering Automation Center
Monterey, CA93943
{Luqi,yqiao,lzhang}@nps.navy.mil

Abstract. Systems of embedded systems (SoES) have been receiving a great amount of attention because of their capabilities of combining individual embedded systems to accomplish broad and common objectives. High confidence is critical for SoES since they are widely used in the fields in which the consequences of systems failure are very serious. Although many approaches have been proposed to construct high-confidence embedded systems, most of them are proposed for the development of monolithic embedded systems. Building high-confidence SoES is still a great challenge. Further efforts on studying novel technologies for the development of SoES are needed. In this paper, we present a computational model which serves as a basis to construct prototype of SoES.

1 Introduction

Requirements for individual embedded systems working together towards broad, common objectives have been receiving greatly increased amounts of attention because of large investments in existing isolated embedded systems. Systems of embedded systems (SoES) should be distinguished from large and complex but monolithic embedded systems by the independence of their components, their evolutionary nature, emergent behaviors, and a geographic distribution [1]. Furthermore, high confidence is critical for SoES since it has been widely used in many fields, such as flight control and avionics, process control, weapon system control and nuclear plant control, in which the consequences of failure are very serious.

High confidence is a perception of how a system should behave. It can only be established by measurable or certifiable attributes. Some relevant attributes of interest for high-confidence embedded systems are functional correctness, timely response, reliability, safety, security, robustness, testability and maintainability etc.

Constructing high-confidence systems of embedded systems is a great challenge. This is because the individual component systems are independently developed and therefore typically run on different platforms and have different data formats and interaction protocols. Novel technologies to construct high-confidence SoES are needed to overcome these challenges in an effective and sound way. At the same time, accurately describing requirements is the most important step in the development of SoES since requirements specification serves as a basis and start point for all further development activities such as prototyping, verification, and code generation

M. Wirsing et al. (Eds.): RISSEF 2002, LNCS 2941, pp. 261–273, 2004.

etc. Thus, in this paper, we present a computational model for SoES to capture the functional and non-functional aspects of requirements for SoES.

The rest of the paper is organized as follows: section 2 introduces some related works; section 3 presents the computational model for SoES; section 4 is conclusion and future work.

2 Related Works

In recent years, many approaches have been proposed to construct high-confidence embedded systems. Those approaches include verification, run-time testing and monitoring, and component-based composition.

[2], [3], [4] and [5] are some typical research works for the approach based on verification. These works use model checking to verify the satisfaction of some properties such as survivability properties. [2] uses a model checker based on NuSMV to verify the survivability properties in embedded systems. [3] focuses on the specification and analysis of publish-subscribe software architecture. It specifies the properties in linear temporal logic and uses the SMV model checker to complete the verification. [4] gives an efficient procedure for verifying that a finite-state concurrent system meets a specification expressed in a (propositional, branching-time) temporal logic. In [5], the requirements are formalized in temporal logic and the system model is abstracted from the source codes so that some real-time properties are verified. [6] focuses on the verification of real-time systems. It presents a modular framework for providing temporal properties of real-time systems basing on clocked transition systems and liner temporal logic. In this framework, the properties of real-time systems can be established by use of deductive verification rules, verification diagrams and automatic invariant generation.

For run-time testing, both [7] and [8] attempt to automatically generate the test suite from the formal requirement specification. Furthermore, [7] uses model checking while [8] uses a test derivation schema to achieve this purpose. [9] presents a comprehensive method for run-time monitoring. It uses a monitor script to generate a filter and event recognizer and generates a run-time checker basing on a formal specification so that a set of resource-specific, safety and real-time properties are monitored and checked at run-time.

For research on component-based composition, [10] presents a method for automatic integration of reusable embedded systems. This research uses a component model, a component behavior model and a control plan to compose reusable components in embedded systems. Timing constraints and resource constraints are considered during the component composition.

However, most research focuses on development methods for building high-confidence monolithic embedded systems rather than the high-confidence SoES. Further efforts on studying novel technologies for high-confidence SoES development are still needed. Moreover, development of large software systems causes a sequence of modeling tasks. It requires the modeling and description of the application domain, software requirements, software architecture, software components, programs, their internal structure and their implementation be it by one or by several modeling concepts [11]. In this paper, we present a computational model which serves as a basis for development of high-confidence SoES.

Furthermore, many approaches assume the existence of an accurate and valid formal specification of the properties that are to hold with high confidence. The last thirty years of computing research have shown that this assumption does not hold in practice, and that a majority of software faults can be attributed to requirements and specification errors. Prototyping has emerged as a preferred method for requirements validation. We therefore explore support for prototyping of SoES as a complement to approaches for certification. The main purpose of this work is to ensure that the specified properties, to be certified by other methods, are valid with respect to the real-world context of the SoES and the real-world needs of its stakeholders.

3 Computational Model for SoES

Computational model is a mathematical model that describes requirements of SoES. It has two views: external view and internal view. The external view model describes requirements from user's view while the internal view model is more close to the design's view.

3.1 External View Model

External view model describes the requirements via functional and non-functional emergent properties. In this context, emergent properties refer to the properties of entire SoES that do not reside in any individual embedded system. They describe additional requirements to be met by the integration of the component systems. They represent the value added by the interaction, and are typically not satisfied when the component systems are all operated as isolated systems.

Formally, the external view computational model can be represented as follows:

$$\zeta' = (G, H) \tag{1}$$

G is a functional emergent property vector which represents the functional requirements of SoES, $G = (g_1, g_2, \ldots, g_l)$. $g_i (i \in [1, l])$ denotes one of functional emergent properties of SoES. l is the number of functional emergent properties. The most typical functional emergent properties identified in external view model are timing properties such as maximum response time (MRT).

H denotes non-functional emergent properties, i.e., high-confidence properties that reveal the non-functional requirements of SoES. Since high confidence can only be established by measurable attributes, these non-functional emergent properties need some practical metrics that can be automatically measured by a repeatable process. Thus, we use a high-confidence metric vector to describe high-confidence properties. In this context, $H = (h_1, h_2, \ldots, h_z)$, where $h_i (i \in [1, z])$ is the set of metrics for a measure of high confidence. The entries of the sets are monotonically increasing real functions, where larger metrics values denote better confidence.

There are several widely accepted measures related to different aspects of high confidence, such as availability, reliability, safety, security, maintainability and integrity, etc. Furthermore, there are one or more than one corresponding metrics for each

measure. Some typical metrics are failure rate, maximum time between two successive failures, the number of faults that can be tolerated, maximum time between safety violations and security level etc. The required confidence level with respect to the specified application field can be represented by a vector of threshold values $W = [w_1, w_2, \cdots, w_z]$. The vector represents the following high confidence goal, i.e., for all $i \in [1, z]$, $h_i \geq w_i$.

G and H reflect the functional and non-functional aspects of requirements for whole SoES. G can be used for (a) generating code for monitoring failure events during the reliability assessment and testing and (b) verification. H can be used for (a) experimental assessment of high-confidence attributes and (b) possible static verification for some metrics, such as Maximum Execution Time (MET).

3.2 Internal View Model

Internal view model describes requirements of SoES via four elements, i.e., component systems, interactions between component systems, local constraints imposed on component systems and interactions between them. It is formally described bellow:

$$\zeta = (S, E, C, D, F_1, F_2) \tag{2}$$

S is the component system set, $S = \{s_i \mid i \in [1, n]\}$. s_i denotes the component system constituting SoES (n is the number of component systems in the whole SoES); E denotes the interaction sets between component systems, $E = \{e_{jk} \mid j, k \in [1, n]\}$, Where e_{jk} denotes the set of interactions from component system s_j to component system s_k; e_{jk} is a set of Γ_{jk} elements. When $\Gamma_{jk} \geq 1$, $e_{jk} = \{e_{jk}^{\gamma} \mid \gamma \in [1, \Gamma_{jk}]\}$; it means that there is more than one interaction from s_j to s_k. When $\Gamma_{jk} = 0$, e_{jk} is empty; this means that there is no interaction from s_j to s_k.

C denotes constraint sets on how the component systems are used in the given environment, $C = \{c_i \mid i \in [1, n]\}$. c_i is a set of constraints with P_i elements. When $P_i \geq 1, c_i = \{c_i^p \mid p \in [1, P_i]\}$, c_i^p denotes the pth constraint for s_i. When $P_i = 0$, c_i is empty; it means that there is no constraint on s_i. D denotes constraint sets on interactions between component systems, $D = \{d_{jk} \mid j, k \in [1, n]\}$, where d_{jk} is a set of constraints with Q_{jk} elements each of which applies to interactions in e_{jk}. When $Q_{jk} \geq 1$, $d_{jk} = \{d_{jk}^q \mid q \in [1, Q_{jk}]\}$, d_{jk}^q denotes the qth constraint for e_{jk}. When $Q_{jk} = 0$, it means that there is no constraint on e_{jk}.

Constraint sets C and D are determined by functional and non-functional emergent properties in external view model, and $C = F_1(G, H)$; $D = F_2(G, H)$, where F_1 and F_2 are two mappings that map emergent properties of SoES into local constraint sets imposed on component systems and interactions between component systems respectively.

In fact, s_i and e_{jk} describe the static structures of SoES while c_i and d_{jk} describe the dynamic behavior of systems of embedded systems. The mappings specify what must be assessed to ensure that the SoES satisfies its requirement with high confidence, if it has already been certified that the individual s_i meet their requirements with high confidence. The constraint sets also represent a design for the systems integration, which will be realized by wrappers around the s_i.

3.2.1 Component Systems

Component systems are a set of independent complex embedded systems. Each of them can fulfill its purpose if disassembled from the overall system, and can be managed (at least in part) for their own purpose rather than the purpose of the whole. As a matter of fact, a component system is a slot rather than a single fixed subsystem. The slot can be filled by any subsystem that meets the constraints specified at the level above. The analysis that leads to the assurance of high confidence should depend only on the constraints associated with the slots. In this way, if we have a set of concrete subsystems for each slot that have been certified to meet the constraints associated with that slot, then we can reconfigure SoES by plugging in different certified subsystems into corresponding slots without need for further recertification for each reconfiguration. Such a structure is needed to support rapid and low cost evolution without compromising high confidence.

Each component system is either atomic or composite. Composite component systems can be described by a hierarchical SoES computational model, i.e., it can be realized by interaction-link networks of lower level component systems while atomic component system can not be decomposed in terms of the SoES computational model.

In order to represent the hierarchical structure of the composite component system, we introduce the *layer* for each component system. Each composite component system in *a higher layer* is composed of several sub-component systems in *the next lower layer*. In this case, the whole system of embedded systems is in the *root layer* which is the highest layer. Component systems that constitute SoES belong to the *layer 1* which is just below the *root layer*.

For any component system in layer θ $(\theta > 0)$, we can also describe it from external view and internal view. Thus, we use $s_i^{'\theta}$ and s_i^{θ} to respectively denote the external and internal view model of the component system.

In the external view model, the hierarchical structure of the composite component system can be formally represented as follows:

$$s_i^{'\theta} = (G_i^{\theta}, H_i^{\theta}) \tag{3}$$

G_i^θ denotes the functional emergent property vector for component system in layer θ while H_i^θ represents the non-functional emergent property vector relevant to high confidence for component system in layer θ. G_i^θ and H_i^θ are decomposed from $G_i^{\theta-1}$ and $H_i^{\theta-1}$ for component system in the higher layer $\theta-1$.

Furthermore, although emergent properties are bottom-up, it is still meaningful to decompose them in SoES. The decomposition of emergent properties can guide the designer to choose suitable component systems to build SoES that meets the requirements. This provides a systematic way to construct SoES rather than arbitrarily choosing current component systems to compose SoES which is hard to satisfy the requirements.

In the internal view model, s_i^θ is formally represented as follows:

$$s_i^\theta = (S_i^{\theta+1}, E_i^{\theta+1}, C_i^{\theta+1}, D_i^{\theta+1}, F_{i,1}^{\theta+1}, F_{i,2}^{\theta+1}) \tag{4}$$

$S_i^{\theta+1}$ denotes the component system set in the lower layer $\theta+1$ that constitutes s_i^θ, and $S_i^{\theta+1} = \{s_{i,t}^{\theta+1} \mid t \in [1, n_i^{\theta+1}]\}$. $n_i^{\theta+1}$ is the number of sub-component systems in lower layer $\theta+1$; $s_{i,t}^{\theta+1}$ is the t-th sub-component system in lower layer $\theta+1$ which constitutes s_i^θ. $E_i^{\theta+1}$ represents the interaction sets in lower layer $\theta+1$, and $E_i^{\theta+1} = \{e_{i,jk}^{\theta+1} \mid j,k \in [1, n_i^{\theta+1}]\}$. $e_{i,jk}^{\theta+1}$ denotes the interaction set between the sub-component system $s_{i,j}^{\theta+1}$ and the sub-component system $s_{i,k}^{\theta+1}$ in lower layer $\theta+1$; it is a set of $\Gamma_{i,jk}^{\theta+1}$ elements, where $\Gamma_{i,jk}^{\theta+1} \geq 0$.

$C_i^{\theta+1}$ denotes the constraint sets on sub-component systems in lower layer $\theta+1$, and $C_i^{\theta+1} = \{c_{i,t}^{\theta+1} \mid t \in [1, n_i^{\theta+1}]\}$. Furthermore, $c_{i,t}^{\theta+1}$ is a set of constraints with $P_{i,t}^{\theta+1}$ elements, where $P_{i,t}^{\theta+1} \geq 0$. $D_i^{\theta+1}$ denotes constraint sets on interactions between sub-component systems in layer $\theta+1$, and $D_i^{\theta+1} = \{d_{i,jk}^{\theta+1} \mid j,k \in [1, n_i^{\theta+1}]\}$, where $d_{i,jk}^{\theta+1}$ is a set of constraints with $Q_{i,jk}^{\theta+1}$ elements each of which applies to interactions in $e_{i,jk}^{\theta+1}$.

Furthermore, as constraints on sub-component systems and interaction between sub-component systems in the lower layer can be derived from the decomposition of constraints in the higher layer, the relationship between constraints in two successive layers can be formally revealed by following equation:

$$C_i^{\theta+1} = F_{i,1}^{\theta+1}(G_i^\theta, H_i^\theta); \quad D_i^{\theta+1} = F_{i,2}^{\theta+1}(G_i^\theta, H_i^\theta) \tag{5}$$

where $F_{i,1}^{\theta+1}$ and $F_{i,2}^{\theta+1}$ are two mappings that map the constraint sets on components system set S_i^{θ} and interaction sets E_i^{θ} in layer θ into constraint sets on components system set $S_i^{\theta+1}$ and interaction sets $E_i^{\theta+1}$ in lower layer $\theta+1$ respectively.

Figure 1 is an example of a composite component system. In this figure, s_i^{θ} is a composite component system in layer θ. s_i^{θ} can be further decomposed into three sub-component systems $s_{i,1}^{\theta+1}$, $s_{i,2}^{\theta+1}$ and $s_{i,3}^{\theta+1}$ in θ+1 layer. Moreover, $e_{i,12}^{\theta+1}$, $e_{i,13}^{\theta+1}$ and $e_{i,23}^{\theta+1}$ denote the interactions between these three sub-components in layer $\theta+1$.

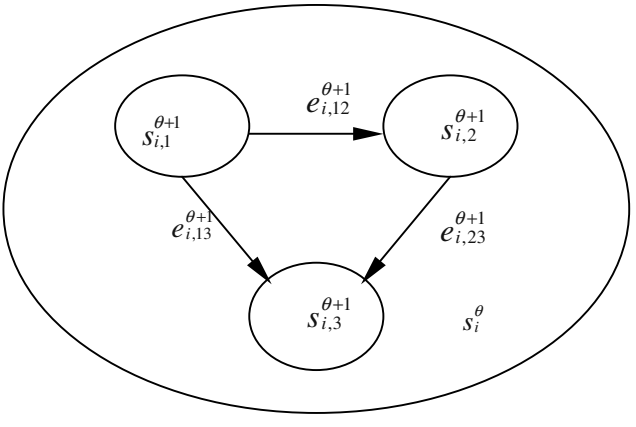


Fig. 1. Composite Component System

Moreover, both composite component systems and atomic component systems are composed of *an operator* and *a wrapper*. The component system in layer θ, i.e., s_i^{θ}, can be represented as follows:

$$s_i^{\theta} = (a_i^{\theta}, I_i^{\theta}) \tag{6}$$

Where a_i^{θ} denotes the operator; I_i^{θ} denotes a wrapper around component system s_i^{θ}.

The operator is the main part of a component system. It is either a functional abstraction or a state machine abstraction and encapsulates a thread of control. Furthermore, it is either timing triggered or event triggered. If the operator is timing triggered, the component system is triggered at fixed time instants and at approximately regular time intervals. However, if the operator is event triggered, the component system is triggered by the arrival of new interaction information such as a new data value, the occurrence of certain events etc.

Wrapper is a software layer around the component system. It has two functions: (a) providing the service interface to other component systems, and (b) intercepting all service calls from other component systems. The wrapper can solve the interface

mismatch between the heterogeneous component systems and provide the error containment and isolation, i.e., protect the wrapped component system from erroneous interacting component systems and also protect error propagation from the wrapped component systems to the outside world.

3.2.2 Interactions between Component Systems

Interaction sets E is composed of several interaction sets. For two specific component systems s_j and s_k, there is an interaction set e_{jk}. Each element in this interaction set, denoted by e_{jk}^{γ} $(\gamma \in [1, \Gamma_{jk}])$, is one interaction between component system s_j and s_k. It is formally represented as follows:

$$e_{jk}^{\gamma} = (s_j \, INT \, s_k, Y_{e_{jk}}) \tag{7}$$

Where, *INT* is an interaction operator; $Y_{e_{jk}}$ represents the interaction information involved in the interaction between component system s_j and s_k.

The interaction information has two types – data stream and event stream. A data stream is used to connect two or more component systems. It is widely used in the typical producer-consumer architecture. Each data stream carries a sequence of data values. It has the pipeline property: Assume Com-1 and Com-2 are two component systems. If *a* and *b* are two data values in data stream *Y* and the data value *a* is generated by Com-1 before the data value *b* is generated then it is impossible for *a* to be delivered to Com-2 after *b* is delivered. Furthermore, there are two types of data streams – data flow streams and sampled streams. The data flow stream guarantees that none of the data values is lost or replicated while a sampled stream does not make such a guarantee [12]. A data flow stream can be thought of as a FIFO queue, while a sampled stream can be thought of as a cell capable of containing just one value, which is updated whenever the producer or publisher generates a new value. Since embedded systems must often operate within a (small) bounded memory, the finite queue length imposes a restriction on the relative execution rates of two component systems communicating via data flow stream. However, a sampled stream imposes no such constraint, since it can deliver a value more than once if the consumer demands more values before the producer has provided a new value, and it can discard the previous value if the producer provides a new value before the consumer has used the previous one.

For example, suppose the component system Com-1 produces a sequence of data values $d_1, d_2, \ldots, d_{n-1}, d_n$ in data stream *Y*, and component systems Com-2 will consume it. A queued sequence for *Y* is shown below.

$$d_n d_{n-1} \cdots d_2 d_1$$

If *Y* is a data flow stream then Com-1 consumes all of the data values in data stream *Y* in the order $d_1, d_2, \ldots, d_{n-1}, d_n$. If Com-1 produces data values in data stream *Y* faster than Com-2 consumes them, the length of the queued sequence increases until an overflow occurs. If Com-2 consumes data values faster than Com-1 produces them, Com-2 has to wait until the queued sequence becomes nonempty. This implies

that with a data flow stream, Com-2 may not meet its real-time constraints unless its data rate closely matches that of Com-1.

If *Y* is a sampled stream instead, the data rates do not have to match. For example, consider the case where Com-1 produces one data value in data stream *Y* every 2 ms, Com-2 consumes one data value every 3 ms, and a size one buffer is used for containing the data values. Thus, since the buffer contains only one data value at a given time, the sequences of data values in sampled stream *Y* consumed by Com-2 starting at different initial times can be different. In this case, Com-2 consumes the values d_2, d_3, d_5, d_6, … with an initial time 2 and d_1, d_2, d_4, d_5, … with initial time 1.

In SoES, individual embedded systems have high autonomy and they are temporally combined together to archive a broad and common objective. It is often hard to confirm which component systems will be involved in the interaction in advance. Thus, the dynamic interaction is popular in SoES. However, data streams only can support static interaction. Since events are used to support publish/subscribe architectures, we introduce event streams in this computational model to support the dynamic interaction in SoES. Event streams are extensions of data streams that have additional operation for dynamic addition and removal of producer actions and consumer actions.

Each event stream is used to connect component systems that generate events to component systems that respond to events. The producers and consumers of an event stream can be added or removed during system execution. However, to ensure dependability of a system, a producer or consumer can be added to an event stream only if it has been pre-certified to meet the constraints associated with the interaction. During the interaction, the component system generating events is taken as the source of the interaction while component systems responding to these events are the destination of the interaction. Like data streams, event streams also have pipeline property. It is impossible that the destination component system responds to an event which is generated later before it responds to an event which is generated earlier.

There are two types of event streams – event flow streams and sampled event streams. These are special cases of the data flow streams and sampled streams described earlier.

3.2.3 Constraints on Component Systems and Interactions

In this computational model, constraints on component systems and interactions between component systems are divided into two categories, i.e., the constraints for the real-time features and the constraints for high-confidence features.

• *Constraints for Real-Time Features*

In the computational model supported by Prototype System Description Languages (PSDL)[12], a lot of work has been done for timing constraints and control constraints for real-time embedded systems. Although these two constraints were originally proposed for individual embedded systems, it is easy to extend them for the case of SoES.

In the case of control constraints for SoES, whether the operator is *PERIODIC* or *SPORADIC*, triggering methods, triggering conditions, and output guards are the major aspects to be specified. This computational model supports both periodic and sporadic component systems. Periodic component systems are triggered at the regular time interval while the sporadic component systems are triggered by the arrival of

new data values or the occurrence of certain events, possibly at irregular time intervals. For the trigger method, this computational model can support data trigger and event trigger. Both data trigger and event trigger have two types, i.e., *triggered by ALL* and *triggered by SOME*. In the case of *triggered by ALL*, the corresponding function of component system is ready to be activated whenever the new data values have arrived on all of the input data streams or the new event instances occur in all the input event streams. In the case of *triggered by SOME*, the corresponding function of component system is ready to be activated whenever any of the input streams gets new data value or the new event instance occurs in any one of the input event streams.

Furthermore, by extending the conditionals in the computational model supported by PSDL to the case of SoES, the trigger conditions and the output guards can be derived in this computational model. The trigger condition serves as a guard for the correspondent function of component system. Thus, only when arrival data in the input data stream or occurring event in the input event stream satisfies the trigger condition, the correspondent function of the component system can execute. The output guards act as the condition for the output of component systems, i.e., only when the generated data or the event satisfies certain conditions, they can be put in the output data stream or event stream of component system.

Timing constraints are critical for SoES. In this case, we can make use of the timing constraints specified in the computational model supported by PSDL [12]. *Maximum Execution Time* is one of the typical timing constraints for SoES. It can be imposed on the component system. It represents the upper bound on the length of time between the instant when a module in the component system begins execution and the instant when it completes. Furthermore, for the periodic component system, the timing constraints include *Period* and *Finish Within*. *Period* is the interval between consecutive trigger events while *Finish Within* is the upper bound between each triggering event and the completion of the activation. For the sporadic component system, the timing constraints include *Maximum Response Time* and the *Minimum Calling Period*. *Maximum Response Time* is an upper bound on the time between the arrival of a new data value or occurrence of a new event instance and the time when the last value or the state change is put into the output streams of the component systems in response to the arrival of new data value or the occurrence of new event instance. *Minimum Calling Period* is a constraint on the environment of a sporadic component system, consisting of a lower bound on the delay between the arrival of one set of inputs and the arrival of the next set. In this computational model, every sporadic component system with a maximum response time constraint must have a corresponding minimum calling period constraint.

In addition, we use *Latency* as the timing constraint imposed on the interactions between component systems. The *Latency* of an interaction is an upper bound on the time between the instant a data value is written into a data stream or an event is generated in an event stream and the instant that data value can be read from the stream or the event can be received from the event stream.

• *Constraints for High-Confidence Features*

To construct the high-confidence SoES, some constraints for high-confidence features that have not been previously included in the computational model supported by PSDL should be discussed.

In this computational model, constraints for high-confidence features are derived in the high-level of internal view model by mapping emergent properties represented in

external view model. Consequently, these high-level constraints can be further refined into lower level constraints.

Since *Reliability* is one of the most typical emergent properties related to high confidence, we take it as the start point to discuss constraints for high-confidence features in SoES. [13] attempts to refine *Reliability* into some constraints of fault tolerance. In this paper, we exploit this result to map reliability into the constraints imposed on component systems and interactions between component systems.

Reliability refers to the property of continuity of correct service. It is the probability that no failure occurs in the time period [0, t]. Reliability, denoted by $R(t)$, is represented by following equation [14]:

$$R(t) = e^{-\lambda t} \tag{8}$$

where λ denotes the failure rate.

According to above equation, reliability can be quantified into failure rate. Based on this, we can analyze which design parameter will impact the value of failure rate so as to find out constraints related to high confidence.

The failure rate refers to the number of failures in the unit time. As we know, fault tolerance is an important way to reduce the failure rate in SoES. This provides a potential way to map the failure rate into some concrete constraints of fault tolerance mechanism in the high-level internal view model.

Reliability states that, each faulty state in the execution trace of the system is followed by a state which is a superset of some states in the trace of the correct execution of some equivalent system, i.e., even if failure products are not removed, the behavior expected by a correct system execution will still be observed. Since masking of failures is a well-known fault tolerance abstraction and expresses the fact that a failure event is not perceived by some part of the failed system's surrounding environment, it captures the fact that a faulty state is followed by a state which is a superset of some states in the trace of the correct execution of the same system. Therefore, in the high-level internal view model, failure rate can be refined into the constraint of *masking* which indicates the basic style for the fault tolerance.

To strengthen the masking, one way is to combine the constraints of *DetectionObj* and *MaskObj*. In here, the former constraint expresses the fact that there exists a component system which sends a message containing a notification about the failure event in the case of a failure. The latter constraint expresses the fact that the failure of a given component system can be masked by an component system with equivalent specification, which has performed until the failure event an execution equivalent to the one of the failed component system, and it continues its execution after the point of the first component system's failure.

Another way to strengthen the *masking* is to combine the constraints of *Diffuse* and *Active*. *Diffuse* constraint expresses the fact that a message sent to a member of a multicasting group will be eventually received by all members of that group. *Active* constraint expresses the semantics of the replication scheme. Thus, in the lower level of internal view model, *masking* can be further refined into four constraints that are *DetectionObj*, *MaskingObj*, *Diffuse* and *Active*.

4 Conclusions and Future Work

High confidence is important for embedded systems that are usually used in the fields where system failures will result in serious consequences. Systems of embedded systems have been used widely in recent years due to requirements for combining the individual embedded systems to accomplish the broad and common objectives. Typical examples of SoES include National Aviation Systems (planes, airports, airways, air traffic control), Naval Mine Countermeasures Force systems (search, sweep, neutralization systems), Naval Surface Fire Support systems (reconnaissance, targeting, weapon systems), and Theater Ballistic Missile Defense systems (surveillance, tracking, interceptor systems).

At present, several approaches have been presented to construct high-confidence embedded systems. These approaches include formal verification based on model checking, run-time testing and monitoring based on formal specification and the component-based composition based on behavior verification analysis from integrated behavior model and component behavior model. However, all these approaches were proposed for the monolithic embedded systems. It is necessary to make further efforts on constructing high-confidence systems of embedded systems. Thus, in this paper, we present a computational model which serves as a basis and start point for the development of high-confidence SoES.

Further research should be carried out to support practical application of the proposed model. In addition, methods for mapping other high-confidence emergent properties such as availability and safety into constraints imposed on component systems and interaction need to be further studied and more constraints related to high-confidence attributes should be developed.

References

1. Maier, M.W.: Architecting Principles for Systems-of-Systems. Technical Report. http://www.infoed.com/Open/PAPERS/systems.htm.
2. Wing, J.: Scenario Graph Generation and MDP-Based Analysis. Presentation at ARO Kickoff Meeting. University of Pennsylvania, Philadelphia, PA, May 24 - 25, 2001.
3. Garlan, D.: Model Checking Publish-Subscribe Software Architectures. Presentation at ARO Kickoff Meeting. University of Pennsylvania, Philadelphia, PA, May 24 - 25, 2001.
4. Clark, E.M., Emerson, E.A., Sistla, A.P.: Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specification. http://citeseer.nj.nec.com/clarke93verification.html.
5. Dwyer, M., Hatcliff, J., Avrunin, G.: Software Model Checking for Embedded Systems, www.cis.ksu.edu/~dwyer/projects/HCES-May-01-1.ppt.
6. Bjorner, N.S., Manna, Z., Sipma, H.B.: Deductive Verification of Real-time Systems Using SteP. Technical Report STAN-CS-TR-98-1616. Computer Science Department, Stanford University (1998).
7. Hong, H.S., I. Lee, Sokolsky, O., Cha, S.D.: Automatic Test Generation from Statecharts Using Model Checking. Proceedings International Workshop on Formal Approaches to Testing of Software. August 2001.
8. Clarke, D., Lee, I.: Automatic Test Generation for the Analysis of a Real-Time System: Case Study. Proceedings of 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97). June 1997.

9. Lee. I., Kannan S., Kim, M., Sokolsky, O., Viswanathan, M.: Runtime Assurance Based On Formal Specifications. Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications. Las Vegas, June 28-July1, 1999.
10. Wang, S., Shin, K.G.: An Architecture for Embedded Software Integration Using Reusable Components. Proceedings of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems. San Jose, CA., 2000.
11. Broy, M.: Specification and Modeling: An Academic Perspective. Proceedings of the 23rd International Conference on Software Engineering (ICSE'01). Toronto, Canada, May 12-19, 2001.
12. Luqi, Berzins, V., Yeh, R.: A Prototyping Language for Real-time Software, IEEE Transactions on Software Engineering. 14 (1988) 1409-1422.
13. Saridakis, T.: Robust Development of Dependable Software Systems. Technical Report. Institut National De Recherche en Informatique et en Automatique (INRIA) (1999).
14. Malek, M.: Dependability Concepts, Measures and Models. Technical Report. http://www.informatik.hu-berlin.de/rok/zs/zs2_1-4.pdf.

Software Evolution as the Key to Productivity

Oscar Nierstrasz

University of Bern, Switzerland
oscar.nierstrasz@acm.org
www.iam.unibe.ch/~oscar

Abstract. Despite the existence of a seemingly continuous stream of new technologies and methods, software productivity remains universally unimpressive. We argue that, as long as industry remains focused on short-term goals, and maintains a technology-centric view of software development, no progress will be made. A clear symptom of this problem is the fact that the metaphors we apply to software development are largely obsolete. Instead of thinking about software as we do about bridges, buildings or hardware components, we should encourage a view of software as a living and evolving entity that is developed and maintained by *people*. We begin with some assertions that are intended as food for thought. We continue by reviewing what we consider to be some of the key difficulties with software development today. We conclude with a few recommendations for research into software practices that take evolution into account.

1 Food for Thought ...

- Software "engineering" is only a metaphor.
- Software "architecture" is only a metaphor.
- Software "components" are only a metaphor.
- The most cost-intensive phase of any successful software project is "maintenance".
- What is termed "maintenance" consists in practice of continuous development and software evolution.
- Poor software quality is the greatest impediment to software evolution.
- Formal methods have a strictly limited impact on software quality in general.
- Constant refactoring is a prerequisite for effective software evolution.
- Aggressive testing is a prerequisite for refactoring.
- Standardized architectures and interfaces are a prerequisite to effective component reuse.
- A good software architect must be a good abstract thinker.
- Comparatively few programmers are good abstract thinkers.
- Software reuse can only have a limited impact on a small portion of the software lifecycle.
- Software reuse does not come for free.
- Object-oriented programming offers a means to model complex domains.

M. Wirsing et al. (Eds.): RISSEF 2002, LNCS 2941, pp. 274–282, 2004.

- It is hard to develop models that can be easily adapted over time.
- Objects are not components.
- Object-oriented designs expose a class hierarchy, not a run-time architecture.
- Raising the level of abstraction is the only way to produce more in the same amount of time.
- Scripting languages are the most effective rapid application development tools known today.
- Scripting languages are good for "programming in the small", not "programming in the large".
- Yesterday's large programs are today's small programs.
- Different people are motivated by very different things.
- Hardly anybody is motivated by money above all else.
- Technologists are often motivated by technology.
- Technology has only minimal impact on the success of a software project.
- Effective communication is the single greatest factor contributing to the success of software projects.
- Short and frequent iterations are a prerequisite for effective communication.
- You can lead a horse to water but you can't make it drink.

2 Software Evolution and Productivity

Despite the appearance of innumerable new software development techniques, tools and methods over the past couple of decades, there is a general perception that software productivity has not actually improved as a result of these innovations.

Why is this?

First of all, let us consider what we mean by *software productivity*. From an Engineering perspective, we usually consider that *productivity* = *units* ÷ *effort* [10]. Units of software can be notoriously difficult to measure, but, without belaboring the point, let us argue that

$$\text{productivity} = \frac{\text{functionality} \oplus \text{quality}}{\text{effort}}$$

where $\oplus$ is some kind of "addition" over functionality and quality. The desired functionality and quality are specified as Software Requirements, and the product is manifested in terms of Software Artefacts. Requirements being the input to our development process, productivity should just depend on the quality of the Requirements specifications, the methods and tools we use, and our own programming skill.

At this point, however, we must not forget that "Software Engineering" is just a metaphor, in particular, one that says that "software is like a physical product". Object-Oriented Programming, Component-Based Software Development (CBSD), and Software Architecture are other popular metaphors that suggest different ways of thinking about software. At the same time, however, metaphors can be dangerous if one forgets that they are *just* metaphors.

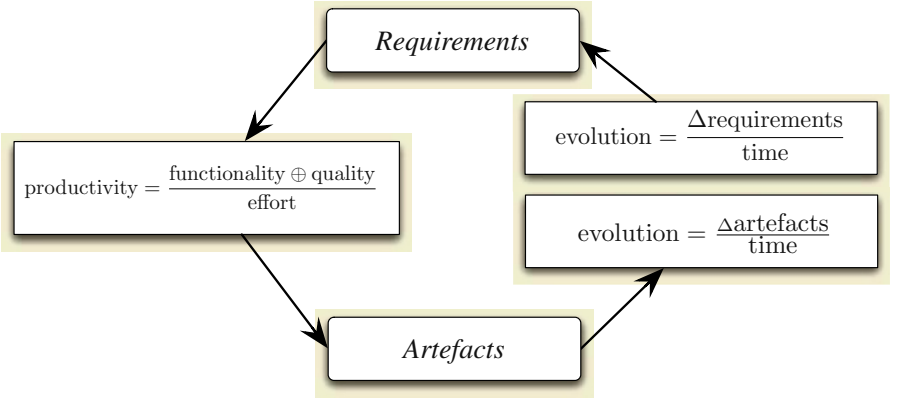


Fig. 1. Software evolution and productivity.

Software Engineering itself may well be one of these dangerous metaphors. There are many ways in which software is *not* like a typical Engineering product. For example, software is not subject to physical constraints. Many software application domains are also highly unstable due to the high rate of innovation. The most striking of these differences, however, are perhaps those highlighted by the following laws of software evolution [13]:

- *The Law of Continuing Change:* A program that is used in a real-world environment must change, or become progressively less useful in that environment.
- *The Law of Increasing Complexity:* As a program evolves, it becomes more complex, and extra resources are needed to preserve and simplify its structure.

This tells us that the Requirements are *not* the only input to our development process, but that legacy Artefacts also constitute an important input. Furthermore, as the Artefacts evolve, Requirements will also evolve in a never-ending cycle (see figure 1), and, as complexity increases, quality will degrade and productivity will decrease.

Oddly, most software development methods seem to assume that a new application is being developed rather than that some existing software base is being extended or modified, whereas in practice the latter is almost always the case. Most of the real problems with software development, in fact, have to do with software evolution: How can we construct software systems that can be gracefully adapted to changing requirements over time? If we consider most of the other perceived problems (poor quality, lack of reuse, and so on), they are largely subsumed by the problem of software evolution.

If we consider the typical lifecycle of a *successful* software product, we quickly see that most of the costs are associated with its life *after* deployment [3,14]. Furthermore, what is often misnamed "maintenance" actually consists mainly of

addition of new functionality, *i.e.*, continuous development, or simply *software evolution* [3,14]. This suggests that the impact of evolution of productivity cannot be merely incidental or occasional, but *fundamental* to software productivity. Attempts to improve productivity that do not take this into account are therefore doomed to failure!

If software evolution is really the key issue in developing successful software systems, why is it almost universally ignored in proposals of new methods and techniques for software development?

2.1 What's Wrong with OOP?

In the 1980s, object-oriented programming was widely considered to offer solutions to a wide range of software woes. In the 1990s, by the time that OOP became mainstream, it was clear that it was not a silver bullet. Worse, the learning curve with OOP is much steeper than with conventional procedural programming [15], reuse with OOP is much harder to achieve than with simple libraries of procedures, and all the problems with legacy applications recur with OOP except that they have an object-oriented flavour [9].

Implicit Architecture. Whereas procedural source code reflects procedural design quite well, object-oriented code does not normally reflect the run-time OO architecture. That means that it is typically much harder to read and understand a well-designed object-oriented program than it is to understand a well-designed procedural one because the source code exposes a class hierarchy, not the set of objects that provide the run-time behaviour. In order to understand the run-time behaviour, one needs to know which objects will be instantiated and how they relate to one another. Due to polymorphism, however, this information can be very hard to extract. This steep learning curve can make it hard to understand and evolve an object-oriented application.

Implicit Reuse Contracts. Although OOP offers very expressive mechanisms for software reuse, these mechanisms can be hard to understand and use correctly. An object-oriented *framework* consists of a class hierarchy that must be subclassed and extended to instantiate an application. Frameworks make use of many common idioms and design patterns, and the rules for correctly subclassing the framework classes typically entail *reuse contracts* [20] that may only be implicit in the code. Not only are OO frameworks hard to develop, but they entail a steep learning curve to use them.

Missing Sockets and Plugs. OO frameworks tend to be based on "white box reuse", *i.e.*, requiring knowledge of implementation details. Black box (component-based) reuse is more attractive since it makes the reuse contracts explicit as plugs (plug 'n play). Current OO analysis and design methods, however, encourage designers and developers to model domain objects in a way that leads to rich interfaces that are *not* plug compatible. This means that OOA/OOD as it is practiced today conflicts with the principles of CBSD.

Refactoring. Although it is well-established that the quality of OO software depends on a culture of refactoring and reengineering [9], it is still hardly standard practice.

Although OOP has demonstrated some benefits for productivity through reuse of libraries and frameworks, object-oriented methods alone do not especially address software evolution, so their impact on productivity is *necessarily* limited.

2.2 Are Components the Solution?

Although "object" is not yet a four-letter word, "component" seems to be the current buzz. But components, like objects, have also been around since the sixties [16].

Szyperski defines a software component as "a unit of independent deployment, a unit of third-party composition, [that] has no persistent state" [21]. Clearly this definition can fit many different kinds of software entity. Whether we call it a "component" or a library or a framework or an application generator or a 4G environment or a programming language does not really matter. In each case, the key idea is to factor out everything that is known and stable and put it into a box, thereby *raising the level of abstraction.* In each case, components may be composed by means of a graphical or textual specification that plugs together compatible parts.

The idea of building applications by "simply plugging together components" is very attractive, and certainly has some merit. But what is often forgotten is that components do not come from thin air. The cost of developing "reusable" components is significantly higher than that of building isolated applications [3,15]. Repositories of existing software elements help no more than do catalogues of the contents of junkyards (unless you are a junkyard artist). CBSD can only be successful when the process of developing components proceeds in parallel with the process of developing applications based on components, *and* the cost of developing components is amortized by the gains in productivity in developing and maintaining the resulting applications [11,17].

So what is the added value of CBSD? Certainly more rapid development is a gain in productivity since components will only have to be developed once. But the cost of developing and maintaining components can be higher than that of developing applications that are not component-based. Certainly higher software quality can be achieved by reusing components that have been thoroughly tested across a range of applications. But this presupposes a significant investment in ensuring the quality of the components, and presupposes a robust infrastructure for composing them. Only such a compositional infrastructure can enable "independent deployment".

If the biggest cost in the lifecycle of applications is evolution, we should ask ourselves if CBSD can help reduce maintenance costs. A component-based application clearly separates what is stable from what is not. Increased productivity during software evolution means that new functionality can be more easily integrated, and existing functionality can be more easily adapted. However this is

not achieved by components alone. We know and understand components, but what is hard to manage is *flexibility*.

So, although CBSD addresses software evolution to a greater degree than does OOP, mainly by facilitating certain kinds of change, it fails to address the hardest problems. The metaphor of "component-based software development" puts the word "component" in the pole position, but this is misleading. The real added value comes from how components are *composed*, so perhaps we should start to think more about *composition-based software development*.

2.3 What about Formal Methods? Testing? ...

Software quality pays off during *deployment* and *evolution*. Correctness, reliability, efficiency, usability, maintainability, portability, and other software qualities have an impact either on the cost of deployment or the cost of development and evolution of a piece of software. Costs, on the other hand, are entailed while *achieving* and in *maintaining* that quality. When is it worth paying for that quality? Whenever a piece of software is expected to live beyond an iteration or two of its lifecycle, the investment in its quality can be amortized over its future lifetime.

Formal methods clearly have their place in software development. However, many critical aspects of software quality are inherently impossible to formalize (for example, whether the user requirements have been adequately captured)[4]. Aside from certain well-understood domains, the cost of formally specifying requirements is not only exorbitant, but the cost of *proving* the correctness of an implementation may be unacceptably high. Furthermore, although there are some well-documented counter-examples [12], such as the application of model-checking tools, formal specifications and proofs typically do not scale well to large systems, and are rarely robust in the face of evolutionary changes. This suggests that formal methods most likely have their place in ensuring the robustness and correctness of individual, functional software components, but not necessarily assemblages of components, or non-functional aspects of software systems.

Testing has the clear disadvantage that it can never be used to demonstrate the *absence* of software defects [7]. Despite its obvious shortcomings, however, aggressive testing *during development and evolution* can have a dramatic improvement in software quality *and* software productivity [2]. Furthermore, tests, particularly automated unit tests, can be highly robust in the face of change. The investment in the development of test cases therefore rapidly pays itself off. Considering that software needs to be tested and debugged in any case, the investment in developing reusable test cases can be paid off even in a single iteration of the software lifecycle. It is therefore astonishing that industry largely continues to consider it to be a completely separate activity from development, and many developers consider it to be an unnecessary luxury.

There are innumerable other techniques and methods that may or may not have an impact on software productivity. Code reviews, coding standards, CASE tools, CRC cards, and so on, affect software quality in various ways, but each may or may not have a positive effect on productivity *in the long run*. We suggest

that any technique should be evaluated in the context of software evolution over a large number of iterations. A technique that is not cost-effective over time and robust in the face of changes cannot help software productivity in the long run, or can do so only in a very limited context.

2.4 Peopleware and Agile Processes

Methods, tools and processes all have their place, but it is important to recall that (i) productivity of software developers varies enormously, independently of tools or techniques applied, and (ii) the most important factor contributing to the success of a software project is typically the motivation of the team. Models and standards like CMM [18] and ISO 9000 [19], can be useful to assess the maturity of the process within an organization, but this need not bear any relation to the *productivity* of its development teams. Teams with immature processes may be highly productive, and organizations with mature processes may be moribund. Optimizing processes with negative and positive feedback are surely a Good Thing, but this is not necessarily the key to higher productivity.

In the Silver Anniversary edition of *The Psychology of Computer Programming*, Weinberg notes:

> In the first edition to this book, I predicted, "... attention to the subject of personality should make substantial contributions to increased programmer performance — whether that attention is paid by a psychological researcher, a manager, or the programmer himself" (p. 158). Even though I knew next to nothing about personality at the time, this turned out to be one of my most successful predictions. [22]

Since that time, numerous authors have pointed out that technology has only a limited effect on software productivity [3,6,8,11]. Not only can there be a huge difference in productivity between individual developers (variously claimed to be 10:1 or even 50:1), but factors such as team communication and motivation have repeatedly been shown to far outweigh any technological factor in the success of projects. Alistair Cockburn, for example, reflecting on 20 years of experience with various projects notes that:

- Almost any methodology can be made to work on some project.
- Any methodology can manage to fail on some project.
- Heavy processes can be successful.
- Light processes are more often successful, and more importantly, the people on those projects credit the success to the lightness of the methodology. [5]

DeMarco and Lister give many reasons for these phenomena in their book *Peopleware*, which can be summed up as:

> The major problems of our work are not so much *technological* as *sociological* in nature. [8]

If we can accept the idea that there can be no purely *technological* silver bullet [3], then we must conclude that our only hope is to pay more attention to sociological issues in the software process. The "Manifesto for Agile Software Development" adopts this point of view by proposing that we should value:

> *Individuals and interactions* over processes and tools
> *Working software* over comprehensive documentation
> *Customer collaboration* over contract negotiation
> *Responding to change* over following a plan [1]

Although there appears to be no claim in this manifesto that agile processes will improve productivity in the long run, it is clear that, *as a metaphor*, agile software development gives change a central role, and downplays purely technological tactics. This, we feel, is an important step in the right direction.

3 Research

We have argued that *software evolution* is the most important factor to influence productivity in any software development project. This leads us to the following observations:

1. *Software evolution* is unavoidable, both before and after deployment. Ignoring its influence will kill productivity.
2. *Raising the level of abstraction* is the only way to produce more in the same amount of time.
3. *Agile processes* take into account that software is a living thing, whose life source comes from the interactions of the people who use it and develop it.

We conclude that further research is urgently needed in the following areas:

- Refactoring, reengineering and round-trip engineering,
- Migrations towards component frameworks and software product lines,
- Testing strategies to support continuous development,
- Agile processes,
- Composition languages and infrastructures,
- Architecture-driven software development.

Weinberg's *Psychology of Computer Programming* was written over 30 years ago, but we have been slow to pick up on its lessons. Cockburn puts his case well:

> His characterizations and recommendations, based upon project interviews in the 1960's, are still accurate and significant 30 years later. That validates the stability and importance of these sorts of issues. It is about time we studied these issues as a core to the field of Software Engineering and stopped rediscovering their importance every 30 years [5].

Acknowledgments

Thanks to Roel Wuyts. Markus Gaelli and the anonymous reviewers for various suggestions and corrections.

References

1. Manifesto for agile software development. `http://agilemanifesto.org`.
2. Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000.
3. Frederick P. Brooks. *The Mythical Man-Month*. Addison Wesley, Reading, Mass., 1975.
4. Frederick P. Brooks. No silver bullet. *IEEE Computer*, 20(4):10–19, April 1987.
5. Alistair Cockburn. Characterizing people as non-linear, first-order components in software development. In *4th International Multiconference on Systemics, Cybernetics, and Informatics*, Orlando, FL, 1999.
6. Alistair Cockburn. *Agile Software Development*. Addison Wesley, 2002.
7. Ole-Johan Dahl, Edsgar W. Dijkstra, and C.A.R. Hoare. *Structured Programming*. Academic Press, 1972.
8. Tom DeMarco and Timothy Lister. *Peopleware, Productive Projects and Teams*. Dorset House, 2nd edition, 1999.
9. Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.
10. Norman Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1996.
11. Adele Goldberg and Kenneth S. Rubin. *Succeeding With Objects: Decision Frameworks for Project Management*. Addison Wesley, Reading, Mass., 1995.
12. Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, Sept. 1990.
13. M. M. Lehman and L. Belady. *Program Evolution – Processes of Software Change*. London Academic Press, 1985.
14. Bennet P. Lientz and E. Burton Swanson. *Software Maintenance Management*. Addison Wesley, 1980.
15. Tom Love. *Object Lessons – Lessons Learned in Object-Oriented Development Projects*. SIGS Books, New York, 1993.
16. M. Douglas McIlroy. Mass produced software components. In P. Naur and B. Randell, editors, *Software Engineering*, pages 138–150. NATO Science Committee, January 1969.
17. Oscar Nierstrasz and Dennis Tsichritzis, editors. *Object-Oriented Software Composition*. Prentice-Hall, 1995.
18. Mark C. Paulk, Charles V. Weber, Bill Curtis, and Mary Beth Chrissis, editors. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison Wesley, 1994.
19. Charles H. Schmauch. *ISO 9000 for Software Developers*. ASQC Quality Press, 1995.
20. Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proceedings of OOPSLA '96 Conference*, pages 268–285. ACM Press, 1996.
21. Clemens A. Szyperski. *Component Software*. Addison Wesley, 1998.
22. Gerald M. Weinberg. *The Psychology of Computer Programming*. Dorset House, silver anniversary edition edition, 1998.

Model-Checking Complex Software – A Memory Perspective*

Murali Rangarajan and Darren Cofer

Honeywell Laboratories
{murali.rangarajan,darren.cofer}@honeywell.com

Abstract. In recent times, there has been growing interest in model checking software systems. Such efforts bring into focus the memory constraints of model checking approaches. In this paper, we present our results from the analysis (at the source code level) of a real-time operating system using the Spin model checker and explain our efforts to understand the reasons for the extremely large state space. Our studies indicate that even hand-optimized models suffer from memory constraints, thereby indicating the need for other approaches that break the problem into smaller pieces.

1 Introduction

Model-checking is attractive for formal analysis of systems due to the automatic nature of their proof processes and also due to their ability to provide counter-examples. Model checking has long been used for verifying protocols and algorithms at an abstract level. Unfortunately, for many systems, the chief design artifact is the source code, and establishing the correctness of that source code is critical. In the past, verification of source code had been considered impractical due to the extremely large memory requirements. Recently, with the availability of cheap memory and identification of various optimization techniques, there has been growing interest in verifying software systems at the source-code level ([5] and [13]).

Under a cooperative research agreement with NASA's Langley Research Center, Honeywell is developing and applying formal techniques to verify safety-critical properties in Integrated Modular Avionics (IMA) components such as the Deos™ real-time operating system. We believe that the use of formal techniques to increase avionics software design assurance will be crucial to aviation safety over the next decade.

This work and the approach taken began with an earlier project undertaken with the Automated Software Engineering group at NASA Ames [11]. The model checker Spin [7] was used to model and verify portions of the original version of the Deos scheduler. The model was derived directly from the actual flight code so that the analysis results would be closely connected to the real system.

* This material is based upon work supported in part by NASA under cooperative agreement NCC-1-399.

M. Wirsing et al. (Eds.): RISSEF 2002, LNCS 2941, pp. 283–296, 2004.

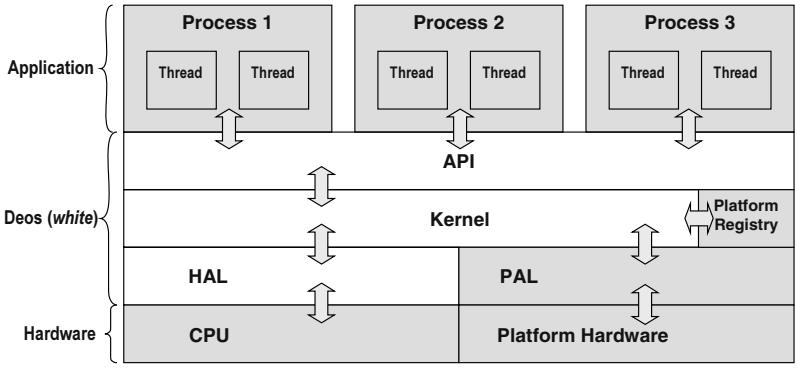


Fig. 1. Deos components and terminology.

Our current work focuses on the modeling and analysis of advanced features of the latest version of Deos, such as slack scheduling and aperiodic interrupt servicing. These new features add considerably to the complexity of the software. Our analyses indicate that, in spite of all the optimizations, the memory requirements are so great that exhaustive verification of even specific configurations of the system cannot be performed using 4GB of memory (the maximum amount of memory that can be used by Spin).

This paper is organized as follows. In the next section, we present an overview of the Deos™ operating system and the features of the operating system that have been incorporated in our model. The sections following this present a brief overview of our modeling approach and the various optimizations incorporated in our model. In section 5, we present the configurations for which the results are provided in section 6. Following this, the current and future directions for our work are presented. The final section discusses the general context within which this particular work is being conducted.

2 The Deos™ RTOS

The Deos real-time operating system [1] was developed by Honeywell for use in the Primus Epic avionics suite for business, regional, and commuter jet aircraft. Deos hosts many safety-critical applications in these aircraft, including primary flight controls, autopilots, and displays.

Deos is a microkernel-based real-time operating system that supports flexible Integrated Modular Avionics applications by providing both space partitioning at the process level and time partitioning at the thread level. Space partitioning ensures that no process can modify the memory of another process without authorization, while time partitioning ensures that a thread's access to its CPU time budget cannot be impaired by the actions of any other thread.

The main components of a Deos-based system are illustrated in ***Figure 1***. A given software application consists of one or more processes. Each process is executed as one or more threads. All threads in a process share the same virtual address space in

memory. Each hardware platform in the system has a separate instance of the Deos kernel running on it. The kernel communicates with its underlying hardware via its hardware abstraction layer (HAL) and platform abstraction layer (PAL) interfaces. The HAL provides access to the CPU and its registers and is considered part of Deos itself. The PAL provides access to other platform hardware, such as I/O devices and interrupt signals. The application threads interact with the kernel and obtain the services it provides by means of a set of functions called the application program interface (API).

Basic Scheduler Operation

Deos must ensure that every application gets its allotted amount of CPU time every period. This provision is called *time partitioning*, and is accomplished by the Deos scheduler. At system startup, each process is given a fraction of the total available CPU resource, called the process's CPU budget, for its use. The process then allocates its CPU budget to its threads. Deos ensures that each of the threads has access to its allocated CPU budget every period.

There are three types of threads managed by Deos: the idle thread, the main thread, and user threads. The idle thread does nothing and has the lowest priority in the system. It is executed whenever there is nothing else for the scheduler to run. Each Deos process has a main thread that is automatically created at startup and manages resources allocated to the process. The main thread may create any number of user threads to perform the work to be done by the application process. Resources for user threads (such as their time budgets) are provided by the main thread. User threads may be static and created at system startup, or dynamically created at run time. Both main and user threads may delete themselves. The state diagram for running threads is shown in ***Figure 2***.

The Deos scheduler enforces time partitioning using a Rate Monotonic Scheduling (RMS) policy. RMS assigns thread priorities so that shorter period (high rate) threads are assigned a higher priority than long period (low rate) threads. Using this policy, threads run periodically at specified rates and they are given per-period CPU time budgets, which are constrained at thread creation so that the system cannot be over-utilized.

Slack Recovery and Scheduling

The restriction of applications to the use of a fixed CPU time budget in each period is often suboptimal. Many applications (for example, network service applications) have highly variable demands on the CPU and have a desired level of performance considerably greater than the minimum required. Giving these applications only the minimum budget necessary will result in low performance; giving them a high enough budget to ensure the performance desired will result in severe underutilization of the CPU most of the time and may "crowd out" other applications that users want to have in the system.

For this reason, the Deos kernel provides a mechanism for "slack scheduling" which assigns system "slack" time to threads on a first-come first-served basis. The classical view of slack scheduling is given in [8]. The version implemented in Deos incorporates several major modifications of this view necessitated by the special features of Deos (e.g. the capability to dynamically activate and deactivate threads, and the existence of aperiodic threads).

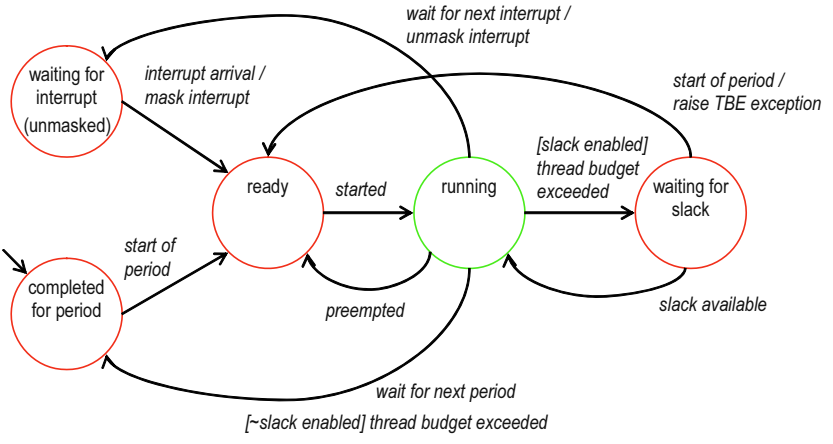


Fig. 2. Thread state diagram with slack and interrupts.

Servicing Aperiodic Interrupts

Deos permits applications to respond to hardware interrupts via the activation of an interrupt service routine (ISR) thread. An ISR thread has a period and budget associated with it, and like any other thread it is guaranteed access to its CPU budget in each of its periods. An ISR thread is activated by the arrival of its associated interrupt. Since the timing of the arrival of the interrupt can not be (in general) predicted, Deos does not guarantee that the ISR thread will complete in the same period that the interrupt arrived. This can happen, for example, if an interrupt arrives near the end of the thread's period when the time remaining is less than the thread's budget.

To prevent an unbounded overloading of the system by bursts of an interrupt, an ISR thread has a period and a budget just like other periodic threads. Deos guarantees that the ISR will not consume more than its allocated CPU budget in each period, but Deos does not restrict the rate or number of interrupts that arrive so long as the ISR thread's CPU budget is sufficient. System overhead (e.g., context switch time and cache effects) associated with preempting a running thread are all charged against the ISR thread's budget.

Overhead Accounting

The basic Rate Monotonic Scheduling (RMS) theory used in Deos implicitly assumes that all of the time that the CPU executes is actually spent on user mode thread execution; that is, it is assumed that context switches take zero time, and that there are no other system processing overheads that take time away from user thread execution. In the real world this is not the case. Deos performs a variety of scheduling and other activities that take time to execute. In order to preserve the assumptions of RMS, the time taken to perform any Deos kernel operation must be accounted for in one of two ways:

1. By subtraction from the total CPU time available to executing threads.
2. By charging the time to the budget of the thread running when the operation occurs.

When the kernel does work directly on behalf of a thread, the time is charged to that thread and deducted from its budget. Work that cannot be identified with a single thread (such as system tick interrupt handling) is deducted from the system timeline.

Correctly accounting for overhead times for all of the functions that the scheduler must perform is critical to maintaining time partitioning guarantees. Kernel overhead time is broken down into a number of components. These times can vary for different hardware platforms. Platform integrators must determine upper bounds on these values for their particular platform configurations. This can be done using an instrumented version of the Deos kernel that provides data on the lengths of critical sections and the other overhead parameters.

3 Project Requirements and Modeling Approach

The key goal of this project is to verify that the time-partitioning property of Deos™ holds in the presence of its many features (such as slack reclamation and interrupts). In addition, the pre-conditions to many of the functions in the C++ code (in the form of comments) must be verified to hold under all circumstances. Finally, automation of the translation is important.

Several approaches were considered for this project. Symbolic model checking using SMV[14] was rejected due to the large differences between the model structure and the source code structure. The approach for modeling sequential code in SMV is to model the code as a state transition system with a program-counter-like field to identify the enabled state (or line of source code) at any given time (for example, see [4]). This approach leads to very large model sizes and low correspondence between the model structure and the source code structure, thereby making the process of verifying model correctness difficult. Approaches based on Bounded Model Checking (BMC)[2] were rejected as they did not provide the necessary degree of correctness guarantee. Any conclusions drawn from the results hold only up to the depth specified during the verification. And increasing the verification depth to include the maximum model depth would result in BMC encountering the same memory problems as other verification techniques. Since the idea was to obtain as comprehensive a guarantee of correctness as possible, BMC-based approaches were rejected. The Spin-based approach was selected due to the high degree of correspondence between the model and the source code, the support for assertions (which have lower memory requirements than use of temporal-logic properties) and its strong usage history.

Our translation approach is based on the techniques developed by researchers at NASA Ames. The Deos kernel is written in an object-oriented style using C++. In contrast, Promela, the input language of the Spin model checker, is a process-based imperative style language that uses shared memory and message passing for communication. In order to model objects within the framework of processes, and to make verification tractable, four basic techniques are used.

- Use data type abstractions for variables such as counters. Integers are modeled as bytes or even booleans, where possible.
- Model C++ objects as arrays and object pointers as indices into the arrays.
- Convert methods into inline macros and replace recursion with distinct function calls.
- Flatten the inheritance hierarchy.

The details of these translations are described in [11].

The model consists of three parts – the kernel, user threads and the environment (see ***Figure 3***). The kernel corresponds to the code translated from C++. It provides services to the user threads and interfaces with the hardware environment. It is the most complex part of the model. The user threads are very simple and just invoke various calls to the kernel and responds to messages (such as preemption) from the kernel. The environment provides the hardware services such as timers and interrupts. Though the environment is not as complex as the kernel, it is more difficult to model realistic hardware behavior in pure software. In our tests, we have found an event-based simulation-like approach to be the most suitable for modeling the environment.

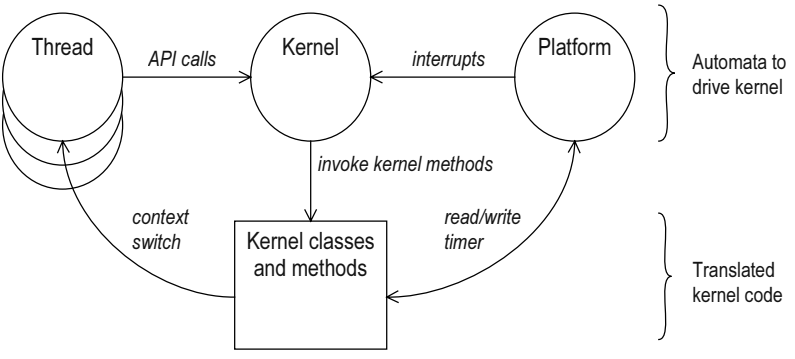


Fig. 3. Modeling Deos in Spin.

4 Model Optimizations

Model checking in general suffers from memory constraints. Therefore, appropriate optimizations are critical. Though the modeling was performed with automatic translation in mind, the fact that it was hand translated enabled us to incorporate many optimizations in the model. Memory optimizations fall into two categories – those that reduce the size of the state vector, and those that reduce the total number of states in the system.

Reducing the size of the state vector involved the replacing of all `integer` data types by the `byte` data type, and by actively tracking the variables that must be stored as part of the state. Promela has a `hidden` data type qualifier that excludes variables from the state vector. Judicious use of hidden variables results in models that make effective use of memory.

The total number of states was reduced by the use of predicate abstractions. For example, it was noticed that the period counter variable was used to track whether individual threads had run at least once in the current period. By replacing the counter with a Boolean predicate, we were able to collapse states that differ only in the value of the period counter into a single state.

5 Configurations Analyzed

Results are provided in the next section for the following selective models that have been verified correct using Spin. Each version represents an incremental improvement in the model or a bug fix from the previous version. A comparative table of the performance of each version with respect to several different parameters follows in Section 6.

1. *Baseline – no slack threads, no ISR thread*: This model consisted of our basic version with the latest changes to incorporate the overhead computations in the model. The model consists of the scheduler, the timer, a main thread, two user threads and the idle thread. None of the threads were slack consumers or ISRs. The overhead parameters were modeled at a high level of abstraction. An exhaustive search of the state space for assertion violations was performed for this configuration, with none found.
2. *No slack threads, no ISR thread, negative values for timer*: During the analysis of the baseline model, we discovered that in the real system the time remaining on the timer can become negative, whereas we had modeled time as an unsigned byte. The remaining time of the various thread budgets can also be negative. This capability was added to the model, and it was analyzed again. With this change, the number of states increased dramatically, making exhaustive verification impractical. Therefore, we had to resort to bitspace (supertrace) verification [6] from this stage onwards. Typical coverage obtained using this technique was > 98%.
3. *Slack threads, no ISR thread, higher resolution for overhead variables*: Once we had a basic configuration working well, the next task was to add slack threads. When the user threads in the configuration were made slack consumers a number of assertion violations occurred. Understanding the reasons for the assertion violations necessitated a better understanding of the relationships between the various system constants. Towards this end, the overhead variables were modeled at a higher degree of resolution. Each overhead variable was split into the constituent system constants as described in the Deos Design Document [1]. The longestCriticalSection (LCS) was given a value of 1. All other system constants were given a value of 0.
4. *Slack threads, no ISR thread, context switch time*: Up to this point the time to perform the context switch in hardware was assumed to be zero. In reality, context switches consume a finite amount of time. In order to reflect this fact, the time to perform a context switch was set to consume a non-zero amount of time (1 unit). Care was taken to maintain the ratios between the values of the various base system constants approximately consistent with realistic values encountered in the Deos Registry. Each time there is a change in the running thread, system time was advanced by the context switch time.
5. *Slack threads, one ISR thread, context switch time*: Once the basic configuration was found to work well in the presence of slack consumers, aperiodic interrupts were introduced. This involved the designation of one of the threads as an ISR thread, and enabling the generation of interrupts by the environment.

6 Performance Results

In this section, we first present an overview of the verification results for selected stages in the evolution of the current model, starting with the initial version to include overhead accounting. It can be easily seen from the results that the memory requirements increase exponentially with addition of new features to the model. We then present an analysis of the results, intended to identify the reasons for the large increase in the memory requirements.

In *Table 1* we present the results obtained for each version of the model discussed in the previous section. E/S indicates the verification technique (Exhaustive or Supertrace) used. Except for the initial model where we used an exhaustive verification, all the other versions of the model were verified using Spin's Supertrace verification technique [6]. The state vector size for all these models was 444 bytes. *Depth* represents the longest single path from the initial state until a previously visited state is reached. *States Stored* represents the number of distinct states maintained by the verifier. *Time* is the total system time for the verification run to complete. *Actual Memory* is the total memory used by the application to store the state information. *Equivalent Memory* is Spin's estimate of how much memory would be required to store all the states of the system if an exhaustive verification (with out any compression or optimization) was used.

With the addition of new scheduling features and overhead accounting the state space of our model has grown substantially. In addition, the longest traces found in the model (the reported depth) are far to long to examine manually. As a result, it has become necessary to introduce some approximations in the analysis and consider more carefully the structure of the model.

Before adding the interrupts and overhead computations, the preferred method of analysis was by *exhaustive* verification in Spin. This technique keeps track of the complete state information for all the reached states while doing the analysis and gives the most accurate results regarding the correctness of the system. When successful, it provides 100% coverage of the state space. When it runs out of memory there are no guarantees about the correctness of the model. Given Spin's current memory addressing limit of 4 GB (32 bits) and the size of the model's state vector, it works very well for models with close to a million states. Once interrupts and overhead accounting were added to the model the state space quickly became too large to be analyzed by exhaustive verification.

The alternative technique we had at our disposal was *bit state* or *supertrace* verification [6]. The Supertrace verification technique of SPIN uses two independent hashing functions and stores only the results of the hashing functions for each state visited, marking the corresponding bits in a large table. If both hash functions yield a previously marked bit then that state is considered to have been previously visited (a cycle). All the states (after the first) that result in such collisions are dropped from the analysis, and all subsequent states that could have been reached from the dropped states are not visited *from the dropped state*. This is less of a problem if the state space is highly connected as the unexplored states may be reached from other system states.

The next issue was the *coverage* we were obtaining in the analysis. SPIN gives an estimate of the coverage it obtained when using the Supertrace approach. Since an exhaustive search is not being performed, it is not possible to compute precise values

for coverage obtained. However, SPIN has a mechanism to generate reasonably accurate approximation (called the *hash factor*) of the coverage that is based on the total memory available for the hash array and the number of distinct states stored in that array. The hash factor is directly proportional to the memory used by the hash array, and is inversely proportional to the number of states stored in the array. Until the addition of overhead computations we typically obtained estimated coverage (based on Hash Factor) of over 98%. Although it cannot be said with certainty that the system is bug-free based on 98% coverage, based our experience so far with the model, we believe that the probability of bugs being present is extremely low.

Table 1. Performance results for verification of models in Section 5.

Model #	E/S	Depth	States Stored (e+06)	Time (H:M:S)	Actual Memory (MB)	Equivalent Memory (MB)
1	E	47,279	3.88	57:26:37	335.492	1786.149
2	S	193,780	43.00	0:49:00	287.351	19448.510
3	S	338,155	51.40	1:35:54	331.911	23011.565
4	S	452,299	89.80	2:02:10	1678.881	40213.067
5	S	2,292,597	212.00	4:59:18	1716.668	95192.269

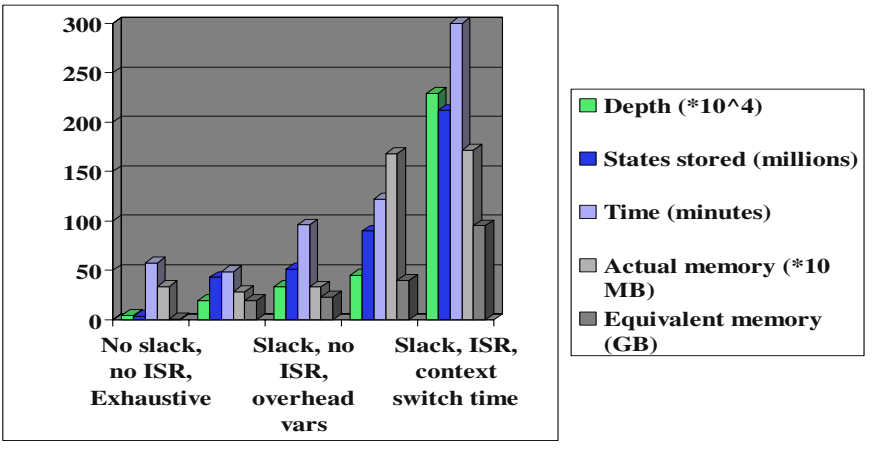


Fig. 4. Chart depicting the above results.

With the addition of overhead computations, the estimated coverage obtained dropped to just above 90%. In our attempts to improve the hash factor (and thereby the coverage obtained), it seemed reasonable to increase the memory used by the hash array. Since the hash factor is directly proportional to the size of the hash array, increasing that should lead to higher values for the hash factor. But in reality we found that increasing the size of the hash array did not lead to a corresponding increase in the hash factor. This was because when the hash array increased in size the number of states stored by SPIN increased proportionally, leading to the ratio being the same. Our conclusion was that the hash factor approximation for the coverage obtained works as expected only if the memory available for the hash array is sufficiently close (the

actual number of states is between one-half and one times the number of bits in the hash array) to the memory that would be needed to store all the states in the system.

One interesting behavior of our model is that the *depth* (the longest sequence of states visited before a previously visited state is reached) is very high, especially with the later versions of the model. Though the complexity of the model could be a factor, it was not clear that the increased complexity alone could account for the very large increases in the depth. One possibility was that the depth was so high because of the sequential nature of all the different behaviors displayed by the model (see ***Figure 5***). In order to validate the theory, we instrumented the model to simulate the behavior of Breadth-First Search. If the number of states reached by BFS were the same as that of the Depth-First Search technique employed by SPIN, it would validate the hypothesis. We found that the number of states reached by using BFS was within 4.5% to 5% of the number of states reached by DFS. We believe that the difference in the number of states reached was due to the noise introduced by the instrumentation. Therefore, our hypothesis regarding the reason for the large depths appeared to be correct.

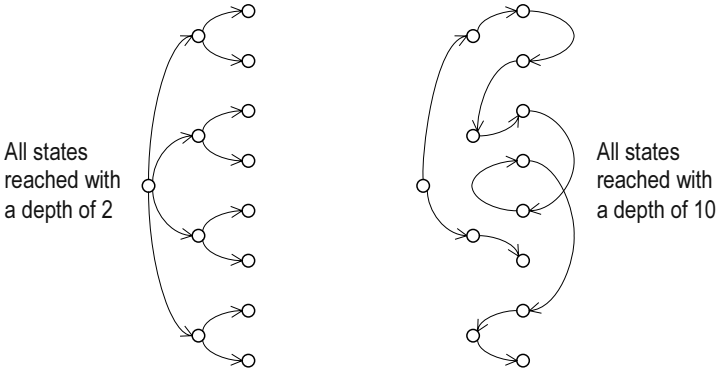


Fig. 5. Breadth-first vs. depth-first search of model.

However, the operation of the supertrace verification mechanism casts some doubt on this conclusion. If the number of states stored is purely a function of the available memory for the hash array when the number of states is extremely large, then either DFS or BFS would give the same approximate value for the states stored. Therefore, in order to really answer the question, we need a model for which all the states can be stored in the hash array. To test this conclusion we reverted to a simpler version of the model.

We performed exhaustive verification on a model without slack, interrupt threads or overhead accounting. We instrumented the model to perform a bounded depth-first search (BDFS), which was our approximation for breadth-first search. The bounds were set to coincide with the end of the longest periods. The results of our tests are shown in ***Table 2***. The first column gives the number of longest periods that formed the bound for BDFS. It should not, in theory, have any effect on the results for DFS. In practice, however, we found a very slight decrease in the depth with increase in the number of longest periods in the bound. The reason for the decrease is not clear to us, but we believe that the decrease in depth is too small to have any effect on our conclusions. The second column presents the results for DFS, and the third column for BDFS. For each search method the number of distinct states visited and the maximum

depth attained are listed. The results indicate that there is at least one longest path in the model whose states cannot be reached by performing BDFS. This implies that traversing the full depth of the model is necessary to visit all the states in the model, which puts our hypothesis in question. The original issue of why the depth should be so large has not been solved yet.

We are continuing to look at other ways to obtain higher coverage. We have been working with Holzmann (developer of SPIN) to increase the amount of memory that can be used in a supertrace verification. We have also been working on compiling SPIN for a 64-bit architecture. (The current version assumes a 32-bit architecture, hence the 4GB memory limit).

Table 2. Results in our analysis of DFS vs. BDFS.

Max depth for BDFS	Results using DFS		Results using BDFS	
	States	depth	states	Depth
1	1.29524 e+06	38240	1.04135 e+06	7337
2	1.29524 e+06	38239	1.04211 e+06	10008
19	1.29524 e+06	38237	1.26304 e+06	28662
29	1.29524 e+06	38236	1.29524 e+06	38236

7 Current Research

Our main goal now is to identify new approaches to increasing the coverage obtained when using exhaustive verification. Simple approaches such as re-compiling Spin on a 64-bit architecture (thereby making more memory available to the process) are in reality not so simple. The core computations in Spin assume a 32-bit architecture, and changes to the core computations to reflect 64-bit architectures must be followed by an evaluation of the "correctness" of the tool. Since such an evaluation is difficult, the ideal solution must leave the core computations unchanged. Also, since we only use assertions to represent our correctness criteria, it is sufficient for our solution to explore the state space and not handle LTL properties. Our leading contender for the solution is the use of secondary memory.

The idea is to minimize the changes to the Spin code while enabling the use of more memory than is possible with 32-bit addressing. Towards this end, our solution is to run Spin as before, but at the point it runs out of memory, instead of reporting an error and exiting, to dump the set of partially explored and fully explored states on to secondary memory. The next run would take a partially explored state and proceed with the verification until all reachable states are explored or until it too runs out of memory. At this point, the states in memory are reconciled with the states stored previously on the secondary memory. A new partially explored state is chosen next, and the verification run is repeated. The runs are repeated until there are no more partially explored states. The main drawback of this approach is the repeated exploration of states, since information about states explored in previous verification runs are "forgotten" after it is stored on the secondary memory. We believe this loss can be offset by distributing the problem over multiple processes.

We are currently using PSPIN [9] as the starting point for our work. PSPIN is a distributed implementation of Spin. It provides a wrapper around Spin that provides

the distributed functionality. PSPIN supports many different schemes for partitioning the state space among the various processors. Our studies with using PSPIN on a Deos model indicates that the partition functions are inadequate. In most cases, the load distribution was highly skewed, with some processors doing nothing while some processors doing a majority of the work. With Partial Order Reduction enabled, this skewed distribution resulted in some processors running PSPIN requiring more memory than the single-processor Spin implementation, thereby defeating the whole purpose of distribution[1]. In the one case (Global Hash partitioning) where the memory was perfectly balanced between all the processors, the memory and time requirements were very high. We are in the process of identifying an appropriate hash function to achieve a good balance between processor memory load, and memory and time requirements. One of the techniques we are exploring is dynamic load balancing among the processors, while attempting to keep the overhead low.

8 Conclusions and the Big Picture

Model checking provides an attractive approach to formal analysis of complex systems. But there are many hurdles in achieving the effective use of model checking for analyzing software. The most basic constraint is the scalability of the analysis tools. Memory constraints have always been a major impediment to the scalability of model checking tools. In this paper, we have described our efforts to alleviate this problem.

The second issue concerns the kinds of software that can be model-checked. In the case of the Deos™ operating system, an arbitrary number of threads and processes may be active at any given time, with dynamic creation and deletion of threads and processes. It is impossible to verify all such configurations using only model checking. In general, a number of (provably correct) optimizations such as datatype abstractions and predicate abstractions are used to restrict the model size. In the case of the Deos™ model, further restrictions on the number of threads and processes are also necessary. These restrictions on the model must be shown to not affect the validity of the model checking results. Therefore, it is necessary to prove that the restricted model displays the same behaviors (including buggy behaviors) as that of the unrestricted model. Towards this end, we have created a simple model of the core Deos scheduler in PVS [12], a theorem prover capable of handling arbitrary number of threads, periods and processes. We are using this model in attempting to prove the validity of model checking results over just three threads, three periods and one process.

The third issue is the actual creation of the model. The model must be created automatically for this approach to be of practical use, and we are in the process of writing a translator from C++ to Promela to address this issue. One luxury available to programmers in C++ that is not available to modelers in Promela is the use of temporary variables. Since C++ supports true functions, programmers can declare temporary variables as needed. When translating these temporary variables into Promela,

[1] Partial Order Reduction (POR) has a large impact on the single-processor verification. Its effect decreases as the degree of parallelism increases. When determining the effectiveness of distributed implementations, it is important to enable POR in order to obtain meaningful results.

our initial approach was to declare them as *hidden* variables. We then realized that, since these hidden variables are not part of the state vector, this approach gave rise to unexpected and erroneous behavior from the verifier. But declaring all the different variables as local to a process or as global variables would result in a tremendous increase in the size of the state vector, thereby resulting in large increases to memory requirements. We are currently building a tool that would take in a translated Promela model with temporary variables declared as hidden variables, and would optimize the model to reuse a set of variables as much as possible. The tool would also insert *atomic* and *d_step* constructs appropriately so as to maximize the sections of the model that would be enclosed within these constructs. This optimization works for Deos because only one part of the model can be active at any given time. These two optimizations together are expected to reduce the memory requirements for the verification considerably.

One final issue is the generation of an environment. The system being modeled may not work in isolation – it may provide services to other systems, or may request services from other systems. For example, Deos receives timing services (such as timers and periodic tick interrupts) from the underlying hardware, while it provides the operating system services to applications running on top of the operating system. Verification of the operating system requires accurate modeling of the environment behavior. Since the environment is not actually part of the system being verified, and since its behavior must be deduced from the expectations the system being modeled has regarding its environment, automated generation of the environment presents considerable challenges. As a starting point for solving this issue, we are considering means to generate stubs for the various functions that are called from within the operating system. Also, since applications running on top of Deos trap into the kernel, we can generic applications that can trap into any kernel call without any restrictions. Then, based on the error traces, we can restrict the behavior of this generic environment to enable only the valid behaviors.

References

1. "Design Description Document for the Digital Engine Operating System," Honeywell specification no. PS7022409.
2. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolid Model Checking without BDDs." Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99), LNCS, Springer-Verlag, 1999.
3. Binns, Pam. "A robust high-performance time partitioning algorithm: the Digital Engine Operating System (DEOS) approach." In *20th Digital Avionics System Conference Proceedings*, October 2001.
4. George S. Avrunin, James C. Corbett, Matthew B. Dwyer, Corina S. Pasareanu, and Stephen F. Siegel, "Comparing finite-state verification techniques for concurrent software," Technical Report UM-CS-1999-069, Department of Computer Science, University of Massachusetts, November 1999.
5. Havelund, K. and Pressburger, T. "Model Checking Java Programs Using Java PathFinder", *International Journal on Software Tools for Technology Transfer* (STTT) 2(4), April 2000.
6. Holzmann, G. "An analysis of bitstate hashing," *Formal Methods in Systems Design*, Nov. 1998.

7. Holzmann, G. "The model checker Spin." *IEEE Transactions on Software Engineering*, vol 23, num 5, pp. 279–295, 1997.
8. Lehoczky, J. P. and S. Ramos-Thuel. "An optimal algorithm for scheduling aperiodic tasks in fixed-priority preemptive systems." *IEEE Real-Time Systems Symposium*, December 1992.
9. Lerda, Flavio, and Sisto, Riccardo, "Distributed Memory Model Checking with SPIN." *5th International SPIN Workshop on Theoretical Aspects of Model Checking,* July, 1999.
10. Liu, C. L. and J.W.Leyland, "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment." *Journal of the ACM* 20(1), January 1973, pp. 46-61.
11. Penix, J., W. Visser, E. Engstrom, A. Larson, and N. Weininger, "Translation and Verification of the Deos Scheduling Kernel." Technical report, NASA Ames Research Center/Honeywell Technology Center, October 1999.
12. S. Owre, J. M. Rushby, and N. Shankar, "PVS: A Prototype Verification System." *11th Conference on Automated Deduction,* Saratoga, NY, June, 1992.
13. The Bandera Project, *http://www.cis.ksu.edu/santos/bandera/*
14. The SMV system, *http://www-2.cs.cmu.edu/~modelcheck/smv.html*

Agile Modeling with the UML

Bernhard Rumpe

Software & Systems Engineering
Technische Universität München
85748 Munich/Garching, Germany

Abstract. This paper discusses a model-based approach to software development. It argues that an approach using models as central development artifact needs to be added to the portfolio of software engineering techniques, to further increase efficiency and flexibility of the development as well as quality and reusability of the results. Two major and strongly related techniques are identified and discussed: Test case modeling and an evolutionary approach to model transformation.

1 Portfolio of Software Engineering Techniques

Software has become a vital part of our lives. Embedded forms of software are part of almost any technical device from coffee machine to cars, the average household uses several computers, and the internet and telecommunication world has considerably changed our lives. All these machines are driven by a huge variety of software. Software that must never fail, must be updated dynamically, must continuously evolve to meet customers needs, must contain its own diagnosis and "healing" subsystems, etc. Software is used to for a variety of jobs, starting with the control of many kinds of processes up to simply to entertain. Software can be as small as a simple script or as complex as an entire operating or enterprise resource planning system.

Nowadays, there is some evidence that there will not be a single notation or process that can cover the diversity of today's development projects. Projects are too different in their application domain, size, need for reliability, time-to-market pressure, and the skills and demands of the project participants. Even the UML [1], which is regarded as a de-facto standard, is seen as a family of languages rather than a single notation and by far doesn't cover all needs. This leads to an ongoing proliferation of methods, notations, principles, techniques and tools in the software engineering domain. Some indicators of this ongoing diversity are:

- New programming languages, such as Python [2] without strong typing systems, but powerful capabilities for string manipulation and dynamic adaptation of their own program structure compete with Java, C++ and other conventional languages.
- The toolsets around XML-based standards [3] are widely and successfully used, even though they basically reinvent all compiler techniques known for 20 years.

M. Wirsing et al. (Eds.): RISSEF 2002, LNCS 2941, pp. 297–309, 2004.

- Methods like Extreme Programming [4] or Agile Software Development [5] discourage the long well known distinction between design and implementation activities and abandon all documentation activities in favor of rigorous test suites.
- Upcoming CASE tools allow to generate increasing amounts of code from UML models, thus supporting the OMG's initiative on "Model Driven Architecture" (MDA) [6]. MDA's primary purpose is to decouple platform-independent models and from platform-specific, technical information. This should increase the reusability of both.
- Completely new fields, like security, need new foundations embedded in practical tools. New logics and modeling techniques [7] are developed and for example used to verify protocols between mutually untrusted partners.

From this observations it is evident that in the foreseeable future we will have a *portfolio of software engineering techniques* that enables developers and managers to select appropriate processes and tools for their projects. To be able for such a selection developers need to be aware of this *portfolio of software engineering techniques* and master at least a comprehensible subset of these techniques. Today, however, it is not clear which elements the portfolio should have, how they relate, when they are applicable, and what their benefits and drawbacks are. The software and systems engineering community therefore must reconsider and extend its portfolio of software engineering techniques incorporating new ideas and concepts, but also try to scientifically assess the benefits and limits of new approaches. For example:

- Lightweight projects that don't produce requirement and design documentation need intensive communications and can hardly be split into independent subprojects. Thus they don't scale up to large projects. But where are the limits? A guess is, around 10 people, but there have been larger projects reportedly "successful" [23].
- Formal methods have built a large body of knowledge, but how can this knowledge successfully and goal-oriented be applied in today's projects? A guess seems to be, formal methods spread best, if embodied in practical tools, using practical and well known notations.
- Product reliability need not be 100% for all developments and already in the first iteration. But how to predict reliability from project metrics and how to adapt the project to increase reliability and accuracy to the desired level while minimizing the project/product costs?
- Promising techniques such as functional programming still need to find their place in practical software engineering, where they don't play an appropriate role today. A guess is that e.g., functional programming will be combined with object-techniques such that functional parts can be embedded in conventional programs.

Based on the observation for a general demand for a broad portfolio of SE techniques, we will in the following examine two trends that currently and in the foreseeable future will influence software engineering. These are on the one hand, the modeling notation UML and on the other hand agile development techniques. Although, these two trends currently work against each other, we can see that there is potential for their combination that takes benefits from both.

Section 2 discusses synergies and problems of using models for a variety of activities, including programming. Section 3 explores the possibilities of increasing efficiency and reducing development process overhead that emerges from the use of models as code and test case descriptions. In Section 4 the overall scenario of a model based test approach is discussed. Section 5 finally presents the benefits of an evolutionary approach to modeling in combination with an intensive, model-based test approach. In particular, the usability of tests as invariant observations for model-transformations is explored. For sake of conceptual discussions, technical details are usually omitted, but can be found in [8].

2 Modeling Meets Programming

UML [1] undoubtedly has become the most popular modeling language for software intensive systems used today. Models can be used for quite a variety of purposes. Among them are most common:

- Informal sketches are used for *communication*. Such a sketch is usually drawn on paper and posted on a wall, but not even used for documentation.
- More precisely defined and larger sets of diagrams are used for *documentation* of requirements and design. But requirements are usually captured in natural language and a few top-level and quite informal drawings that denote an abstract architecture, use cases or activity diagrams.
- Architecture and designs are captured and documented with models. In practice, these models are used for *code generation* increasingly often.

More sophisticated and therefore less widespread uses of models are analysis of certain features (such as throughput, failure likelihood) or development of tests from models. Many UML-based tools today offer functionality to directly simulate models or generate at least parts of the code. As tool vendors work hard on continuous improvement of this feature, this means a sublanguage of UML will become a high-level programming language and modeling at this level becomes identical to programming. This raises a number of interesting questions:

- Is it critical for a modeling language to be also used as programming language? For example analysis and design models may become overloaded with details that are not of interest yet, because modelers are addicted to executability.
- Is the UML expressive enough to describe systems completely or will it be accompanied by conventional languages? How well are these integrated?
- How will the toolset of the future look like and how will it overcome round trip engineering (i.e. mapping code and diagrams in both directions)?
- What implications does an executable UML have on the development process?

In [8,9] we have discussed these issues and have demonstrated, how the UML in combination with Java may be used as a high-level programming language. But, UML cannot only be used for modeling the application, but more importantly for modeling tests on various levels (class, integration, and system tests) as well. Execu-

table models are usually less abstract than design models, but they are more compact and abstract as the implementation.

One advantage of using models for test case description is that application specific parts are modeled with UML-diagrams and technical issues, such as connection to frameworks, error handling, persistence, or communication are handled by the parameterized code generator. This basically allows us to develop models independent of any technology or platform, as for example proposed in [10]. Only in the generation process platform dependent elements are added. When the technology changes, we only need to update the generator, but the application defining models can directly be reused. This concept also directly supports the above mentioned MDA-Approach [6] of the OMG. Another important advantage is that both, the production code and automatically executable tests at any level, are modeled by the same UML diagrams. Therefore developers use a single homogeneous language to describe implementation and tests. This will enhance the availability of tests already at the beginning of the coding activities. Similar to the "test first approach" [11,12], sequence diagrams are used for test cases and can be taken from the previously modeled requirements.

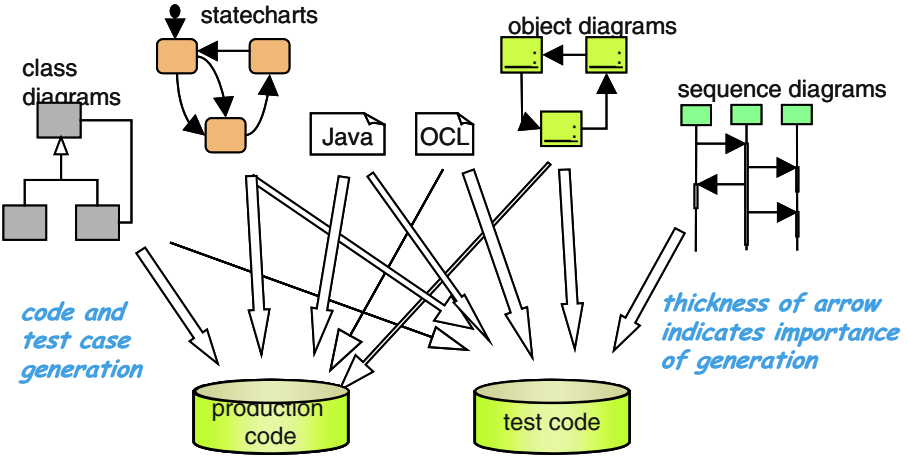


Fig. 1. Mapping of UML-models to code and test code.

Part of the UML-models (mainly class diagrams and statecharts) are used constructively, others are used for test case definition (mainly OCL, sequence and enhanced object diagrams). Fig. 1 illustrates the key mappings.

3 Agile Modeling: Using Models in Agile Projects

In the last few years a number of agile methods have been defined that share a certain kind of characteristics, described in [13]. Among these Extreme Programming (XP) [4] is the most widely used and discussed method. Some of the XP characteristics are:

- It focuses on the primary goal, the production code. Documentation instead is widely disregarded, but coding standards are enforced to document the code well.
- Automated tests are used on all levels. Practical experience shows, that when this is properly done, the defect rate is considerably low. Furthermore, the automation allows to repeat tests continuously.
- Very small iterations with continuous integration are enforced and the system is kept as simple as possible.
- Refactoring is used to improve the code structure and tests ensure a low defect rate introduced through refactoring.

The abandoning of documentation is motivated by the gained reduction of workload and the observation, that developers don't trust documents, because these usually are out of date. So, XP directly focuses on code. All design activities directly manifest in the code. Quality is ensured through strong emphasis on testing activities, ideally on development of the tests before the production code ("test first approach" [11]). An explicit architectural design phase is abandoned and the architecture emerges during coding. Architectural shortcomings are resolved through the application of refactoring techniques [14,15]. These are transformational techniques to refactor a system in small steps to enhance its structure. The concept isn't new [16], but through availability of tools and its embedding in XP, transformational development now becomes widely used.

When using an executable version of UML to develop the system within an agile approach, the development project should become even more efficient. On the one hand, through the abstractness of the platform independent models, these models are more compact and can more easily be written, read and understood than code. On the other hand in classic development projects these models are developed anyway. But, increased reuse of these models for later stages now becomes feasible through better assistance. Therefore, model-based development as proposed by the MDA-approach [6] becomes applicable. The separation of application models and platform specific parts that are combined through code generation only exhibits some characteristics of aspect oriented programming [17]. These UML-models also serve as up-to-date documentation much better than commented code does.

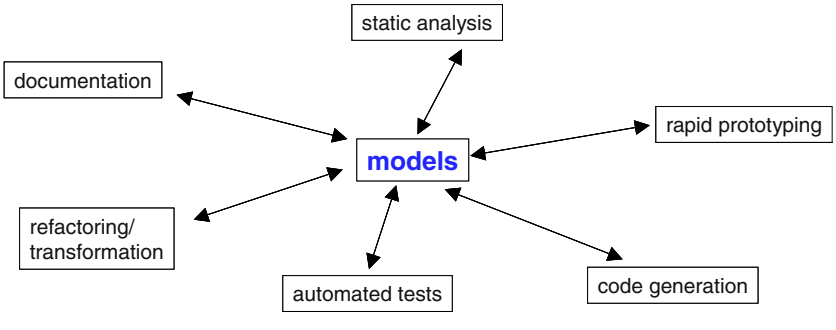


Fig. 2. The potential uses of UML-models.

To summarize, Fig. 2 shows the techniques used on models. This is quite in contrast to [18], where models are only used as informal drawings on the wall without further impact.

4 Model-Based Testing

There exists a huge variety of testing strategies [19,20]. The use of models for the definition of tests and production code can be manifold:

- Code or at least code frames can be generated from a design model.
- Test cases can be derived from an analysis or design model that is not used/usable for constructive generation of production code. For example behavioral models, such as statecharts, can be used to derive test cases that cover states, transitions or even paths.
- The modeling technique itself can be used to describe a test case or at least a part thereof.

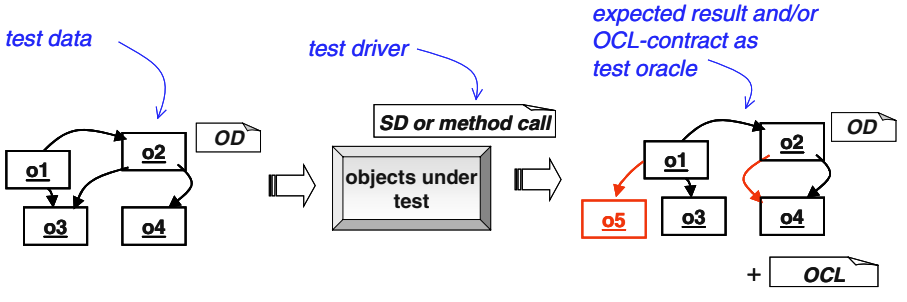


Fig. 3. Structure of a test modeled with object diagrams (OD), sequence diagram (SD) and the Object Constraint Language (OCL).

The first two uses are already discussed e.g. in [20]. Therefore, in this section we concentrate on the development of models that describe tests. A typical test, as shown in Fig. 3 consists of a description of the test data, the test driver and an oracle characterizing the desired test result. In object-oriented environments, the test data can usually be described by an object diagram (OD). It shows the necessary objects as well as concrete values for their attributes and the linking structure. The test driver can be modeled using a simple method call or, if more complex, a sequence diagram (SD). An SD has the considerable advantage that not only the triggering method calls can be described, but it is possible to model desired interactions and check object states during the test run.

For this purpose, the Object Constraint Language (OCL, [21]) is used. In the sequence diagram in Fig. 4, an OCL constraint at the bottom ensures that the new closing time of the auction is set to the time when the bid was submitted (bid.time) plus the extension time to allow competitors to react (the auction system containing this structure is in part described in [8,22]). Furthermore, it has proven efficient to model

the test oracle using a combination of an object diagram and OCL properties. The object diagram in this case serves as a property description and can therefore be rather incomplete, just focusing on the desired effects. The OCL constraints used can also be general invariants or specific property descriptions.

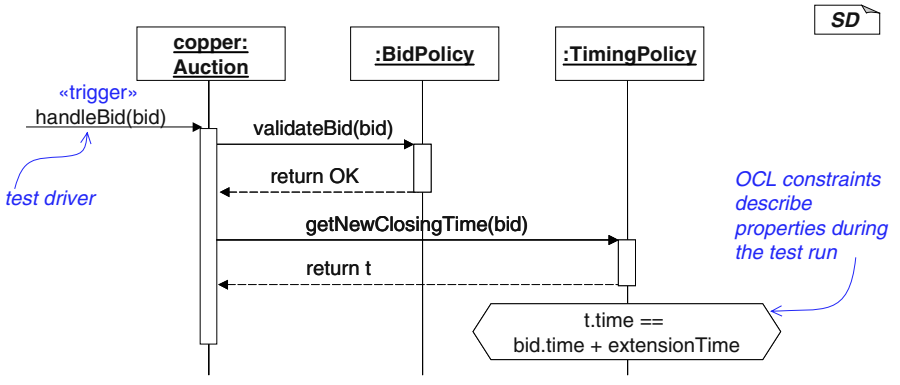


Fig. 4. A sequence diagram (SD) describing the trigger of a test driver and some test interactions as well as an OCL property that holds at that point of time.

As already mentioned, being able to use the same, coherent language to model the production system and the tests allows for a good integration between both tasks. It allows the developer to immediately define tests for the constructive model developed. It is imaginable that in a kind of "test-first modeling approach" the test data in form of possible object structures is developed before the actual implementation.

5 Model Evolution Using Automated Tests

Neither code nor models are initially correct. For code, many sources of incorrectness can rather easily be analyzed using type checkers of compilers and automated tests that run on the code. For models this is usually a problem that leaves many errors undetected in analysis and design models. This is particularly critical as conceptual errors in these models are rather expensive if detected only late in the development process. The use of code generation and automated tests helps to identify errors in these models.

Besides detecting errors, which might even result from considerable architectural flaws, nowadays, it is expected that the development and maintenance process is capable of being flexible enough to dynamically react on changing requirements. In particular, enhanced business logic or additional functionality should be added rapidly to existing systems, without necessarily undergo a major re-development or re-engineering phase. This can be achieved at best, if techniques are available that systematically evolve the system using transformations. To make such an approach manageable, the refactoring techniques for Java [14] have proven that a comprehensible set of small and systematically applicable transformation rules seems optimal. Trans-

formations, however, cannot only be applied to code, but to any kind of model. A number of possible applications are discussed in [16].

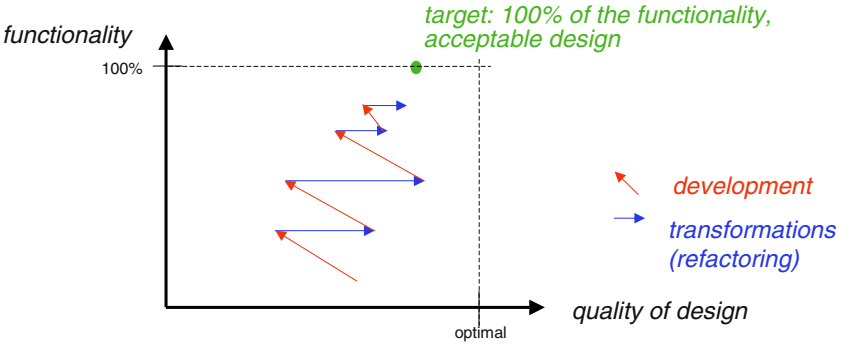


Fig. 5. Transformations to improve the quality of design opposed to development steps that add functionality.

Having a comprehensible set of model transformations at hand, model evolution becomes a crucial step in software development and maintenance. Architectural and design flaws can then be more easily corrected, superfluous functionality and structure removed, structure for additional functionality or behavioral optimizations be adapted, because models are more abstract, exhibit higher-level architectural and design information in a better way. During development, the situation can roughly be described with Fig. 5. It shows the dimension of functionality (measured for example in function points) and the "quality of design" (without a good metrics and therefore as informal concept). The development process tries to reach the 100% functionality while at the same time targets a reasonable good design.

The core development process ideally consists of step from two categories:

- Development (or programming) steps add functionality. But usually they in practice also introduce "erosion" of the design quality. For example repeated adding of new methods to a class overloads that class, simplifications of the code might come up, etc. Thus design quality usually suffers.
- Transformational (refactoring) steps build the second category. They improve structure and design, without changing the "externally observable behavior".

Two simple transformation rules on a class diagram are shown in Fig. 6. The figure shows two steps that move a method and an attribute upward in the inheritance hierarchy. The upward move of the attribute is accompanied by the only context condition, that the other class "Guest" didn't have an attribute with the same name yet. In contrast, moving the method may be more involved. In particular, if both existing method bodies are different, there are several possibilities: (1) Move up one method implementation and have it overridden in the other class. (2) Just add the method as abstract signature in the superclass. (3) Adapt the method implementations in such a way that common parts can be moved upward. This can for example be achieved by factoring differences between the two implementations of "checkPasswd" into

smaller methods, such that at the end a common method body for "checkPasswd" remains. As a context condition, the moved method may not use attributes that are available in the subclasses only.

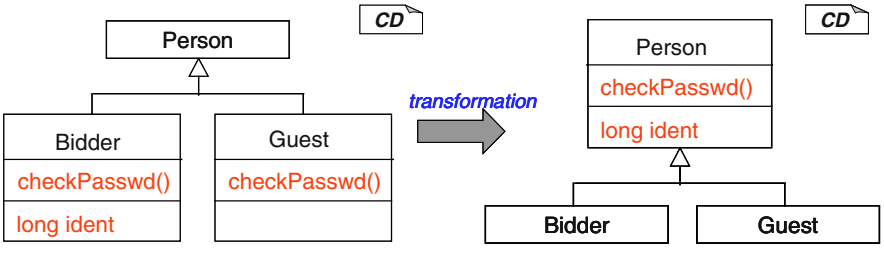


Fig. 6. Two transformational steps moving an attribute and a method along the hierarchy.

Many of the necessary transformation steps are as simple as the upward move of an attribute. However, others are more involved and their application comes with a larger set of context conditions and accompanying steps similar to the adaptation necessary for the "checkPasswd" method. These of course need automated assistance. The power of these simple and manageable transformation steps comes from the possibility to combine them and evolve complex designs in a systematic and traceable way.

Following the definition on refactoring [14], we use transformational steps for structure enhancement that does not affect "externally visible behavior". For example both transformations shown in Fig. 6 do not affect the external behavior if made properly.

By "externally visible behavior" Fowler in [14] basically refers to behavioral changes visible to the user. This can be generalized by introducing an abstract "system border". This border serves as interface to the user, but may also act as interface to other systems. Furthermore, in a hierarchically structured system, we may enforce behavioral equivalence for "subsystem borders" already. It is therefore necessary to explicitly describe, which kind of behavior is regarded as externally visible. For this purpose tests are the appropriate technique to describe behavior, because (1) tests are already available through the development process and (2) tests are automated which allows us to check the effect of a transformation through inexpensive, automated regression testing.

A test case thus acts as an "observer" of the behavior of a system under a certain condition. This condition is also described by the test case, namely through the setup, the test driver and the observations made by the test. Fig.7 illustrates this situation.

Fig. 7 also shows that tests do not necessarily constrain their observation to "externally visible behavior", but can make observations on local structure, internal interactions or state properties even during the system run. Therefore, it is essential to identify, which tests are regarded as "internal" and are evolving together with the transformed system and which tests need to remain unchanged, because they describe external properties of the system. Tests in one categorization can roughly be divided into unit tests, integration tests and acceptance tests.

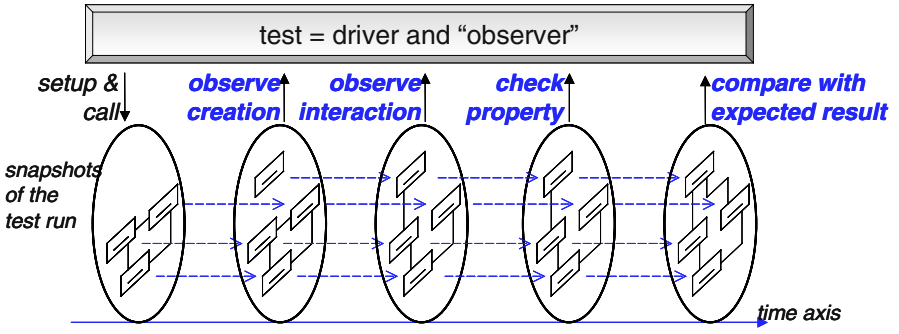


Fig. 7. A test case acts as observation.

Unit and integration tests focus on small parts of the system (classes or subsystems) and usually take a deep look into system internals. It therefore isn't surprising that these kinds of tests can become erroneous after a transformation of the underlying models. Indeed, these tests are usually transformed together with the code models. For example, moving an attribute upward as shown in Fig. 6 induces object diagrams with Guest-objects to be adapted accordingly by providing a concrete value for that attribute. In this case it may even be of interest to clone tests in order to allow for different values to be tested. Contrary, tests may also become obsolete if functionality or data structure is simplified. The task of transforming test models together with production code models can therefore not be fully automated.

Unit and integration tests are usually provided by the developer or test teams that have access to the systems internal details. Therefore, these are usually "glass box tests". *Acceptance tests*, instead, are "black box" tests that are provided by the user (although again realized by developers) and describe external properties of the system. These tests must be a lot more robust against changes of internal structure. Fig. 8 illustrates a commuting diagram that shows how an observation remains invariant under a test.

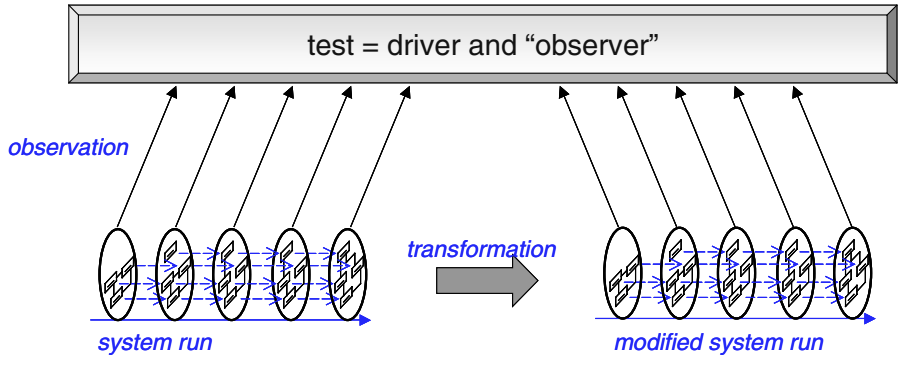


Fig. 8. The transformed system model is invariant under a test observation.

To achieve robustness, acceptance tests should be modeled against the published interfaces of a system. In this context "published" means that parts of the system that are explicitly marked as externally visible and therefore usually rather stable. Only explicit changes of requirements lead to changes of these tests and indeed the adaptation of requirements can very well be demonstrated through adaptation of these test models followed by the transformations necessary to meet these tests afterwards in a "test-first-approach". An adapted approach also works for changes in the interfaces between subsystems.

To increase stability of acceptance tests in transformational development, it has proven useful to follow a number of standards for test model development. These are similar to coding standards and have been found useful already before the combination with the transformational approach:

- In general an acceptance test should be abstract, by not trying to determine every detail of the tested part of the system.
- A test oracle should not try to determine every part of the output and the resulting data structure, but concentrate on important details, e.g. by ignoring uninteresting objects and attribute values.
- OCL property descriptions can often be used to model a range of possible results instead of determining one concrete result.
- Query-methods should be used instead of direct attribute access. This is more stable when the data structure is changed.
- It should not be tried to observe internal interactions during the system run. This means that sequence diagrams that are used as test drivers concentrate on triggers and on interactions with the system border only.
- Explicitly published interfaces that are regarded as highly stable should be introduced and acceptance tests should focus on these interfaces.

6 Conclusions

The proposal made in this paper can be summarized as a pragmatic approach to model-based software development. It uses models as primary artifact for requirements and design documentation, code generation and test case development. A transformational approach to model evolution allows an efficient adaption of the system to changing requirements and technology, optimizing architectural design and fixing bugs. To ensure the quality of such an evolving system, intensive sets of test cases are used. They are modeled in the same language, namely UML, and thus exhibit a good integration and allow to model system and tests in parallel.

However, the methodology sketched here still is a major proposal, adequate to be described in proceedings about the future of software technology. Major efforts still have to be done. On the one hand, even though initial works on various model transformations do exist, they are not very well put in context and not very well integrated with the UML in its current version. For example, it remains a challenge to provide automated support for the adaptation necessary for sequence diagrams that are af-

fected by statechart changes. Similarly, object diagrams usually are affected when the underlying class diagrams are changed. At least in the latter case, a number of results can be reused from the area of database schema evolution. But neither the pragmatic methodology, nor theoretic underpinning are very well explored yet, even though there is currently intensive research in the area of test theory development.

On the other hand, model based evolution will become successful only if well assisted by tools. This includes parameterized code generators for the system as well as for executable test drivers, analysis tools and comfortable help for systematic transformations on models. Today, there is not yet enough progress in these direction.

As a further obstacle, these new techniques, namely an executable sublanguage of the UML as well as a lightweight methodological use of models in a development process are both a challenge to traditional software engineering. They exhibit new possibilities and problems. Using executable UML allows to program in a more abstract and efficient way. This may finally downsize projects and decrease costs. The free resources can alternatively be used within the project for additional validation activities, such as reviews, additional tests or even a verification of critical parts of the system. Techniques such as refactoring and test-first design will change software engineering and add new elements to its portfolio.

To summarize, models can and should be used as described in this paper, but of course they are not restricted to. Instead it should be possible to have a variety of sophisticated analysis and manipulation techniques available that ideally operate on the same notations. These techniques should be used whenever appropriate. Even though, data and control flow techniques, model checking or even interactive verification techniques are already available, they still have to find their broad application to models.

The Model Driven Architecture (MDA) [6] initiative from the OMG has currently received lots of interest and several tools and approaches like “executable UML” are coming up to assist this approach. However, in the MDA community there is generally a belief that MDA only works for large and rather inflexibly run projects. It may therefore remain a challenging task, to derive efficient tools as described above and to adapt the currently used development processes to this very model-centric and agile development process.

Acknowledgements

I would like to thank Markus Pister, Bernhard Schätz und Tilman Seifert for commenting an earlier version of the paper as well as for valuable discussions. This work was partially supported by the Bayerisches Staatsministerium für Wissenschaft, Forschung und Kunst and through the Bavarian Habilitation Fellowship, the German Bundesministerium für Bildung und Forschung through the Virtual Software Engineering Competence Center (ViSEK).

References

1. OMG - Object Management Group. Unified Modeling Language Specification. V1.5. 2002.
2. Lutz M., Ascher D. Learning Python. O'Reilly & Associates. 1999.
3. W3C. Extensible Markup Language (XML) 1.0 (2nd ed.). http://www.w3.org/xml, 2000.
4. Beck, K. Extreme Programming explained, Addison-Wesley. 1999.
5. Cockburn, A. Agile Software Development. Addison-Wesley, 2002.
6. OMG. Model Driven Architecture (MDA). Technical Report OMG Document ormsc/2001-07-01, Object Management Group, 2001.
7. Jürjens J. UMLsec: Extending UML for Secure Systems Development. In: J.-M. Jezequel, H. Hussmann, S. Cook (eds): UML 2002 - The Unified Modeling Language, pages:412-425, LNCS 2460. Springer Verlag 2002.
8. Rumpe, B. Agiles Modellieren mit der UML. Habilitation Thesis. To appear 2003.
9. Rumpe, B. Executable Modeling with UML. A Vision or a Nightmare? In: Issues & Trends of Information Technology Management in Contemporary Associations, Seattle. Idea Group Publishing, Hershey, London, pp. 697-701. 2002.
10. Siedersleben J., Denert E. Wie baut man Informationssysteme? Überlegungen zur Standardarchitektur. Informatik Spektrum, 8/2000:247-257, 2000.
11. Link J., Fröhlich P. Unit Tests mit Java. Der Test-First-Ansatz. dpunkt.verlag Heidelberg, 2002.
12. Beck K. Aim, Fire (Column on the Test-First Approach). IEEE Software, 18(5):87-89, 2001.
13. Agile Manifesto. http://www.agilemanifesto.org/. 2003.
14. Fowler M. Refactoring. Addison-Wesley. 1999.
15. Opdyke W., Johnson R. Creating Abstract Superclasses by Refactoring. Technical Report. Dept. of Computer Science, University of Illinois and AT&T Bell Laboratories. 1993
16. Philipps J., Rumpe B.. Refactoring of Programs and Specifications. In: Practical foundations of business and system specifications. H.Kilov and K.Baclawski (Eds.), 281-297, Kluwer Academic Publishers, 2003.
17. Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopez C., Loingtier J.-M., Irwin J. Aspect-Oriented Programming. In ECOOP'97 - Object Oriented Programming, 11th European Conference, Jyväskylä, Finnland, LNCS 1241. Springer Verlag, 1997.
18. Ambler S. Agile Modeling. Effective Practices for Extreme Programming and the Unified Process. Wiley & Sons, New York, 2002.
19. Binder R. Testing Object-Oriented Systems. Models, Patterns, and Tools. Addison-Wesley, 1999.
20. Briand L. and Labiche Y. A UML-based Approach to System Testing. In M. Gogolla and C. Kobryn (eds): «UML» - The Unified Modeling Language, 4th Intl. Conference, pages 194-208, LNCS 2185. Springer, 2001.
21. Warmer J., Kleppe A. The Object Constraint Language. Addison-Wesley. 1998.
22. Rumpe B. E-Business Experiences with Online Auctions. In: Managing E-Commerce and Mobile Computing Technologies, Julie Mariga (Ed.) Idea Group Inc., 2003.
23. Rumpe B., Schröder A. Quantitative Survey on Extreme Programming Projects. In: Third International Conference on Extreme Programming and Flexible Processes in Software Engineering, XP2002, May 26-30, Alghero, Italy, pg. 95-100, 2002.

Predictable Component Architectures Using Dependent Finite State Machines

Heinz W. Schmidt[1], Bernd J. Krämer[2], Iman Poernomo[3], and Ralf Reussner[3]

[1] School of Computer Science and Software Engineering
Monash University, Melbourne, Australia
hws@csse.monash.edu.au
[2] Faculty of Electrical and Information Engineering
FernUniversität, Hagen, Germany
bernd.kraemer@fernuni-hagen.de
[3] CRC for Enterprise Distributed Systems Technology
(DSTC) Pty Ltd, Monash University
Melbourne, Australia
{imanp,reussner}@dstc.com

Abstract. The software architect is concerned with both functional and non-functional design. An important task in functional design is the adaptation of a component's provided interface for use by other components. In non-functional analysis the focus is rather on the prediction and reasoning about reliability and performance properties. We present a method for automatic adaptation, based upon *parameterised contracts*. This concept extends the notion of design-by-contract from precondition, postcondition and invariant assertions on objects to dynamic protocol descriptions for required and provided interfaces of components. We introduce a novel state machine based model, called *dependent finite state machines* (DFSMs), and show how DFSMs provide a natural framework for both automatic component adaptation and computational reasoning about timing properties of components and architectures. We use the well-known production cell example for demonstrating our architectural description language.

Keywords: automated interface adaptation, component-based interface specification, component-based prediction, finite state machines, production cell, protocol types, parameterised contracts, software architecture.

1 Introduction

The software architect is concerned with both functional and non-functional design. In component-based architectures, the former task can involve specifying, choosing, adapting, analysing and connecting components. The latter task involves analysis over properties such as reliability, availability, or performance. As system complexity increases, so do the costs associated with those tasks. It is therefore desirable to define methods and formalisms that can simultaneously address functional and non-functional concerns.

As a step in this direction we introduce a new modelling paradigm, called *Dependent Finite State Machines* (DFSMs); then we present component-oriented and distributed

M. Wirsing et al. (Eds.): RISSEF 2002, LNCS 2941, pp. 310–324, 2004.

software architecting features that extend the traditional object-oriented view of interface contracts to *parameterised contracts*. Previous presentations of parameterised contracts used finite state machine (FSM) descriptions of components for adaptation [21]. In this paper, we show how DFSMs permit compositional descriptions for components, making parameterised contracts more readily applicable to architecture description and analysis. DFSMs can also be used for the adaptation of interface functionality. Component adaptation is a well-known problem in software engineering. Components are designed to provide interfaces for external use but also to use interfaces of other components. It is often necessary to adapt a component's provided interface for use by another component. This is a costly task, and some level of automation is useful.

Our method for automatic adaptation relies on parameterised contracts, which adapt the notion of design-by-contract from objects to components and from precondition, postcondition and invariant assertions to dynamic protocol descriptions. Parameterised contracts define the contractual use of a component for composition with other components in some architecture. This permits the automatic adaptation through re-computation of a component's provided functionality depending on the functionalities required of other components. Conversely we can compute the minimal set of required services for a selected subset of a component's provided services.

Parameterised contracts can also be applied to non-functional analysis because parameterised contracts allow us to express how a component's non-functional behaviour depends on the behaviour of the components it uses. For example, the worst-case time of a real-time component may be given as a function of the time it takes to perform critical system services that are provided by the components it uses and that are mapped into the DFSM model. That the model allows us to address other non-functional properties as well has been shown for reliability in [19].

The rest of the paper is organised as follows. The running example is briefly discussed in section 2. Section 3 introduces DFSMs, architectural compositions on such machines and their role in the *Rich Architectural Description Language*, *radl* [20]. In section 4 we discuss design-by-contract for components and define parameterised contracts. Then we apply the DFSMs and parameterised contracts to model component and architecture adaptation and timing analysis. In section 6 we discuss related work and section 7 concludes.

2 Example Application

To illustrate our concepts and models, we use a variant of the production cell introduced in [10]. It describes the operation of a metal processing factory located in the southwest of Germany. A VRML[1] reconstruction of the production cell is depicted in Fig. 1. In this example we assume an external gadget to deposit individual metal blanks on the left end of the `feed belt` (left front) one after the other in arbitrary time intervals. The belt conveys these blanks to the `rotary table` at the belt's other end. Once the blank has been passed over, the table moves up and slightly rotates to bring the blank into a position and onto a level from where robot arm 1 can pick it up. After `arm1` has been loaded, the `robot base` rotates counter-clockwise into a position in which arm1

[1] Virtual Reality Modelling Language.

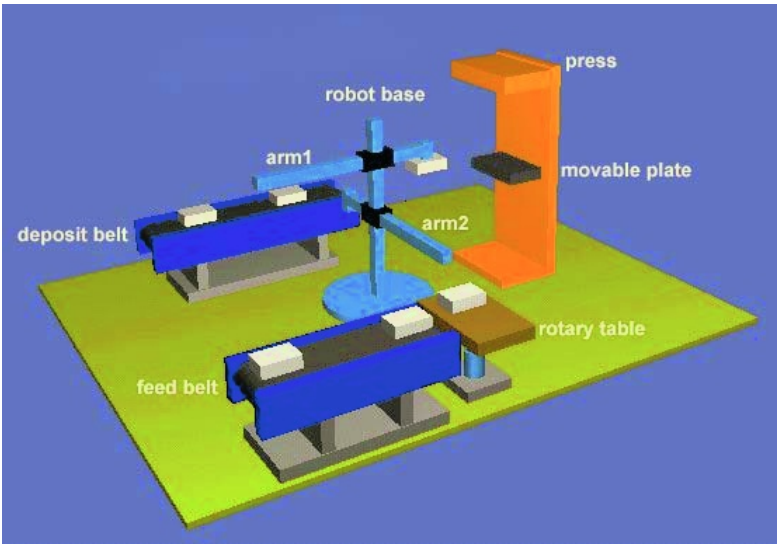


Fig. 1. Production Cell.

points at the press to introduce the blank into the `press` where it is forged while the `movable plate` of the press is closed.

To increase the use of the press, the robot is equipped with a second arm. `Arm2` picks up the part forged in the previous cycle from the press, the robot rotates counter-clockwise until arm 2 points toward the `deposit belt` and unloads the piece of metal on that belt. This belt moves processed parts to its other end. From here they are removed one by one by a second gadget in the environment. Arms1 and arm2 are mounted on different vertical levels and access the press while its movable plate rests in two different vertical levels (middle and bottom, respectively). The arms can take load with their grippers independently through magnets. They can also be extended or retracted horizontally. Sensors monitor the load and unload zones of the conveyor belts and signal significant positions of certain actuators. We assume that the robot, the press and the conveyor belts have their own software controller. These controllers communicate by some form of message passing.

In the rest of the paper we abbreviate the various operations according to the mapping shown in Table 1. The following operations are non atomic: `load` extends the arm, takes a load and retracts the arm again; `unload` extends the arm, drops the load and retracts the arm again; `process` moves the plate up from the middle position and thereby forges a blank placed on the movable plate, then moves the plate two levels down; `remove` removes the processed blank and moves the press plate one level up.

3 Foundations

This section introduces the basic notions underlying the *radl* architectural description language. It presents finite state machine models that provide semantics to interaction

Table 1. Action abbreviations with worst-case time.

Actuator actions		
`on`	switch gripper magnet on	
`off`	switch gripper magnet off	
`in`	switch arm expander to `pull in arm`	
`out`	switch arm expander to `push out arm`	
`tl`	turn robot base one step left	
`tr`	turn robot base one step right	
`up`	move press plate one level up	
`down`	move press plate one level down	
Controller actions with time		
`take`	take load with gripper magnet	10
`drop`	drop load held by gripper magnet	10
`ret`	retract arm	300
`ext`	extend arm	300
`rcw`	rotate robot base clockwise by one position	400
`rcc`	rotate robot base counter clockwise by one position	400
`press`	move press plate one level up	30
`remove`	remove processed blank	10
`bot`	signal bottom position	2
`mid`	signal middle position	2
`rup`	elevate rotary table	40
`rdown`	lower rotary table	40
`load`	load arm	
`unload`	unload arm	

protocols. Such protocols form an integral part of *radl* interfaces. A translation relation serves to connect a component's provided and required interfaces. Dependent finite state machines lay the foundation of parameterised contracts presented in section 4.

3.1 Component Based Software Architectures

Based on extensions of design-by-contract to distributed systems' components, the *radl* architectural description language (ADL) was originally described in [20] and has been extended later in [17]. In *radl*, a system is decomposed into hierarchies of *kens*, linked to each other by connections between *gates*.

Kens are coarse-grain, computational entities typically communicating by message passing via interface objects. Kens can represent components or other architectural units. Fig. 2 shows the kens of the production cell architecture as boxes. *Primitive Kens* are the lowest-level kens. They are usually implemented as binary components, whose internal structure is transparent to the architect. In our example, the subkens of `Robot`, `Arm1`, `Arm2` and `Base`, are primitive kens. *Composite kens* address compositionality. They represent hierarchically organised collections of kens with well-defined connections. In our example, `Robot` and the production cell system are composite kens.

Gates protect kens, which cannot be directly accessed. All calls, all data communicated and all objects migrated to a ken must come through gates. They represent

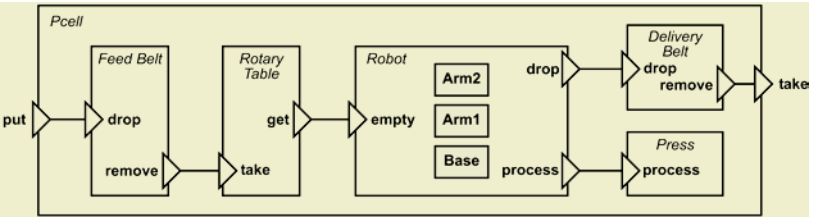


Fig. 2. Production Cell Architecture.

interface types and objects. In architecture diagrams, gates are represented by triangles, the direction of which indicates control, not necessarily signal flow. For example, the `Robot` controls the press via a gate, despite the fact that acknowledgement signals may flow back via this same gate. Depending on its role as controlled or controlling, a gate is *provided* or *required* (short PGate and RGate). *Connections* are established between gates as part of the architectural composition, like in Darwin [12], at deployment or at run time (dynamic connections are not needed for our example).

Kens and gates are analogous to components and services, respectively, in Darwin; to components and ports, respectively, in C2 and ACME; or to processes and ports in MetaH [14]. However, the *radl* distinguishes different kinds of components reflecting their architectural role in distributed systems analysis, design and implementation.

3.2 Finite State Machine Models of Component and Connector Protocols

In *radl* protocols are part of contracts. Protocols are types. A protocol type is defined as a set of valid sequences of service calls and is represented by FSMs. For example, the `Expander` FSM in Fig. 3 (left) permits sequences `(ext,ret)*`.

Services have parameter and result types specified in signatures. Signatures are part of interface definitions in *radl* and are used in assertions. We distinguish between input, output and hidden symbols in signatures. Relative to a component of interest, the inputs are controlled from outside, they are incoming service calls; the outputs correspond to outgoing calls; they are controlled by the component. The hidden symbols represent auxiliary tests and actions. The protocols in terms of FSMs do not depend on service parameters, although notations such as UML state charts permit passing of such parameters to actions associated with protocol elements. Both PGates and RGates have associated interface specifications including protocol types for interoperability checking or automatic adaptation.

In elementary components such as a robot arm expander, gripper magnet, robot base, or press we permit PGates to model *machine interfaces*, i.e., operations that directly control the hardware, notably sensors and actuators. These gates are *ungrounded*, i.e., unconnected *terminator gates*. They can be viewed as the abstract machine of a real-world physical primitive ken not shown. The terminator PGate models the full physical behaviour, i.e., all possible actions of the actuator including destructive actions. For example, the arm expander can either be extended or retracted (note that we are only interested in the machine's discrete behaviour). Hence, the two machine operations

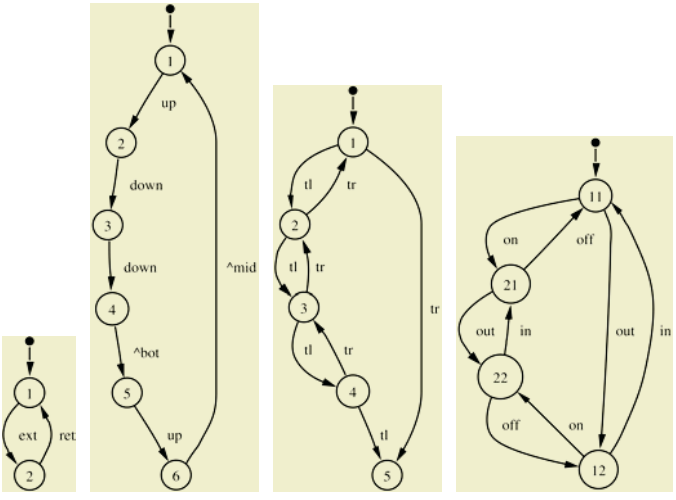


Fig. 3. A renaming `Expander=Switch[on: ext, off: ret]` (left); a substitution with acknowledgement `Press=Switch[on: (up,down,down,^bot), off: (up,^mid)]` (center left); the robot `Base` PGate actuator protocol (center right) can physically reach an unsafe state (destruction in 5); the `Arm` protocol (right) is the product of an `Expander` and `Gripper`.

`in` and `out` can only be applied in sequence. This behaviour is a signature renaming `Arm=Switch[on: in, off: out]`, of the `Switch` type, which maps `on` to `in` and `off` to `out`. We use a caret (`^`) to mark output symbols (signals).

Protocol Types and Conformance. Formal definitions of FSMs and their relationships to formal languages and Boolean algebra can be found in automata theory textbooks. For brevity, we use all operations on regular language as operations on FSMs, too, and loosely identify an FSM A and the language L_A representing sets of transition sequences leading from the initial state to some final state. Since we treat protocols as a separate aspect of interface types in *radl*, we define a *protocol type* as the equivalence class of FSMs A generating the same language L_A. Then the inclusion of FSMs lends itself as a notion of protocol subtyping.

Definition 1. *Given two protocols A and B, A* is a protocol subtype *of B (denoted $A \leq B$) iff $L_B \subseteq L_A$. In this case we also say that A* conforms to *B.*

If A conforms to B, then all sequences expected according to a protocol B are acceptable by protocol A. A may optionally allow more sequences than B. We should explicitly mention the fact that the equivalence and conformance of two protocol type specifications A and B is decidable.

A common constructive form of conformance is the use of restriction $A|_B = \{x \in A | (x)_{\Sigma_B} \in B\}$, which selects only traces in A that observe the protocol B. Here, $(x)_B$ ($(A)_B$) denotes the projection of trace x (of language A) to symbols in B, deleting

symbols[2] not in B. For example, we can derive an adapter called `SafeBase` through a restriction of robot `Base` in Fig. 3 by simply subtracting state 5, which represents the destructive state into which the robot can get by overturning the base in left or right direction. `SafeBase`, which is part of the robot base controller's provides interface, prevents such a destructive move to happen. Obviously, we have that $L_{\texttt{Base}} \leq L_{\texttt{SafeBase}}$.

All the above operators are decidable. Not only can static configurations of kens and gates be checked for protocol-type correctness (under conformance) but it is also realistic to perform conformance checks dynamically when components are deployed or loaded.

3.3 Complex Connectors and Adapters

In general, interoperability may require more complex translations from required to provided gates, the two end points of a connection. Sometimes translations involve several gates. To this end we partition the alphabet into three pairwise disjoint sets $\Sigma = I \cup H \cup O$: the input I, hidden H and output symbols O. Furthermore we assume Σ is equipped with a reflexive and symmetric relation D, called dependency. Σ is then also called a dependence alphabet, which uniquely extends (sequential) words into (parallel) partial orders. Dependence alphabets extend regular languages to parallel trace languages [5] and regular expressions (finite state automata) to rational star-connected expressions (or Petri nets that are automata products, respectively) [16]. Modularity and closure properties are preserved in this extension. Intuitively the complement C_D of D is viewed as a *concurrency* relation. With aC_Db, the word ab is equivalent to ba and represents the single-step parallel transition of a and b that takes time $t(ab) = max(t(a), t(b))$ if a symbol $s \in \Sigma$ takes time $t(s)$. In contrast, $t(ab) = t(a) + t(b)$ if aDb. Dependence alphabets extend our protocols to protocols for trace languages.

Now we can "observe" input, hidden and output subprotocols in trace languages L_A using the corresponding projections $(L_A)_I$, $(L_A)_H$, and $(L_A)_O$, which "delete" symbols not in the subalphabet listed. A protocol type A over a partitioned dependence alphabet can then be viewed as a translation from parallel inputs to parallel outputs.

Definition 2. *The* translation relation *(short* translation*) of a protocol type A, $\Theta_A \subseteq (L_A)_I \times (L_A)_O$ is defined by the minimal set of pairs satisfying*

$$w_I \Theta_A w_O \text{ if } w \in L_A.$$

Θ, the function taking protocol types to their translations as defined above, is called the translation functor.

Due to this underlying translation we call the languages and FSMs on partitioned alphabets *translator protocol type*. We use these types for characterising the dependency a component's computation establishes between its provided and required gates.

Theorem 1. *Protocol translation types are closed under the usual regular language operations.*

[2] The result of a projection is usually not a sublanguage. However restrictions are sublanguages defined via intermediate projections.

Note that, in contrast, finite state transducers[3] are not closed under all regular language operations.

Using the algebra of translations with regular language operators, we can now build complex adapters and connectors for true parallel (trace) languages starting from simple input- and output-only gates. Synchronised and asynchronous call abstractions may be defined by substitutions as shown in Fig. 3. In [22], we showed the construction of more complex adapters for synchronising concurrent protocols.

3.4 Dependent FSMs

Protocol subtyping permits decidable conformance tests – yet that is not sufficient for dealing effectively with non-functional properties. Protocols types as defined above capture functional interface types – performance analysis often requires an understanding of *system configuration* partly at the level of instances permitting the distinction of two separate or parallel entities from two tightly synchronised or even identical instances (possibly of the same type). This observation has led architecture definition languages such as our $radl$ or the forthcoming UML2 architecture definition to view gates (and their connections) as instances of such protocol types and group multiple gates into ports and associate multiple ports with a single component. In addition, components are intrinsically *open*. Their performance properties critically depend on properties of external components. For example, the reliability of a component, from the viewpoint of its provided services is not only a function of its own implementation but also of the reliability of invoked external components including those in the underlying platform (for instance, middleware framework, operating system, hardware). Worse, each of the invoked external services must also be factored into the invoking component's reliability in a different way. For example, if RGate R is accessed many times while RGate S is accessed infrequently, then the reliability of R must weigh in considerably more significantly than that of S. Thus components are intrinsically *parameterised* by the protocols of those gates and the component translation protocol acts on those parameters.

The above aspects are captured in the following notion of dependent FSMs. They establish (1) the association between gates and a component, (2) the translation relationship the component imposes on its gates, and (3) a hierarchical structuring of this translation type. This enables us to build *architectural networks* of dependent components and compute dependent FSMs from simpler ones following the architectural compositions from primitive to composite gates and components.

Definition 3. *A* dependent FSM*, short DFSM, is a triple $D = (R, A, P)$, where $P = (P_i)_{i<l}$ is a family of FSMs, called the* provided parameters *of D, $R = (R_j)_{j<m}$ is a family of FSMs, called the* required parameters *of D, and, A is an FSM called the* abstract effect *or* abstract machine *of D, with $l, m \in \mathbb{N}$. Furthermore, we require that $\Sigma_P = I$ and $\Sigma_R = O$ and call D* consistent *if the following two conditions hold:*

1. *$(A)_{P_i} \leq P_i$, i.e. A accepts all inputs acceptable to the provided gates.*
2. *for all $R_i \in R$ we have that $R_j \leq (A)_{R_j}$, i.e., A generates outputs acceptable to the required gates.*

[3] Functions from input to output languages defined by Mealy machines.

Note that the above notion of type consistency adds constraints beyond pure connection interoperability. A collection of $DFSM$s with gate bindings, which impose so-called *R-P dependencies* on an architecture, can now be treated as architectural protocols. Beyond the interface definition P and R, obviously, a DFSM includes an abstract model (A) of the dependency imposed on these interfaces by the component implementation. We call this *P-R dependency* (see also Fig. 4). As A models aspects of the implementation, in whatever abstraction level, we promote a *grey-box* approach to components. For each provided gate P_i and each provided (input) symbol s (in fact for each input sequence s), the translation type Θ_A defines the output language $s\Theta_A$.

By extension, the output language of P by considering $\bigcup_{i<m} L_{P_i}\Theta_A$ is the *output language* of D.

Since regular languages are closed under the operations in the above definition, the following proposition is noteworthy:

Corollary 1. Equivalence *and* inclusion *of DFSMs* are decidable. *There is a straightforward algorithm for DFSM minimisation implied by Definition 2.*

Notice that this result holds for DFSMs over partitioned independence alphabets.

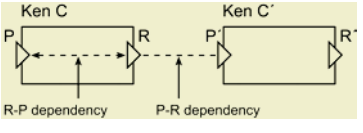


Fig. 4. P-R dependency models consistency of interfaces with component behaviour (Def. 3), while R-P dependency models conformance (Def. 1).

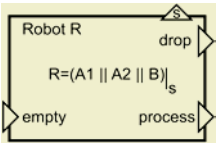


Fig. 5. Composition constrained by synchronisation constraint S, which is actualised by different parameters in different configurations.

4 Generalizing Parameterised Contracts

When addressing the contractual use of components, we need to distinguish between the use of a component at run time or at composition time. In the first case, one is actually using the methods of the component. According to Meyer's design-by-contract principle [15], the contractual use of a component at run time implies that a method's precondition is met by the caller while it must ensure the postcondition itself. To lift this notion from the level of methods to the level of the components, we view the required interface specification as the component's precondition, and the provided interface as the component's postcondition.

We denote the precondition of a component c by pre_c and its postcondition by $post_c$. For checking whether a component c can be replaced safely by a component c', one has to ensure that the contract of c' is a subcontract of c. The notion of a subcontract is described in [15, p. 573] like contravariant typing for methods: A contract c' is a subcontract of contract c iff

$$pre_{c'} \trianglelefteq pre_c \wedge post_{c'} \trianglerighteq post_c \tag{1}$$

where $\trianglerighteq$ means "stronger", i.e., subtype. If pre_c and $post_c$ are predicates, $\trianglerighteq$ corresponds to $\Rightarrow$. In our DFSM semantics for protocol types, $\trianglerighteq$ is subtyping.

To check the interoperability (i.e., R-P dependency) between components c and c' (as defined in section 3.4, see also Fig. 4), one has to check whether:

$$pre_c \trianglelefteq post_{c'}. \tag{2}$$

When extending these concepts to DFSM enriched interfaces, we can consider the precondition of a component as the set of required method call sequences, i.e., the shuffle product of the languages accepted by the FSMs of the RGates. The postcondition is the set of provided call sequences (i.e., the union of the languages accepted by the FSMs of the PGates). In this case, the checks described in formulas (1) and (2) boil down to inclusion checking.

While interoperability tests check the required interface of a component against the provided interface of *another* component, parameterised contracts link the provided interface of one component to the requires interface of *the same* component (see P-R dependency in Fig. 4). Parameterised contracts were formally introduced in a more limited fashion in [18]. Here we extend these notions to work with multiple gates and partial actualisation. Actualisation can, e.g., be used to impose synchronisation constraints on the composition of protocols of composite components.

An example is shown in Fig. 5. The contract of a `Robot` instance is specified as the parallel composition of the contracts of its constituent component instances `Arm1` (`A1`), `Arm2` (`A2`) and `Base` (`B`) constrained by parameter `S`. A typical actualisation of `S` would be a sequential organisation of safety-critical controller actions:

$$\texttt{S} = \texttt{A1.load}, \texttt{B.rcc}, \texttt{A2.load}, \texttt{B.rcc}, \texttt{A1.unload}, \texttt{B.rcc}, \texttt{A2.unload}, \texttt{B.ret} \tag{3}$$

where `B.ret` = `B.rcw`, `B.rcw`, `B.rcw`. Note also that the controller actions `rcw` and `rcc` control the actuator actions `tl` and `tr`, respectively (see also Table 1).

Originally, the concept of parameterised contracts was motivated by the observation that in practice often only a subset of a component's functionality is used. So, even if the environment offers less functionality than required, the component might still be able to provide its own full functionality. The binding of a formally required gate to an actual non-conforming component gate is then viewed as parameter actualisation and the protocol algebra is used to compute exactly the weakest provided parameters that can still be offered in the given context. The parameterisation modelled with DFSMs below goes beyond such adaptations. It permits non-functional parameterisation such as variations in synchronisation constraints (S above), timing, and other properties.

For a PGate P_i (or RGate R_i), we define the set of all subtypes of this gate as $\mathbf{P}_i$ (or $\mathbf{R}_i$, respectively). The set of all possible preconditions $\mathbf{Pre}_C$ (or postconditions $\mathbf{Post}_C$) of a component C is then the set of all possible tuples in $\mathbf{R}_0 \times \cdots \times \mathbf{R}_{l-1}$ (or $\mathbf{P}_0 \times \cdots \times \mathbf{P}_{m-1}$, respectively). We denote the union of all these tuple sets by $\mathbf{Typ}*$.

The P-R dependency between the PGates and the RGates of a component C (as introduced in Section 3.4) can be written as $C[\mathbf{Pre} \rightarrow \mathbf{Post}]$ and interpreted as mapping between corresponding tuples. A *parameterised contract* is any mapping $\Phi : \mathbf{Typ}* \rightarrow \mathbf{Typ}*$ that is total, bijective and monotone with respect to $\trianglerighteq$ (hence from $pre_1 \trianglerighteq pre_2$ we can deduce $\Phi(pre_1) \trianglerighteq \Phi(pre_2)$). Φ is called *parameterised contract* since it

parameterises the postcondition with the precondition of the component and vice versa. It can be considered as a generalisation of "classical contracts", which use a fixed pre- and postcondition.

If component A uses only a subset of the functionality offered by B, we compute a new required interface for B with the parameterised contract Φ_B:

$$\Phi_B^{-1}(req_A \cap prov_B) =: req'_B \subseteq req_B \tag{4}$$

This gives us a subset of the functionality B requires from C. The new required interface req'_B is weaker (requires possibly less) than the original required interface $req_B := \Phi_B^{-1}(prov_B)$ (but not more) since Φ_B is monotone and $req_A \cap prov_B \subseteq prov_B$.

For example, in a different configuration of the production cell we might have another type of physical `Press`, which has only one access level (e.g., `^mid` but not `^bot`) and thus only admits a behaviour equivalent to the `Switch` protocol (`up`, `down`)*, which defines the required interface of the new `Press`. As a consequence, the provided interface of the `Press` is restricted to (`up`, `down`, ^mid)* and the trigger `bot` for `a2.load` in the `Robot` provided protocol disappears. This would imply that `Arm 2` cannot be used in this production cell configuration.

Likewise, if component C does not provide all the functionality required by B, one can compute a new provided interface $prov'_B$ with Φ_B:

$$\Phi_B(req_B \cap prov_C) =: prov'_B \subseteq prov_B \tag{5}$$

Since Φ_B is monotone, Φ_B^{-1} is monotone as well. With $req_B \cap prov_C \subseteq req_B$ we have $prov'_B \subseteq prov_B := \Phi(req_B)$.

Since Φ is monotone and bijective, the required parameterised contract always has to return the strongest provided interface (strongest postcondition), while the provided parameterised contract always returns the weakest required interface (weakest precondition[4]).

5 Analysing Timing Properties

In an industrial cooperation in the area of embedded control systems, the first author is currently developing component-based methods for predicting worst case time based on DFSMs with timed transitions. Here a set of DFSMs represents a real-time distributed control system. We solve the problem with a one-pass algorithm computing the maximum time of all paths ending in a state before adding the times for successive transitions following from that state. We assume bounded loops (which corresponds to the requirements for function blocks) and rely on a well-formedness restriction for intertwined loops.

The plain algorithm is based on graph theory and allows us to compute time bottom up and compositionally in terms of the component machines of a DFSM. Parameterisation allows us to actualise synchronisation constraints and update the time models. Here, synchronisation constraints are viewed as required gates that are provided at configuration or deployment time (see also Fig. 5).

[4] Note the similarity to the wp-calculus.

For the composition given in Fig. 5, the synchronisation constraint S specified in (3) and the time estimates for primitive controller actions given in Table 1, the cycle time of the cell core is

$$4 * (300 + 10 + 300) + 6 * 400 + 8 * 2.5 = 4860 \text{ (i.e., } 4.9\text{sec)}$$

if we assume the worst-case synchronisation time is $2.5(msec)$.

The synchronisation constraint in (3) is overly conservative as it imposes a tight sequential order on the robot's actions. It is, however, safe to relax the constraint such that the arm expander movements and base rotations can overlap in time. An actualisation of the Robot composition with the following relaxed constraint and the same elementary times

$$\mathtt{S}' = \mathtt{A1.M.take}, \mathtt{B.rcc}, \mathtt{A2.M.take}, \mathtt{B.rcc}, \mathtt{A1.M.drop}, \mathtt{B.rcc}, \mathtt{A2.M.drop}, \mathtt{B.ret} \quad (6)$$

relaxes the worst-case execution time to: $4 * 10 + 6 * 400 + 8 * 2.5 = 2460$ (i.e., 2.5sec) because $t(\mathtt{ret}) = t(\mathtt{exp}) \leq t(\mathtt{rcc}) = t(\mathtt{rcw})$. Now the critical trace does not include the expander movements of the arms and hence the cycle time of the core cell is approximately halved. Note that this relaxation is a change of interaction constraints between robot arm1 and robot base, i.e., internal to the robot. In general, such relaxations may invalidate safety invariants, which must therefore be re-established.

In general, worst-case execution time models are only compositional under certain sufficient constraints for well-behaviour. For example, if we rely on failure-free synchronisation and deadlock freedom, our time models are compositional. This reliance is a proof obligation, which may or may not require access to the details of the components or the environment they are working in. Therefore, depending on the component model chosen or the compositions considered, this premise itself may not be compositional.

6 Related Work

In a series of papers, we have explored aspects of our approach for a range of areas, such as modelling transactional contexts [17] and adapting component protocols [22]. This paper is the first time we have used DFSMs and parameterised contracts as a single unifying framework for adaptation and non-functional analysis.

Our basic approach to syntax and overall behavioural semantics is similar to that of other ADLs [14]. Some form of semantics is used to define models of the behaviour of basic components. For example, the semantics of Darwin [12] has been successfully modelled with the π-calculus and labelled transition systems [13]; Rapide [11] uses partially ordered event sets and Wright [1] makes use of a variant of CSP. The language of the ADL is used primarily to impose structure on a system's behaviour, expressing its composition from components. Composition-forming constructs are associated with semantic functions, which are used to analyse a composite component's behavioural semantics in terms of its subcomponents' behavioural semantics. Typically, such analysis involves establishing global checks such as liveness and safety properties.

Cheung used architectural information in combination with usage profiles for predicting system reliability following the flow of control across component boundaries [4].

Wang extended Cheung's work by analysing different architectural styles such as pipelines or filters [24]. This line of work seems to aim at availability. Moreover, it is not open for parameterisation and compositionality.

Hamlet in [7] advocates a statistical view of component-oriented architectures. He identifies the need for usage profiles clearly but requires source code for his computations. In contrast, our method works with binary black-box components and architectural assembly and it recognises the strong influence of required components' reliability.

In [6] Frolund and Koistinen provide a syntax for defining and specifying quality of service attributes. Similar to our work, they emphasise the contractual use of quality of service attributes. However, they specify reliability as constant, while we use parameterised contracts for computing context-dependent component reliability.

To the best of our knowledge, little work has been done previously on combining protocol-based adaptation and non-functional prediction with an ADL. Parameterised contracts allow us to model functional and non-functional properties and dependencies in one framework. State machines are a well-known notation for protocol specification and adaptor generation [26]. There is a range of adaptation mechanisms that do not use interface information [3], such as delegation, wrappers, superimposition, or metaprogramming [9]. Since $radl$ strongly depends on protocol information, tools for generating provided FSMs and abstract machine DFSMs through control-flow analysis of code [8] are vital. Furthermore, tool support exists for deriving FSMs from message sequence charts [25].

7 Conclusions

The functionality provided by a component always depends on the functionality received from its environment. Hence, especially in those systems in which many different configurations and environments exist, the ability to perform fine-grained adaptations and to model and predict non-functional properties is crucial. We presented our ADL $radl$ as a unified framework for specifying and computing functional and non-functional properties of component-based software-architectures. The underlying principle for automatic protocol adaptation and reliability prediction are parameterised contracts, which we presented as a generalisation of interoperability checks between components (i.e., classical contracts of components). We introduced DFSMs as a novel model for specifying parameterised contracts. The well-known production-cell example was partly specified in $radl$ and the benefits of parameterised contracts were discussed in the context of this example.

Our approach has been applied to industrial control automation, which increasingly relies on the international standard IEC 61131-3. This standard supports a component concept including hierarchical composition in terms of so-called Function Blocks and it defines a basic set of standardized function blocks. Control automation applications benefit from the possibility to can proven or calculated properties together with the code as they are typically built from library components. In addition, the certification of critical properties such as safety, reliability and timing is often a must [23]. Other classes of applications that may benefit from our protocol specifications include telecommunication applications [22] and contract-aware applications of distributed object technology. In

Beugnard's four level model we are particularly addressing synchronisation level and quality of service level contracts [2].

The *radl* editor and DFSMs are currently implemented in Java. The core of the representation is a `FSM` class based on a number of auxiliary classes including hash library classes. It is used to represent the elements of a DFSM, i.e., its gates and its connecting abstract machines.

For the future, it seems worthwhile to explore the use of parameterised contracts within a technique for compositional reasoning on global system properties in terms of local component properties. When considering an adapted component together with its environment as a component, we are able to describe the functionality of this higher-level component in terms of its constituent components. This prediction of properties of composed systems currently only works for layered systems. Since in practice many other composition pattern besides layered systems occur, future work is concerned with more general mechanisms to predict properties of the overall software architecture from properties of the single components and their interaction patterns.

References

1. R.J. Allen. *A Formal Approach to Software Architecture*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PE, USA, May 1997.
2. A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *IEEE Computer*, 32(7):38–45, 1999.
3. J. Bosch. *Design and Use of Software Architectures – Adopting and evolving a product-line approach*. Addison-Wesley, Reading, MA, USA, 2000.
4. R.C. Cheung. A user-oriented software reliability model. *IEEE Transactions on Software Engineering*, 6(2):118–125, 1980.
5. V. Diekert and G. Rosenberg (ed.) The book of traces. World Scientific Publ. Co., 1995.
6. S. Frolund and J. Koistinen. Quality-of-service specification in distributed object systems. Technical Report HPL-98-159, Hewlett Packard, Software Technology Laboratory, 1998.
7. D. Hamlet, D. Mason, and D. Woit. Theory of software reliability based on components. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE-01)*, 361–370, IEEE Computer Society 2001.
8. G. Hunzelmann. Generierung von Protokollinformation für Softwarekomponenten-schnittstellen aus annotiertem Java-Code. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe (TH), Germany, April 2001.
9. G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28(4):154–154, 1996.
10. C. Lewerentz and T. Lindner, editors. *Formal Development of Reactive Systems - Case Study Production Cell*. Number 891 in LNCS. Springer-Verlag, Berlin, Germany, 1995.
11. D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.
12. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of ESEC '95 - 5th European Software Engineering Conference*, volume 989 of *Lecture Notes in Computer Science*, pages 137–153, Springer-Verlag, Berlin, 1995
13. J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. Wiley & Sons, New York, NY, USA, 1999.
14. N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

15. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, USA, 2 edition, 1997.
16. E. Ochmanski. Recognizable Trace Languages. in [5] 165–203
17. I.H. Poernomo, R.H. Reussner, and H.-W. Schmidt. Architectures of Enterprise Systems: Modelling Transactional Contexts. In *Proceedings of the First IFIP/ACM Working Conference on Component Deployment (CD 2002)*, volume 2370 of *Lecture Notes in Computer Science*, 233–243. Springer-Verlag, Berlin, 2002.
18. R.H. Reussner. *Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten*. Logos Verlag, Berlin, 2001.
19. R.H. Reussner, H.-W. Schmidt, and I.H. Poernomo. Reliability prediction for component-based software architectures. *Journal of Systems and Software*, 66(3):241–253, Elsevier 2003.
20. H.-W. Schmidt. Compatibility of interoperable objects. In Bernd Krämer, Michael P. Papazoglou, and Heinz W. Schnmidt, editors, *Information Systems Interoperability*, 143–181. Research Studies Press, Taunton, England, 1998.
21. H.-W. Schmidt. Trustworthy components: Compositionality and prediction. *Journal of Systems and Software*, 65(3):215–225, Elsevier 2003.
22. H.-W. Schmidt and R.H. Reussner. Generating Adapters for Concurrent Component Protocol Synchronisation. In *Proceedings of the 5th International Conference on Formal Methods for Open Object-based Distributed Systems*, in B Jacobs and A Rensink (eds.), 213–229, Kluwer Academic Publisher, Massachusetts 2002
23. N. Völker, B.J. Krämer. Automated verification of function block-based industrial control systems. Science of Computer Programming, Vol. 42, (1):101-113, Elsevier 2002
24. W.-L. Wang, Y. Wu, and M.-H. Chen. An Architecture-Based Software Reliability Model. In *Proceedings of the 1999 Pacific Rim International Symposium on Dependable Computing, December, Hong Kong, China*. IEEE, 1999.
25. B. Wydaeghe. *Component Composition Based on Composition Patterns and Usage Scenarios*. Dissertation, Department of Computer Science, Vrije Universitiet Brussel, 2001.
26. D. Yellin and R. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.

From Object Orientation to Goal Orientation: A Paradigm Shift for Requirements Engineering

Axel van Lamsweerde and Emmanuel Letier

Département d'Ingénierie Informatique
Université catholique de Louvain
B-1348 Louvain-la-Neuve (Belgium)
{avl,eletier}@info.ucl.ac.be

Abstract. Requirements engineering (RE) is concerned with the elicitation of the objectives to be achieved by the system envisioned, the operationalization of such objectives into specifications of services and constraints, the assignment of responsibilities for the resulting requirements to agents such as humans, devices and software, and the evolution of such requirements over time and across system families. Getting high-quality requirements is difficult and critical. Recent surveys have confirmed the growing recognition of RE as an area of primary concern in software engineering research and practice. The paper reviews the important limitations of OO modeling and formal specification technology when applied to this early phase of the software lifecycle. It argues that goals are an essential abstraction for eliciting, elaborating, modeling, specifying, analyzing, verifying, negotiating and documenting robust and conflict-free requirements. A safety injection system for a nuclear power plant is used as a running example to illustrate the key role of goals while engineering requirements for high assurance systems.

Keywords: Goal-oriented requirements engineering, high assurance systems, safety, specification building process, lightweight formal methods.

1 Introduction

The requirements problem has been with us for a long time. An early empirical study over a variety of software projects revealed that inadequate, inconsistent, incomplete, or ambiguous requirements are numerous and have a critical impact on the quality of the resulting software [Bel76]. Late correction of requirements errors was observed to be incredibly expensive [Boe81]. A consensus has been growing that engineering high-quality requirements is difficult; as Brooks noted in his landmark paper on the essence and accidents of software engineering, *"the hardest single part of building a software system is deciding precisely what to build"* [Bro87]. In spite of such early recognition, the requirements problem is still with us – more than ever. Recent surveys over a wide variety of organizations and projects in the United States and in Europe have confirmed the problem on a much larger scale; poor requirements have consistently been recognized to be the major cause of software problems such as cost overrun, delayed delivery or failure to meet expectations [Sta95, ESI96]. The problem gets even more serious in the case of safety-critical or security-critical systems; most

M. Wirsing et al. (Eds.): RISSEF 2002, LNCS 2941, pp. 325–340, 2004.

severe failures have been recognized to be traceable back to defective specification of requirements [Lut93, Lev95, Kni02].

Semi-formal modeling notations à la UML and formal specification techniques have been proposed as candidate solutions to address the requirements problem. The strength of the former is their usability (at the price of fairly imprecise semantics), their support for multiple system views and their standardization. The strength of the latter is the wide variety of analysis tools they provide for algorithmic model checking, deductive verification, specification animation, specification-based testing, specification reuse and specification refinement; as a result, the number of success stories in using formal specification technology for real systems is steadily growing from year to year [Lam00c].

In spite of such good news, traditional semi-formal modeling and formal specification techniques suffer from serious weaknesses that explain why they are not fully adequate for the upstream, critical phase of *requirements* elaboration and analysis.

- ***Limited scope.*** The vast majority of techniques focus on the modeling and specification of the software alone. They lack support for reasoning about the *composite system* made of the software and its environment. Inadequate assumptions about the environment in which the software operates are however known to be responsible for many errors in requirements specifications [Jac95, Lev95]. *Non-functional requirements* are also generally left outside any kind of treatment. Such requirements form an important part of any specification; they are known to play a prominent role in the evaluation of alternatives, the management of conflicts, the derivation of architectures and evolution management [Chu00, Lam00b, Lam03].
- ***Lack of rationale capture.*** Detailed requirements specifications are difficult to understand. Efforts have been made towards formal notations that are more readable [Har87, Heim96, Heit96]. Such efforts however do not address the problem of understanding requirements in terms of their rationale with respect to some higher-level concerns in the application domain.
- ***Poor guidance.*** The main emphasis in modeling and specification has been on suitable sets of notations and tools for *a posteriori* analysis. Constructive methods for building correct models/specifications for complex systems in a systematic, incremental way are by and large non-existent. The problem is not merely one of translating natural language statements into some semi-formal model and/or formal specification. Requirements engineering in general requires complex requirements to be elicited, elaborated, structured, interrelated and negotiated.
- ***Lack of support for exploration of alternatives.*** Requirements engineering is much concerned with the exploration of alternative system proposals in which more or less functionality is automated. Different assignment of responsibilities among software/environment components yield different software-environment boundaries and interactions. Traditional modeling and specification techniques do not allow such alternatives to be represented, explored, and compared for selection.

In this paper, we argue that *goals* offer the right kind of abstraction to address such inadequacies, notably, in the specific context of high assurance systems, that is, systems for which compelling evidence is required that the system delivers its services in a manner that satisfies safety, security, fault-tolerance and survivability requirements [Lea95].

G*oals* are declarative statements of intent to be achieved by the system under consideration [Dar93, Lam00b]. The word "system" here refers to the software-to-be together with its environment [Fea87, Fic92]. Goals are formulated in terms of prescriptive assertions (as opposed to descriptive ones) [Zav97]; they may refer to functional or non-functional properties and range from high-level concerns (such as "safe nuclear power plant") to lower-level ones (such as "safety injection overridden when block switch is on and pressure is less than 'Permit'). *Agents* are system components such as humans playing specific roles, devices and software. A *requirement* is a goal whose achievement is under responsibility of a single software agent. An *expectation* is a goal whose achievement is under responsibility of a single environment agent.

Modeling and reasoning about goals is especially important for high assurance systems as some of the system goals correspond to the application-specific safety, security, fault tolerance and survivability properties that need be achieved with high assurance. Positive/negative interactions with the other system goals can be captured in goal models and managed appropriately [Lam98]; exceptional conditions in the environment that may prevent critical goals from being achieved can be identified and resolved to produce more robust requirements [Lam00a]; the goals can be specified precisely and refined incrementally into operational software specifications that provably assure the higher-level goals [Dar96, Let02a, Let02b]. Requirements in fact "implement" goals much the same way as programs implement design specifications.

The paper discusses the relevance and benefits of explicitly modeling and reasoning about goals at various levels of abstraction in the specific context of high assurance systems. We illustrate the use of a comprehensive set of goal-oriented techniques to build and analyze the requirements for a safety injection control system [Cou93]. Although fairly small, this case study comes from a real application, raises many of the issues found in high assurance systems and is frequently used to illustrate other methods such as, e.g., the SCR method [Heit96] and its analysis techniques [Bha99, Jef98, Gar99]. Other illustrations involving first-order formalizations can be found in [Lam00a, Lam00b, Let01].

2 Goal-Oriented RE in Action: Elaborating Requirements for a Safety Injection System

We follow our KAOS method to gradually derive operational requirements for the safety injection software from the underlying system goals. (KAOS stands for "Keep All Objectives Satisfied").

A goal refinement graph is elaborated first by identifying relevant goals from the preliminary system description [Cou93], typically by looking for intentional keywords in natural language statements and by asking *why* and *how* questions about such statements (*goal elaboration step*); conceptual classes, attributes and associations are derived from the goal specification (*object modeling step*); agents are identified together with their potential monitoring/control capabilities, and alternative assignments of goals to agents are explored (*agent modeling step*); operations and their domain pre- and postconditions are identified from the goal specifications, and strengthened pre-, post- and trigger conditions are derived so as to ensure the corresponding goals (*operationalization step*). In parallel, two other steps of the method

handle conflicting goals and obstacles that may obstruct goal satisfaction, respectively. The suggested ordering among steps corresponds to an idealized process; in practice however there is significant intertwining and backtracking between them.

Our presentation will be succinct and fragmentary for space reasons; the interested reader may refer to [Let02c] for a full treatment of the case study.

2.1 Goal Identification from the Source Document

Fig. 1 shows some preliminary goals that have been directly identified from the first two paragraphs of the preliminary description of the safety injection system [Cou93]. This figure can be read as follows. One goal in a nuclear power plant is to maintain an effective coolant system (EffectiveCoolantSystem). This goal can be obstructed by an *obstacle* such as LossOfCoolant. (Obstacles may be seen as a high-level faults derived from goal negations; techniques for systematically identifying ways in which a system may fail will be discussed more precisely below.)

The goal SafetyInjection**Iff**LossOfCoolant is introduced to mitigate the obstacle. This goal is then refined into

- an accuracy property about the environment: LossOfCoolant**Iff**LowWaterPressure,
- the subgoal SafetyInjection**Iff**LowWaterPressure.

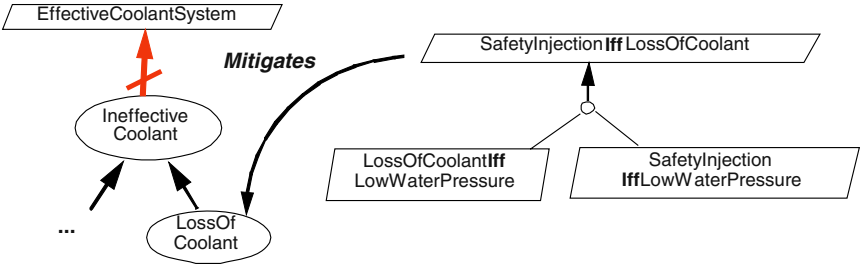


Fig. 1. Preliminary goals identified from initial description of the safety injection system [Cou93]

2.2 Formalizing Goals, Modeling Objects and Identifying State Variables

Formal analysis techniques may complement informal or semi-formal ones in order to provide higher assurance in the correctness and completeness of the system requirements. Goals then need to be formalized to enable their use. As we will see, goal formalization also allows for more systematic guidance in the requirements elaboration process.

In addition to the usual logical connectives, the following linear temporal operators will be used in this paper:

◊ P	P holds in some future state
□ P	P holds in all future states
A ⇒ C	In every future state A implies C, i.e., □(A → C)
A ⇔ C	In every future state A is equivalent to C, i.e., □ (A ↔ C)
● P	P holds in the previous state
@ P	P has just become true, i.e., ●¬ P ∧ P

For example, the goal Maintain[SafetyInjection**Iff**LowWaterPressure] may be defined as follows:

Goal Maintain [SafetyInjection**Iff**LowWaterPressure]

InformalDef *The safety injection signal should be 'On' when and only when the water pressure is below the 'Low' set point.*

FormalDef SafetyInjectionSignal = 'On' ⇔ WaterPressure < 'Low'

The above goal refers to state variables WaterPressure and SafetyInjectionSignal that are declared as attributes of corresponding conceptual classes in a preliminary object model (see Fig. 2).

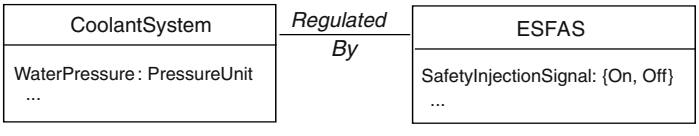


Fig. 2. Goal-driven object modeling

These attributes receive the following physical interpretation:

WaterPressure: *the actual pressure of water in the coolant system*

SafetyInjectionSignal: *signal sent by the ESFAS (Engineered Safety Feature Actuation System) to safety features components to command the actual safety injection mechanisms*

Conceptual classes, attributes and associations are incrementally identified and defined as the requirements model is elaborated. When first-order formalizations are used, associations are typically derived from atomic formulas involved in the formal goal assertions [Lam00b]. As opposed to standard OO modeling where it is never clear how and why such or such class/attribute/association should enter the picture, *goal-based object modeling is grounded on a precise criterion for identifying elements of the object model*; such elements are modelled when and only when they are involved in declarative assertions about goals and requirements. Also note the difference with use-case driven modeling; here we start from higher-level, general, declarative and precise statements of intent rather than generally overspecific, operational and often imprecise descriptions of operations achieving goals left implicit. In fact, use cases can be trivially generated at the very last, operationalization step of our method (see Section 2.7).

2.3 Detecting and Resolving Goal-Level Conflicts

Another goal appearing in the available source document is to avoid actuation of the safety injection system during normal start-up or cool down phases:

Goal Avoid [SafetyInjectionDuringNormalStartUp/CoolDown]

InformalDef *Safety injection signals should not be sent during normal start-up or cool down.*

FormalDef (NormalStartUp $\vee$ NormalCoolDown) $\Rightarrow$ SafetyInjectionSignal = 'Off'

This new goal introduces a conflict with the goal Maintain[SafetyInjection**Iff**LowWaterPressure] previously identified. This conflict is detected formally using a predefined conflict pattern from [Lam98]. The two goals are in fact not logically inconsistent; however, they become inconsistent when the plant is in start-up or cool down phase and the water pressure is below 'Low'. This condition is called *boundary condition for conflict* [Lam98]; its formal definition is generated formally by instantiation of our formal conflict pattern which yields:

$$\Diamond\ (\ (\text{NormalStartUp} \vee \text{NormalCoolDown}) \wedge \text{WaterPressure} < \text{'Low'})$$

Conflict resolution tactics from [Lam98] may then be used to propose alternative resolutions; in this case, the conflict is resolved by *weakening* the goal Maintain[SafetyInjection**Iff**LowWaterPressure] with the predicate appearing in the boundary condition. We thereby obtain:

Goal Maintain [SafetyInjection**Iff**LowWaterPressure**Except**DuringStartUp/CoolDown]

InformalDef *The safety injection signal should be 'On' whenever there is a loss of coolant, except during normal start-up or cool down.*

FormalDef SafetyInjectionSignal = 'On' $\Leftrightarrow$

WaterPressure < 'Low' $\wedge \neg$ *(NormalStartUp* $\vee$ *NormalCoolDown)*

This goal will be refined and operationalized in the following sections.

2.4 Refining Goals and Identifying Agent Responsibilities

Goals have to be refined until they can be assigned as responsibilities of single agents. However, a goal can be assigned to an agent only if this agent has sufficient monitoring and control capabilities to realize the goal [Let02a]. (Our terminology here is based on the 4-variable model [Par95] and the notion of shared phenomena [Jac95].)

For example, the goal Maintain[SafetyInjection**Iff**LowWaterPressure**Except**DuringStartUp/CoolDown] is unrealizable by the 'Engineered Safety Feature Actuation System' (ESFAS) because this agent cannot monitor whether the plant is in normal startup or cooldown phase.

A catalog of agent-based refinement tactics has been defined to guide the process of refining unrealizable goals until realizable subgoals are reached [Let02a]. Each tactic suggests the application of a formal refinement pattern (see Fig. 3).

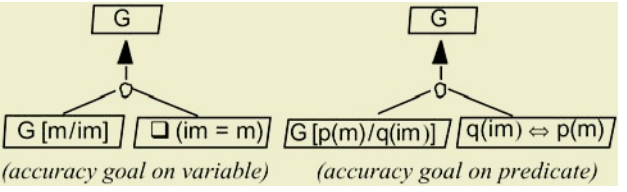


Fig. 3. The 'Introduce accuracy goal' tactics

The first tactic in Fig. 3 may be used to resolve ESFAS' lack of monitorability of state variables NormalStartUp and NormalCoolDown. Applying the corresponding pattern yields a new, monitorable state variable, Overridden say, and a refinement of the unrealizable goal Maintain[SafetyInjection**Iff**LowWaterPressure**Except**DuringStartUp/CoolDown] into two subgoals:

- a subgoal SafetyInjection**Iff**LowWaterPressure**Except**When*Overriden*, formally defined by

  SafetyInjectionSignal = 'On' ⇔

  WaterPressure < 'Low' ∧ ¬ *Overridden*

- a companion *accuracy* goal SafetyInjection*Overridden*DuringStartUp/CoolDown, formally defined by

  Overridden ⇔ (NormalStartUp ∨ NormalCoolDown)

Such formal definitions are generated by instantiation of the formal refinement pattern associated with the selected tactic. Goal refinement patterns are proved correct once for all [Dar96]; the STEP verification system [Man96] may be used to check that the conjunction of leaf nodes entails the parent node. At every pattern application the user gets an instantiated proof of correctness of the refinement for free.

The above first subgoal SafetyInjection**Iff**LowWaterPressure**Except**When*Overriden* is now realizable by the ESFAS software agent because it is entirely defined in terms of variables that turn to be monitorable and controllable by this agent; the first subgoal therefore becomes a *requirement* on that agent.

The accuracy subgoal SafetyInjection*Overridden*DuringStartUp/CoolDown is still not realizable by the ESFAS agent because this agent still lacks monitorability of state variables NormalStartUp and NormalCoolDown. Agent-based refinement tactics may again be used to guide the generation of alternative refinements for this goal. One alternative consists in:

(1) introducing two new variables, Block and Reset, that represent manual *block* and *reset* buttons controlled by a human Operator agent;

(2) assigning to the Operator agent the responsibility of pushing the *block* button when and only when the plant enters normal cooldown/startup, and the responsibility of pushing the *reset* button when and only when the plant leaves normal cooldown/startup (the latter two subgoals turn out to be realizable by the Operator agent and therefore become environment *assumptions*); and

(3) assigning to the ESFAS agent the responsibility of overriding safety injection if and only if 'block' is pushed, and the responsibility of enabling safety injection if and only if 'reset' is pushed (the latter two subgoals turn out to be realizable by the ESFAS software agent and therefore become *software requirements*).

Further details about the generated goal graph and responsibility assignments may be found in [Let02c].

Note that both software requirements and environmental assumptions are in general needed to prove higher-level goals.

2.5 Deriving Agent Interfaces

Capturing the agents' monitoring and control capabilities is an important aspect of the requirements elaboration process [Fea87, Par95, Jac95]. Such capabilities were gradually identified during the previous goal refinement step. The resulting agent interface model for the safety injection system is shown in Fig. 4. It corresponds to a context diagram [Jac95].

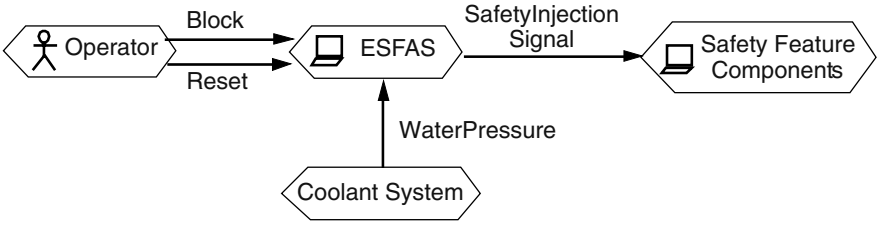


Fig. 4. Derived agent interface model for the safety injection system

Note that alternative goal refinements and alternative responsibility assignments in general lead to alternative software-environment boundaries, that is, alternative system proposals and agent interfaces in which more or less is automated.

2.6 Generating and Resolving Obstacles to Goal Achievement

First-sketch specifications of goals, requirements and assumptions tend to be over-ideal; they are likely to be violated from time to time in the running system due to unexpected behavior of agents. The lack of anticipation of exceptional behaviors may result in unrealistic, unachievable and/or incomplete requirements. We capture such exceptional behaviors by formal assertions called *obstacles* to goal satisfaction.

An obstacle *O* is said to obstruct a goal *G* iff

$$\{O, Dom\} \models \neg G \quad \textit{obstruction}$$

$$Dom \not\models \neg O \quad \textit{domain consistency}$$

Obstacle analysis consists in taking a pessimistic view at the goals, requirements, and assumptions elaborated. The idea is to identify as many ways of breaking such

properties as possible in order to resolve them and produce more complete requirements for more robust systems.

We just illustrate a few results from obstacle analysis for some of the terminal goals we identified before. For example, in the previous goal refinement process, we made the following idealized assumption on the behavior of the Operator agent:

Assumption Avoid[ManualBlock**When**NoStartUp/CoolDown]

InformalDef *The block button should not be pushed when the plant is not entering normal startup or cool down.*

FormalDef $\neg$ @ (NormalStartUp $\vee$ NormalCoolDown) $\Rightarrow \neg$ @ (Block = 'On')

UnderResponsibilityOf Operator

In this case, by just taking the negation of the above assumption we would identify the following obstacle:

Obstacle OperatorPushesBlock**WhenNot**InStartUp/CoolDown

InformalDef *'Block' is pushed when the plant is not entering normal startup or cool down.*

FormalDef $\Diamond$ ($\neg$ @ (NormalStartUp $\vee$ NormalCoolDown) $\wedge$ @ (Block = 'On'))

Similarly, from the assumption Achieve[ManualResetOnExitFromStartUp/CoolDown] assigned to the Operator agent, we would identify the obstacle OperatorForgetsToReset. Other obstacles to assumptions on the Operator agent and to requirements on the ESFAS agent can be identified in the same way [Let02c].

Formal techniques for obstacle generation and refinement are detailed in [Lam00a]. The basic technique amounts to a precondition calculus that regresses goal negations backwards through known properties about the domain; formal obstruction patterns may be used as an alternative to shortcut formal derivations. A formal completeness criterion is also given in [Lam00a]; such completeness is bound by the set of properties known about the domain. Our techniques allow the analyst to incrementally elicit new domain properties as well.

Obstacles should be resolved once they have been generated. Obstacle resolution involves assessing the likelihood and criticality of the obstacle, investigating alternative ways of resolving it, and choosing one resolution alternative based on various criteria such as cost, risks, performance, etc.

Obstacle resolution tactics may be used to generate alternative resolutions [Lam00a]. For example, one of our tactics yields a resolution of the obstacle OperatorPushesBlockWhenNotInStartUp/CoolDown in which an alternative refinement of the higher-level goal SafetyInjectionOverriddenDuringStartUp/CoolDown is considered; in this alternative, the responsibility of the Operator agent is *weakened,* so as to partially cover the obstacle, whereas the responsibility of the ESFAS agent is *strengthened*. Such an alternative design might be identified by observing that pushing the block button when the water pressure is above some specified value 'Permit' is necessarily an Operator's error because of a domain property stating that the plant cannot be in normal startup/ cooldown at such high pressure. Accordingly, the requirement on the ESFAS agent is strengthened so that safety injection does *not* become overridden if the block button is pushed when the water pressure is above 'Permit':

Goal Maintain [SafetyInjectionOverridden**When**BlockSwitchOn**And***PressureLessThanPermit*]

InformalDef *Safety injection should become overridden when, and only when, the block switch is set to 'On' while the water pressure is less than 'Permit'.*

FormalDef @ Overridden $\Leftrightarrow$
@ (Block = 'On') $\wedge$ *WaterPressure* $\leq$ *'Permit'* $\wedge$ $\bullet$ $\neg$ Overridden

UnderResponsibilityOf ESFAS

The obstacle OperatorForgetsToReset is resolved in a similar way by weakening the responsibility of the Operator agent and strengthening the responsibility of the ESFAS agent. In this case, the requirement of the ESFAS agent is strengthened so that safety injection becomes automatically enabled when the water pressure raises above 'Permit'.

Our resolution tactics so far include goal substitution, agent substitution, goal weakening, goal restoration, obstacle prevention and obstacle mitigation [Lam00a]. In general several generated resolutions will be applicable so that a "best" alternative needs to selected according to non-functional goals from the goal graph (we come back to this below). The selection and application of a resolution may be carried out at specification time, to produce more robust requirements specifications, or at run time, when a requirements monitor detects that the obstacle does occur or is likely to occur [Fea98].

Note that obstacle analysis is an iterative process; it may produce new goals for which new obstacles may need to be identified. In the resulting software specification, some of the obstacles may be totally or partially resolved, some obstacles may remain unchanged (e.g., if they are highly unlikely, do not matter or are deferred to run time) and some new obstacles may appear as a result of previous resolutions.

As mentioned before, the selection among alternative resolutions and the decision to iterate further obstacle analysis cycles should be based on some trade-off assessment among various non-functional, application-specific goals about safety, security, cost, performance, etc. This is an area where much work remains to be done. Qualitative techniques might help here by exposing the competing influences of various alternatives with respect to non-functional goals. A preliminary proposal can be found in [Chu00] where a procedure is proposed for propagating positive/negative influences along alternative paths in the goal graph. For high assurance systems, however, more accurate, quantitative techniques are required. For example, probabilistic risk assessment techniques might provide more precise input to the decision making process. Such techniques, however, rely on the availability of accurate estimates of probabilities of failure events. Obtaining such data may be problematic; the use of such quantitative techniques has therefore been controversial [Lev95]. The real challenge is probably to define a decision process that combines *qualitative reasoning* for those non-functional aspects of the system for which no accurate quantitative weighting can be made and *quantitative reasoning* for those non-functional aspects for which meaningful weighting can be obtained.

Obstacle analysis may be seen as a goal-oriented, formal, constructive method for building fault trees and recovery actions. It is particularly relevant to high assurance systems as many problems and failures of such systems are known to be caused by poor designs that are unable to cope with errors caused by humans, devices and software [Lev95].

2.7 Deriving Operational Requirements from System Goals

The next step of the requirements elaboration process consists in deriving operational software specifications from the terminal goals assigned to software agents. The result is an operation model that defines the various services to be provided by the software in terms of their pre-/postcondition in the domain and strengthened conditions ensuring that the underlying goals in the goal model are met by the services.

A catalog of formal operationalization patterns is available to support the operationalization step [Let02b]. For example, the 'Immediate Achieve' pattern is shown in Fig. 5.

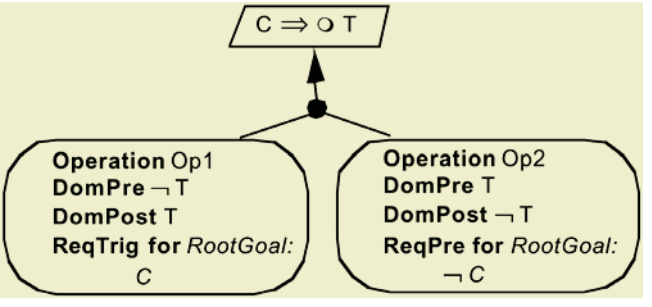


Fig. 5. The 'Inmediate Achieve' pattern

Let us come back to the goal

Maintain [SafetyInjectionOverridden**When**BlockSwitchOn**And***PressureLessThanPermit*]

that we assigned to the ESFAS agent, and to the right-to-left implication in the formal definition of this goal given in Section 2.6. The 'Immediate Achieve' operationalization pattern can be used to derive the following operational requirements:

Operation OverrideSafetyInjection

PerformedBy ESFAS

Input Block, WaterPressure; **Output** Overridden

DomPre ¬ Overridden

DomPost Overridden

ReqPre/Trig For SafetyInjectionOverridden**When**BlockSwitchOn
AndPressureLessThanPermit:

@ (Block = 'On') ∧ WaterPressure ≤ 'Permit'

Note that a distinction is made between *domain* pre- and postconditions that capture what any application of the operation means in the application domain, and *required* pre-, trigger, and postconditions that capture requirements on the operations that are necessary to achieve the goals. (Such distinction somewhat corresponds to the distinction between indicative and optative properties in [Jac95, Zav97].)

In the above operation, the ReqPre/Trigger keyword is a syntactic shortcut to express that the condition is both a required pre- *and* a required trigger- condition for the satisfaction of the corresponding goal; the operation *may* be applied *only if* the condi-

tion is true and *must* be applied *if* the condition becomes true and the domain precondition is true.

Similarly, from the goal Maintain[SafetyInjectionEnabled**When**PressureAbovePermit**Or**ManualReset], we can systematically derive the need for an operation EnableSafetyInjection together with strengthened conditions on this operation that will guarantee the satisfaction of this goal. Specifications for the operations SendSafetyInjectionSignal and StopSafetyInjectionSignal are similarly derived from the specification of the goal Maintain[SafetyInjection**When**LowWaterPressure**AndNot**Overridden], see [Let02c] for details.

It is worth noting that our *goal-oriented requirements elaboration process ends where most traditional specification techniques would start.* For example, the operational specifications obtained above can be mapped to SCR tables for the same system through a series of transformation steps each of which resolves a semantic, structural or syntactic difference between the source (KAOS) specification and the target (SCR) one [Del03].

Note again the difference with use-case driven modeling; we started from higher-level, general, declarative and precise statements of intent rather than generally over-specific, operational and often imprecise descriptions of operations achieving goals left implicit. Use cases emerge at the last step of our method as aggregations of the operations that operationalize functional goals assigned to software agents.

3 Conclusion

We used a safety injection system as a running example to illustrate the benefits of a constructive, goal-oriented approach to requirements elaboration and analysis. The key points illustrated by this elaboration process are the following.

- Goal-oriented modeling and specification takes a wider system engineering perspective; goals are prescriptive assertions that should hold in the system made of the software-to-be *and its environment*; domain properties and expectations about the environment are explicitly captured during the requirements elaboration process, in addition to the usual software requirements specifications.
- Operational requirements are derived incrementally from the higher-level system goals they "implement".
- Goals provide the rationale for the requirements that operationalize them and, in addition, a correctness criterion for requirements completeness and pertinence [Yue87].
- Obstacle analysis helps producing much more robust systems by systematically generating (a) potential ways in which the system might fail to meet its goals and (b) alternative ways of resolving such problems early enough during the requirements elaboration and negotiation phase.
- Alternative system proposals are explored through alternative goal refinements, responsibility assignments, obstacle resolutions and conflict resolutions.

- The goal refinement structure provides a rich way of structuring and documenting the entire requirements document.
- A multiparadigm, 'multi-button' framework allows one to combine different levels of expression and reasoning: *semi-formal* for modeling and structuring, *qualitative* for selection among alternatives, and *formal*, when needed, for more accurate reasoning.
- Goal formalization allows RE-specific types of analysis to be carried out, such as
  - guiding the goal refinement process and the systematic identification of objects and agents [Lam00b, Let02a];
  - checking the correctness of goal refinements and detecting missing goals and implicit assumptions [Dar96];
  - guiding the identification of obstacles and their resolutions [Lam00a];
  - guiding the identification of conflicts and their resolutions[Lam98];
  - guiding the identification and specification of operational requirements that satisfy the goals [Dar93, Let02b].

Several important topics are however not yet sufficiently addressed by current goal-oriented techniques.

- Current support for the evaluation and selection among multiple alternatives explored during the requirements elaboration process is highly limited. As discussed before, a blend of qualitative and quantitative reasoning techniques should be devised for more accurate evaluation of alternatives in terms of measurable quantities. Such techniques should probably be based on specific models for specific types of non-functional goals, e.g., risk models for safety goals, cost models for cost-related goals, performance models for performance-related goals, etc.
- Much work also remains to be done to provide *specialized* techniques for goal refinement and obstacle/conflict analysis that are targeted to *specific goal categories* relevant to high assurance systems (e.g., safety or security) and to specific domains (e.g., air traffic control, medical applications). This means characterizing and refining goal categories more thoroughly (maybe in domain-specific terms at some point), defining suitable notations and techniques for modeling and specifying properties in each category, and finding systematic ways of reasoning about their positive/negative interactions *at the goal level*.
- Further work is also needed to integrate the methodological support provided by our goal-oriented requirements engineering method with existing specification analysis tools. Such integration may occur at two levels. First, we would like to use existing tools to automate some of the *RE-specific* formal reasoning described above. For example, we recently built a tool prototype for early model checking of *goal* models; the generated counter-examples suggest inconsistent or missing goals. Second, we would like to map the result of goal-oriented requirements elaborations to specialized tools for formal analysis of operational specifications. For

example, we did a mapping of KAOS models to SCR tables [Del03]. Other mappings, e.g., to the NuSMV model checker (http://nusmv.irst.itc.it/) or the Alloy analyzer (http://sdg.lcs.mit.edu/alloy/), are under way.

Acknowledgement

The work of Emmanuel Letier was supported by the "Fonds National de la Recherche Scientifique" (FNRS). We are grateful to the KAOS/GRAIL crew at CEDITI for using some of the techniques presented here in industrial projects and to members of the FAUST project at CETIC for developing the (much needed) formal analysis toolkit.

References

[Bel76] T.E. Bell and T.A. Thayer, "Software Requirements: Are They Really a Problem?", *Proc. ICSE-2: 2nd Intrnational Conference on Software Enginering,* San Francisco, 1976, 61-68.

[Bha99] R. Bharadwaj and C. Heitmeyer, "Model Checking Complete Requirements Specifications Using Abstraction," *Automated Software Engineering*, Vol 6, No. 1, January 1999, 37-68.

[Boe81] B.W. Boehm, *Software Engineering Economics*. Prentice-Hall, 1981.

[Bro87] F.P. Brooks "No Silver Bullet: Essence and Accidents of Software Engineering". *IEEE Computer*, Vol. 20 No. 4, April 1987, pp. 10-19.

[Chu00] L. Chung, B. Nixon, E. Yu and J. Mylopoulos, *Non-functional requirements in software engineering*. Kluwer Academic, 2000.

[Cou93] P.J. Courtois and D.L. Parnas, "Documentation for Safety-Critical Software", *Proc. ICSE'1993: 15th International Conference on Software Engineering*, ACM Press, 1993, 315-323.

[Dar93] A. Dardenne, A. van Lamsweerde and S. Fickas, "Goal-Directed Requirements Acquisition", *Science of Computer Programming*, Vol. 20, 1993, 3-50.

[Dar96] R. Darimont and A. van Lamsweerde, "Formal Refinement Patterns for Goal-Driven Requirements Elaboration", *Proc. FSE'4 - Fourth ACM SIGSOFT Symp. on the Foundations of Software Engineering*, San Francisco, October 1996, 179-190.

[Del03] R. De Landtsheer, E. Letier and A. van Lamsweerde, "Deriving Tabular Event-Based Specifications from Goal-Oriented Requirements Models", *Proc. RE'03 - International Joint Conference on Requirements Engineering*, Monterey (CA), IEEE, September 2003. Expanded version to appear in the *Requirements Engineering Journal*.

[ESI96] European Software Institute, "European User Survey Analysis", Report USV_EUR 2.1, ESPITI Project, January 1996.

[Fea87] M. Feather, "Language Support for the Specification and Development of Composite Systems", *ACM Trans. on Programming Languages and Systems* 9(2), April 1987, 198-234.

[Fea98] M. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard, "Reconciling System Requirements and Runtime Behaviour", *Proc. IWSSD'98 - 9th International Workshop on Software Specification and Design*, Isobe, IEEE CS Press, April 1998.

[Fic92] S. Fickas and R. Helm, "Knowledge Representation and Reasoning in the Design of Composite Systems", *IEEE Trans. on Software Engineering*, June 1992, 470-482.

[Gar99] A. Gargantini and C. Heitmeyer, "Using Model Checking to Generate Tests from Requirements Specifications", *Proc., ESEC'99 & 7th ACM SIGSOFT Intern. Symp. on Foundations of Software Eng. (ESEC/FSE99)*, Toulouse, September 1999.

[Har87] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, vol. 8, 1987, 231-274.

[Heim96] M. Heimdahl and N.G. Leveson, "Completness and Consistency Checking in Hierarchical State-Based Requirements", *IEEE Transactions on Software Engineering*, Vol. 22, No. 6, June 1996, 363-377.

[Heit96] C. Heitmeyer, R. Jeffords and B. Labaw, "Automated Consistency Checking of Requirements Specifications", *ACM Transactions on Software Engineering and Methodology*, Vol. 5, No. 3, July 1996, 231-261.

[Jac95] M. Jackson, *Software Requirements & Specifications - A Lexicon of Practice, Principles and Prejudices*. ACM Press, Addison-Wesley, 1995.

[Jef98] R. Jeffords and C. Heitmeyer, "Automatic Generation of State Invariants from Requirements Specifications", *6th International Symposium on the Foundations of Software Engineering (FSE-6)*, Orlando FL, November 1998.

[Kni02] J.C. Knight, "Safety-Critical Systems: Challenges and Directions", Invited Mini-Tutorial, *Proc. ICSE'2002: 24th International Conference on Software Engineering*, ACM Press, 2002, 547-550.

[Lam98] A. van Lamsweerde, R. Darimont and E. Letier, "Managing Conflicts in Goal-driven Requirements Engineering", *IEEE Transactions on Software Engineering*, Special Issue on Inconsistency Management in Software Development, Vol. 24, No. 11, November 1998, 908-926.

[Lam00a] A. van Lamsweerde and E. Letier, "Handling Obstacles in Goal-Oriented Requirements Engineering", *IEEE Transactions on Software Engineering*, Special Issue on Exception Handling, Vol. 26, No. 10, October 2000, 978-1005.

[Lam00b] A. van Lamsweerde, "Requirements Engineering in the Year 00: A Research Perspective". Invited Keynote Paper, *Proc. ICSE'2000: 22nd International Conference on Software Engineering*, ACM Press, 2000, 5-19.

[Lam00c] A. van Lamsweerde, "Formal Specification: a Roadmap", in *The Future of Software Engineering*, A. Finkelstein (ed.), ACM Press, 2000.

[Lam03] A. van Lamsweerde, "From System Goals to Software Architecture", In *Formal Methods for Software Architecture*, M. Bernardo & P. Inverardi (eds.), LNCS 2804, Springer-Verlag, 2003.

[Lea95] J. McLean and C. Heitmeyer, "High Assurance Computer Systems: A Research Agenda", America in the Age of Information, National Science and Technology Council Committee on Information and Communications Forum, Bethesda, 1995.

[Let01] E. Letier, *Reasoning about Agents in Goal-Oriented Requirements Engineering*. PhD Thesis, Université Catholique de Louvain, Dépt. Ingénierie Informatique, Louvain-la-Neuve, Belgium, May 2001. http://www.info.ucl.ac.be/people/eletier/thesis.html

[Let02a] E. Letier and A. van Lamsweerde, "Agent-Based Tactics for Goal-Oriented Requirements Elaboration", *Proc. ICSE'02: 24th Intl. Conf. on Software Engineering*, Orlando, IEEE Computer Society Press, May 2002.

[Let02b] E. Letier and A. van Lamsweerde, "Deriving Operational Software Specifications from System Goals", *Proc. FSE'10:* 10th *ACM SIGSOFT Symp. on the Foundations of Software Engineering*, Charleston, November 2002.

[Let02c] E. Letier, *Goal-Oriented Elaboration of Requirements for a Safety Injection Control System*. Research Report, Département d'Ingénierie Informatique, UCL, June 2002.

[Lev95] N. Leveson, *Safeware: System Safety and Computers.* Addison-Wesley, 1995.

[Lut93] R. Lutz, "Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems", *Proc. RE'93: First IEEE International Symposium on Requirements Engineering*, January 1993, 126-133.

[Man96] Z. Manna and the STep Group, "STeP: Deductive-Algorithmic Verification of Reactive and Real-Time Systems", *Proc. CAV'96 - 8th Intl. Conf. on Computer-Aided Verification*, LNCS 1102, Springer-Verlag, July 1996, pp. 415-418.

[Sta95] The Standish Group, "Software Chaos", http:// www.standishgroup.com/chaos.html.

[Yue87] K. Yue, "What Does It Mean to Say that a Specification is Complete?", *Proc. IWSSD-4, Fourth International Workshop on Software Specification and Design*, Monterey, 1987.

[Zav97] P. Zave and M. Jackson, "Four dark corners of requirements engineering", *ACM Trans. on Software Engineering and Methodology*, Vol. 6, No. 1, January 1997, 1 - 30.

View Consistency in Software Development*

Martin Wirsing and Alexander Knapp

Ludwig–Maximilians–Universität München
{wirsing,knapp}@informatik.uni-muenchen.de

Abstract. An algebraic approach to the view consistency problem in software development is provided. A view is formalised as a sentence of a viewpoint language; a viewpoint is given by a language and its semantics. Views in possibly different viewpoints are compared over a common view for consistency by a heterogenous pull-back construction. This general notion of view consistency is illustrated by several examples from viewpoints used in object-oriented software development.

1 Introduction

The use of views in software development supports an often desirable "separation of concerns". Each stakeholder of a software system may express his view of the system from his own viewpoint and may employ the notation most appropriate for this viewpoint. In particular, most viewpoints taken by system stakeholders concentrate on parts of the whole system under construction which may either be rather orthogonal and separated by clean interfaces, or may overlap in intricate ways. However, the use of different views in software development poses the problem to ensure consistency, i.e., to guarantee that there is an overall integration of the views that is implementable in a software product. On the one hand, this means to integrate partial descriptions of the system; on the other hand, different notations and their semantics have to be compared.

The "system-model" solution to the view consistency problem embeds all viewpoints used for software development in a single, unifying system model and compares the embedded views over the system model's semantics. This approach has been put forward, for example, by stream-based [6], graph grammar [7] and rewrite system models [13], or the integration of different specification formalisms, like CSP and Z [8, 19] or a combination of algebraic specifications and labelled transition systems [12, 16]. However, an encompassing single system model renders reasoning on different views in a formalism suitably adapted to the view's viewpoint rather difficult. The "heterogeneous-specification" line of research concentrates on the comparison and integration of different, heterogeneous specification formalisms, retaining the formalisms most appropriate for expressing parts of the overall problem. Most prominently, institutions [9] and general logics [11] are used as a formal basis establishing a powerful framework for heterogenous specifications and heterogenous proofs [2, 15, 14]. These investigations, which concentrate on formal, logic-based software development, is complemented by set-based frameworks for view comparison and integration [3, 5].

* This research has been partially sponsored by the DFG project WI 841/6-1 "InOpSys".

M. Wirsing et al. (Eds.): RISSEF 2002, LNCS 2941, pp. 341–357, 2004.

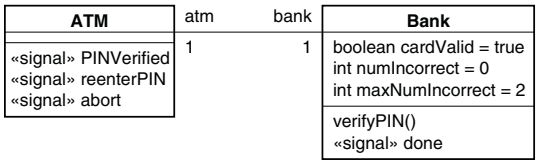
Our approach to the view consistency and integration problem in software development follows the "heterogeneous specification" approach, but applies these ideas to general software development. A viewpoint is given by a language and its semantics; a view in a viewpoint is a sentence of the viewpoint's language. We introduce a suitable notion of translations between viewpoints and view consistency for views in different viewpoints. We also define a general notion of consistency of views in possibly different viewpoints over a common view in maybe yet another viewpoint using a heterogenous pull-back construction. In particular, this construction avoids the unification of all different viewpoints into a single, formal system model. We illustrate our notion of consistency by several examples for viewpoints used in common object-oriented software development such as class diagrams and state machine diagrams.

The remainder of this paper is structured as follows: Sect. 2 motivates the use of multiple views and the consistency problem by means of examples. Sect. 3 introduces our algebraic notion of viewpoints and views. The translation of viewpoints and views is defined in Sect. 4. The algebraic notion of view consistency is presented in Sect. 5. We conclude by an outlook to further research topics. The appendices contain the formal definitions for the viewpoints used in the examples.

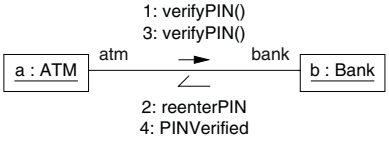
2 Multiple Views

In software development, multiple views and viewpoints are ubiquitous. As a simple example, consider the description of the interaction of an automatic teller machine (ATM) with a bank in Fig. 1. Using the "Unified Modeling Language" (UML [4]), the static structure of such a system may be specified by a UML class diagram as in Fig. 1(a). The dynamic behaviour of instances of the classes ATM and Bank may be given by state machine diagrams, see Fig. 1(d) for an ATM, Fig. 1(e) for a Bank. Finally, collaboration diagrams may be used for specifying desired (cf. Fig. 1(b)) or undesired behaviours (cf. Fig. 1(c)) of interaction.

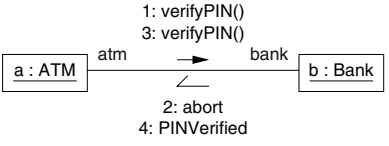
These views overlap and thus must be checked for consistency in several ways: First of all, the static structure and both of the state machines can be considered consistent if the state machines refer only to attributes, association ends, operations, and receptions that are declared in the class diagram. This syntactical notion of consistency amounts to extracting a class diagram from the state machines and checking whether this extracted class diagram is contained in the given class diagram of the static structure. In the same way, the collaboration diagrams can be checked for their class-diagram compatibility and, moreover, the same signature check must be applied to the collaborations and the state machines. Thus, when comparing the diagrams from a class-diagram or signature viewpoint, there must be translations from the views and their viewpoints under comparison into the signature viewpoint where consistency checking is signature inclusion. The same technique of consistency checking applies for showing that a collaboration is indeed realised by interacting state machines. As a collaboration specifies possible message exchanges and their order, the message exchanges of the interacting state machines have to be compared to these possible partial orders of message exchanges for inclusion. Hence, comparing diagrams from an interaction viewpoint involves translations from the views and their viewpoints under comparison into the interaction viewpoint



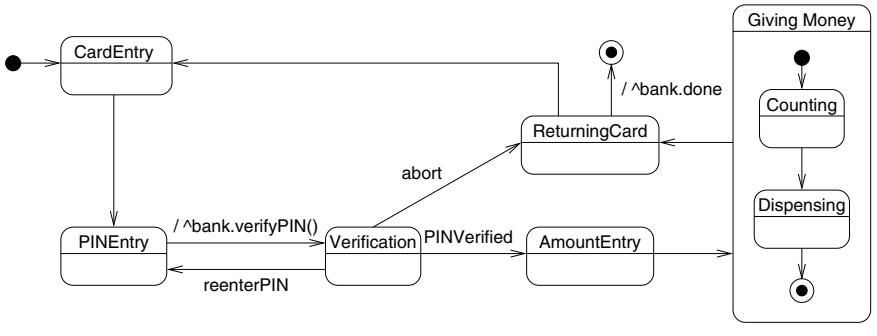
(a) Class diagram



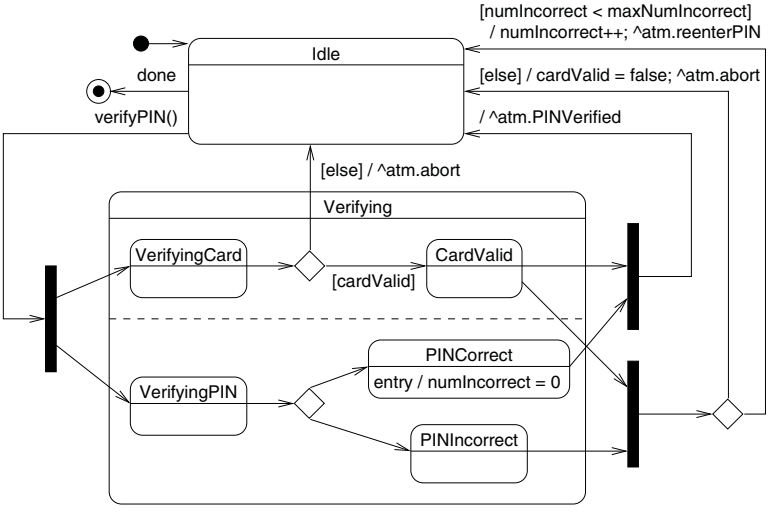
(b) Expected collaboration



(c) Erroneous collaboration



(d) State machine diagram for class `ATM`



(e) State machine diagram for class `Bank`

Fig. 1. Multiple views in a UML model of an ATM.

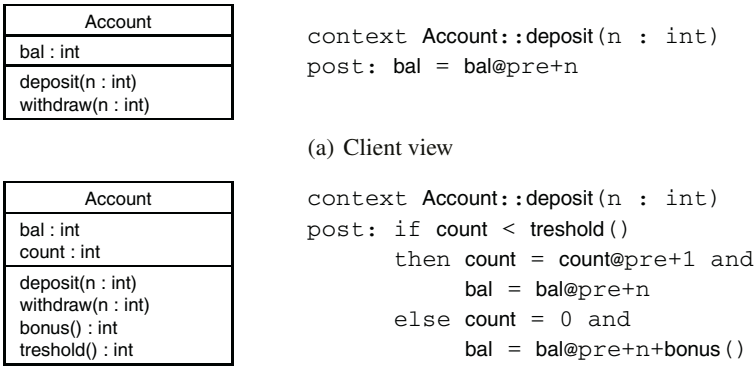


Fig. 2. Multiple views in UML models of a bank account.

where consistency checking is partial order inclusion. Similarly, the interaction viewpoint can be used to check that message exchanges given by a collaboration diagram are not included in the message exchanges of interacting state machines.

An analogous approach applies when a system is viewed from different angles by different stakeholders. Consider the (over-simplified) descriptions of a bank account depicted in Fig. 2, using UML and the "Object Constraint Language" (OCL [18]) as a pre-/postcondition language: A client may take the straightforward view of an account represented in Fig. 2(a), whereas the management of the bank may want to apply a special bonus rule for frequent customers thus taking the view in Fig. 2(b).

As in the previous example, several viewpoints are employed in both descriptions. From the structural viewpoint, UML class diagrams are used for specifying the static structure of a system, from the behavioural viewpoint the desired behaviour is expressed in terms of OCL pre-/postconditions on operations. Thus, taking the views as heterogenous views combined from views in the structural and the behavioural viewpoint, both the management and the client view can be checked for internal consistency. Furthermore, the views in the structural viewpoint may be considered to be consistent as the structural view in the management view simply extends the structural view in the client view. However, the behavioural views of the client and the management on the behaviour of the operation Account::deposit are inconsistent, as the client specification does not take into account the bonus feature of the management view.

3 Viewpoints and Views

Generalising from the examples above, a viewpoint of software development consists of a syntactic domain, a semantic domain, and a mapping from the syntax into the semantics. The syntactic domain captures the viewpoint notation, most conveniently in terms of an abstract syntax. A sentence of this abstract syntax conveys the information expressed in a view according to the viewpoint's notation. The semantic domain defines appropriate models for the abstract syntax.

More concretely, a *viewpoint* $\mathscr{V}$ is represented by a formal language category L, a semantic domain category D, and a semantic functor $\mathrm{Mod} : L^{\mathrm{op}} \to D$. A *view* in a viewpoint $\mathscr{V} = (L, D, \mathrm{Mod})$ is a specification object V in the formal language category L; its semantics, i.e. its models, is given by $\mathrm{Mod}(V) \in D$. Note that we choose the opposite language category L^{op} for the semantics in order to express the well-known contravariance of syntax and semantics in logic and model-theory [1].

Example 3.1. The *structural viewpoint* $Struct$, cf. Fig. 1(a) and the class diagrams in Fig. 2(a) and 2(b) for concrete examples and App. B.1 for a detailed definition, uses the language $StructDiag$ of structure or class diagrams, comprising classes with typed attribute and method signatures and the (binary) associations between classes. As the semantic domain $StructAlg$ the class of many-sorted algebras over the many-sorted algebraic signatures induced from structure diagrams is employed. The signature of a structure diagram consists of sorts from the classes and operations from the attributes, methods, and associations. The model functor Mod_{Struct} maps a structure diagram to the algebras over its induced signature.

Example 3.2. The *behavioural viewpoint* Beh, cf. Fig. 2(a) and 2(b) for concrete examples and App. B.2 for a detailed definition, uses the language $BehSpec$ of pre-/postconditions for annotating methods of classes. As the semantic domain $BehAlg$ the class of many-sorted algebras over the many-sorted algebraic signature induced from pre-/postcondition annotations is employed. The signature of pre-/postconditions consists of sorts for the classes and operations from the attributes and associations. The model functor Mod_{Beh} maps a pre-/postcondition specification to the algebras over its induced signature such that the methods applied to a state satisfy the pre-/postconditions.

Example 3.3. The *instance viewpoint* $Inst$, cf. App. B.3 for a detailed definition, uses the language $InstDiag$ of object diagrams, comprising typed objects with typed slots and their values and (binary) links between objects. As the semantic domain $InstAlg$ the class of many-sorted algebras over the many-sorted algebraic signatures induced from object diagrams is employed. The signature of an object diagram consists of sorts for the types of the objects, operations from the attributes and links, and constant operations for the objects. The model functor Mod_{Inst} maps an object diagram to the algebras over its induced signature such that the valuations of the slots and the links are satisfied.

Example 3.4. The *interaction viewpoint* $Inter$, cf. Fig. 1(b) and 1(c) for concrete examples and App. B.4 for a detailed definition, uses the language $InterDiag$ of collaboration diagrams, comprising objects and a partial order of message exchanges between these objects. As the semantic domain $InterAlg$ the class of algebras representing partial orders of messages between objects is employed. The model functor Mod_{Inter} maps a collaboration diagram to the class of partial orders of messages between objects that contain at least the partial order of message exchanges specified in the collaboration.

Example 3.5. The *machine viewpoint* $Mach$, cf. Fig. 1(d) and 1(e) for concrete examples and App. B.5 for a detailed definition, uses the language $MachDiag$ of state machine diagrams, comprising classes with their attributes and methods, and a mapping of classes to state machines. Each state machine is given by a set of states, an initial

state, and a set of transitions. Each transition is annotated with an event accepted by the transition, its effects and messages that are sent when the transition is taken. As the semantic domain $MachAlg$ the class of algebras representing the possible transitions of the state machines is employed. The model functor Mod_{Mach} maps a state machine diagram to the class of algebras with transitions for the machines in the diagram.

4 Translation

Viewpoints and thus views may be related by translations: Certain information of a view can be extracted and reformulated in another viewpoint. Translations may well induce partial loss of information, as not all viewpoints carry comparable data and not all viewpoints allow the specifier to express a software design at the same level of detail. In particular, translation may not always be possible syntactically. However, whenever information is shared between viewpoints, such as information on types in the structural, instance, interaction, and machine viewpoints, translations afford the necessary extraction mechanism.

A translation transfers information from one viewpoint $\mathscr{V}_1 = (L_1, D_1, \mathrm{Mod}_1)$ into another viewpoint $\mathscr{V}_2 = (L_2, D_2, \mathrm{Mod}_2)$. A *syntactic* viewpoint translation τ from $\mathscr{V}_1$ to $\mathscr{V}_2$ uses viewpoint notations: $\tau : L_1 \to L_2$. Corresponding to the contravariant behaviour of the model functors, a *semantic* viewpoint translation μ from $\mathscr{V}_1 \to \mathscr{V}_2$ operates contravariantly on the semantic domains of the viewpoints: $\mu : D_2 \to D_1$. A viewpoint translation from $\mathscr{V}_1$ to $\mathscr{V}_2$ consists of a pair of syntactic and semantic viewpoint translations $(\tau, \mu) : \mathscr{V}_1 \to \mathscr{V}_2$.

Example 4.1. We consider the translation of the instance viewpoint $Inst$ into the structural viewpoint $Struct$. Syntactically, a class diagram can be induced from an object diagram by turning type names, slots, links, and links ends into classes, attributes, associations, and association ends. Thus we have a syntactic viewpoint translation $\tau : InstDiag \to StructDiag$. As an algebra in $StructAlg$ always can be interpreted as an algebra in $InstAlg$ by forgetting the methods, we also have a semantic viewpoint translation $\mu : StructAlg \to InstAlg$.

Example 4.2. There is also a syntactical translation of the structural viewpoint $Struct$ into the instance viewpoint $Inst$: $\tau : StructDiag \to InstDiag$ can be trivially defined by translating each structure diagram into the empty object diagram. However, there is a more natural semantic translation from the structural viewpoint into the instance viewpoint $\mu : InstAlg \to StructAlg$ by forgetting the additional object constants.

Example 4.3. In a similar way to translating the instance viewpoint into the structural viewpoint, there is a syntactical translation $\tau : InstDiag \to InterDiag$ of the instance viewpoint $Inst$ into the interaction viewpoint $Inter$: The objects are kept, but the interaction diagram will contain no messages. Conversely, the interaction viewpoint can be translated syntactically into the instance viewpoint, $\tau : InterDiag \to InstDiag$, by keeping the objects and adding links between objects which exchange messages.

Example 4.4. The examples 4.1 and 4.3 can be combined into a syntactic translation $\tau : Inter \to Struct$ which infers the classes and associations from the interaction

diagram. However, there is also a direct syntactic translation from $Inter$ to $Struct$ that also keeps the messages.

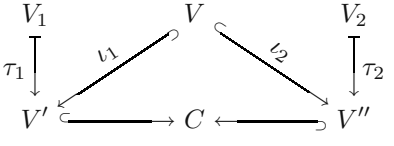
Example 4.5. A syntactic translation of the interaction viewpoint into the machine viewpoint can be achieved as follows: Each class of an object in the interaction is associated with a machine that accepts the messages incoming to the object. An incoming message is answered by all following outgoing messages [10]. This syntactic translation is complemented by a a semantic translation of the interaction viewpoint $Inter$ into the machine viewpoint $Mach$, i.e. $\mu : MachAlg \to InterAlg$. The machines in a machine algebra are run concurrently from their initial states. Each machine that fires a transition sends messages to the other machines. The receiving machine reacts to the incoming event by subsequently firing a transition. Thus, every run of the machines induces a partial order of exchanged messages.

5 Consistency

Given the notions of viewpoints, views, and viewpoint translations introduced above, the view consistency problem in software development amounts to relating views in different viewpoints by syntactical or semantical translations, and checking whether the overlapping parts of the views are acceptable to the software engineers. In particular, views may be consistent from certain viewpoints, but deemed to be inconsistent from others. Therefore it seems advisable to introduce a point of comparison between two views. This point of comparison can be chosen as a view in the viewpoint from one of the views to be related, or may be of yet another viewpoint. Moreover, the point of comparison view can specify the minimal requirements for the compared views or can embrace all the information, relative to the chosen viewpoint, that is expected to be available in the views to be compared. Finally, views can be compared syntactically involving syntactic translations, or semantically using semantic translations.

5.1 Syntactic Consistency

The syntactic consistency check for two views over a common point of comparison view necessitates the syntactic translation of the views under comparison into the viewpoint of the point of comparison. We call two views *consistent* over a common view if there are embeddings of the common view in the translated versions of the views under comparison. More precisely, when comparing the views V_1 and V_2 in viewpoints $\mathscr{V}_1 = (L_1, D_1, \mathrm{Mod}_1)$ and $\mathscr{V}_2 = (L_2, D_2, \mathrm{Mod}_2)$, respectively, over a point of comparison view V in a viewpoint $\mathscr{V} = (L, D, \mathrm{Mod})$ we require that there are syntactic viewpoint translations $\tau_1 : \mathscr{V} \to \mathscr{V}_1$, $\tau_2 : \mathscr{V} \to \mathscr{V}_2$ from the viewpoint $\mathscr{V}$. Given these translations there must be embeddings $\iota_1 : V \to \tau_1(V_1)$ and $\iota_2 : V \to \tau_2(V_2)$. If D admits push-outs, we obtain the following diagram:

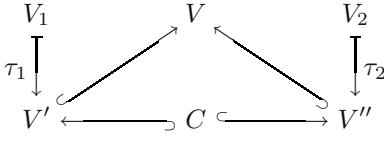


Here, the point of comparison view V is *shared* between the (translated) views V_1 and V_2, that is, V contains symbols that should have the same meaning in V_1 and V_2.

Example 5.1. Considering the structure diagram S from Fig. 2(b) and the behavioural specifications B_1 and B_2 in Fig. 2(a) and Fig. 2(b), syntactic consistency of B_1 and B_2 over S can be seen from the translations of behavioural specifications into structure diagrams: The additional pre-/postconditions are forgotten, thus the push-out in $StructDiag$ is isomorphic to S. Note that a comparison of B_1 and B_2 over the structure diagram from Fig. 2(a) would result in an enrich structure diagram also containing bonus and threshold.

In general two behavioural specifications B_1 and B_2 are consistent over a structure diagram S if, and only if all classes, attribute signatures, operation signatures, and association end signatures of S are contained in both behavioural specifications. The push-out constructs the union of all features of B_1 and B_2 separately renaming the features in B_1 and B_2 that are not present in S.

As the example illustrates it may sometimes be desirable to require that the point of comparison already contains all information that can be deduced from the views under comparison by translating these views into the common viewpoint. Such a point of comparison view is called *embracing* and leeds to a similar consistency diagram as for the shared case, but with arrows reversed such that C now becomes a pull-back in D:

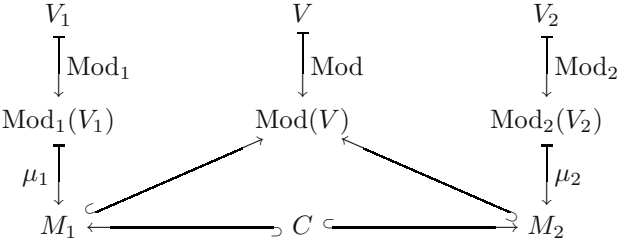


Example 5.2. We consider the comparison of the collaboration views I_1 and I_2 in Fig. 1(b) and Fig. 1(c) over the embracing point of comparison view S in Fig. 1(a). Translating I_1 and I_2 into structure diagrams, there are embeddings of these translated interaction diagrams into S. Therefore I_1 and I_2 are syntactically consistent with respect to S. The pull back contains only the classes ATM and Bank with their respective methods PINVerified and verifyPIN.

5.2 Semantic Consistency

The semantic consistency check for two views over a comparison view involves the construction of a common model such that the comparison view is extended by the compared views consistently. The construction is dual to syntactic consistency. However, the existence of embeddings is sufficient to assess whether views are consistent or not, as embeddings may also exist for semantically inconsistent views. We therefore have to inspect the pull-back, i.e., roughly speaking, the intersection, of the model classes of the views. If the pull-back is empty, then the views are inconsistent; if it is not empty, the views are formally consistent, but the feature interaction of the views may still result in undesired behaviour which can be revealed by inspecting the models of the pull-back.

More specifically, let V_1 and V_2 be views in the viewpoints $\mathscr{V}_1 = (L_1, D_1, \mathrm{Mod}_1)$ and $\mathscr{V}_2 = (L_2, D_2, \mathrm{Mod}_2)$, respectively, and let V be a (shared) point of comparison view in viewpoint $\mathscr{V} = (L, D, \mathrm{Mod})$. We say that V_1 and V_2 are *semantically consistent* with respect to V, if there are semantic viewpoint translations $\mu_1 : \mathscr{V}_1 \to \mathscr{V}$, $\mu_2 : \mathscr{V}_2 \to \mathscr{V}$ from the viewpoint $\mathscr{V}$ and there exists a pull-back C in D such that the following diagram commutes:



Note that again the common view provides the viewpoint of the integrating model. The commutation of the diagram above implies that V is a shared view between V_1 and V_2. The opposite case, that V embraces all information that can be extracted by viewpoint translations from V_1 and V_2 leads to a diagram where the embedding arrows between V_1' and V, and V_2' and V have to be reversed.

Example 5.3. The semantical consistency of the behavioural specifications B_1 and B_2 in Fig. 2(a) and Fig. 2(b) over the structure diagram S from Fig. 2(b) can be checked as follows: The models of B_1 are $\mathrm{Sig}_{Beh}(B_1) = \langle\{\mathsf{Account}, \mathsf{int}, \mathrm{State}\}, \{\mathsf{bal} : \mathrm{State} \times \mathsf{Account} \to \mathsf{int}, \ldots\}\rangle$-algebras subject to the behavioural specification `post: bal = bal@pre+n`, the models of B_2 are $\mathrm{Sig}_{Beh}(B_1)$-algebras subject to the behavioural specification `post: if count < threshold() ... endif`. But the models of B_1 and B_2 in the behavioural viewpoint Beh can be translated into models in the structural viewpoint $Struct$ forgetting the additional sorts and operations. These translated models can be trivially embedded in $\mathrm{Mod}(S) = \mathrm{Sig}_{Struct}(S)$-$Alg$. The pull-back in $StructAlg$ turns out to be the category of algebras over the amalgamated sum of the signatures of B_1 and B_2 over the signature of S where all symbols from S are shared. In particular, the pull-back is not empty and thus B_1 and B_2 are semantically consistent over S. However, this formal consistency may be misleading as bonus^A is identically 0 for $A \in |C|$ or $\mathsf{count}^A(s,a) <^A \mathsf{threshold}^A(s,a)$ for all $s \in \mathrm{State}^A$, $a \in \mathsf{Account}^A$; this may not be desirable. Moreover, if a software designer requires bonus to be greater than zero, the views become inconsistent.

More generally, in the case of behavioural specifications the semantic consistency, i.e. the calculation of pull-backs, can be reduced to logical consistency by computing the conjunction of the theories of the views under consideration (modulo renamings required by the shared view).

Example 5.4. A comparison of the machine view A with the state machines in Fig. 1(d) and 1(e) with the interaction view I of the collaboration in Fig. 1(b) leaves several possiblities for choosing a common point of comparison view. We employ the empty interaction view $\emptyset$ as the shared view. The translation of $\mathrm{Mod}_{Mach}(A)$ into the interaction viewpoint amounts to all possible interaction sequences of the state machines. The pull-

back C now contains all partial orders of message exchanges that are part of the partial orders in $\mathrm{Mod}_{Inter}(I)$ and the partial orders from the translation of $\mathrm{Mod}_{Mach}(A)$.

In general, an interaction diagram is consistent with a machine diagram w. r. t. the empty interaction diagram if there is a run of the collaboration diagram that is also a run of the state machines. In the case of finite state systems, like the ATM example, this property can be checked efficiently by model checking [17].

6 Conclusions

We have presented an algebraic framework for view consistency in software development. The framework is inspired by the institution-based approaches to heterogeneous specifications. Viewpoints are formalised as a pair of a syntactic and a semantic category linked by a model functor. Views are objects in the syntactic category. Consistency of views is defined by a heterogeneous pullback construction.

However, view consistency is merely a stepping stone to the successful employment of different views in software development. Views on a system will evolve over the software life-cycle, some may extend over all software development phases, some may be replaced, refined, or be combined during construction. Taking views and viewpoints serious hence in particular means to provide further support for separation of concerns by view maintainance allowing the different stakeholders to keep their viewpoints of the system. Correct view development, replacement, and refinement remain a challenge.

References

1. Jon Barwise, editor. *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1977.
2. Gilles Bernot, Sophie Coudert, and Pascale Le Gall. Towards Heterogenous Formal Specifications. In Martin Wirsing and Maurice Nivat, editors, *Proc. 5th Int. Conf. Algebraic Methodology and Software Technology*, volume 1101 of *Lect. Notes Comp. Sci.*, pages 458–472. Springer, Berlin, 1996.
3. Eerke A. Boiten, Howard Bowman, John Derrick, Peter F. Linington, and Maarten Steen. Viewpoint consistency in ODP. *Computer Networks*, 34(3):503–537, 2000.
4. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison–Wesley, Reading, Mass., &c., 1998.
5. Howard Bowman, Maarten W. A. Steen, Eerke A. Boiten, and John Derrick. A Formal Framework for Viewpoint Consistency. *Formal Methods in System Design*, 21:111–166, 2002.
6. Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems:* FOCUS *on Streams, Interfaces, and Refinement*. Springer, Berlin, 2001.
7. Gregor Engels, Reiko Heckel, Gabi Taentzer, and Hartmut Ehrig. A Combined Reference Model- and View-Based Approach to System Specification. *Int. J. Softw. Knowl. Eng.*, 7(4):457–477, 1997.
8. Clemens Fischer. CSP-OZ: How to Combine Z with a Process Algebra. In Howard Bowman and John Derrick, editors, *Proc. 2nd Int. Conf. Formal Methods for Open Object-Based Distributed Systems*, volume 2, pages 423–438. Chapman & Hall, Boston, 1997.
9. Joseph A. Goguen and Rod M. Burstall. Institutions: Abstract Model Theory for Specification and Programming. *J. ACM*, 39(1):95–146, 1992.

10. Ingolf Krüger, Radu Grosu, Peter Scholz, and Manfred Broy. From MSCs to Statecharts. In Franz J. Rammig, editor, *Distributed and Parallel Embedded Systems*, pages 61–71. Kluwer Academic, Boston–Dordrecht, 1999.
11. José Meseguer. General Logics. In Heinz-Dieter Ebbinghaus, José Fernández-Prida, Manuel Garrido, Daniel Lascar, and Mario Rodríguez Artalejo, editors, *Proc. Logic Colloquium '87*, volume 129 of *Studies in Logic and the Foundatons of Mathematics*, pages 275–329. Elsevier, North Holland, Amsterdam, 1989.
12. José Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theo. Comp. Sci.*, 96:73–155, 1992.
13. José Meseguer. Formal Interoperability. In *Proc. 5th Int. Symp. Mathematics and Artificial Intelligence*, Fort Laurerdale, Florida, 1998. 9 pages.
14. Till Mossakowski. Heterogenous Development Graphs and Heterogeneous Borrowing. In Mogens Nielsen and Uffe Engberg, editors, *Proc. 5th Int. Conf. Foundations of Software Science and Computation Structures*, volume 2303 of *Lect. Notes Comp. Sci.*, pages 326–341. Springer, Berlin, 2002.
15. Till Mossakowski, Andrzej Tarlecki, and Wiesław Pawłowski. Combining and Representing Logical Systems Using Model-Theoretic Parchments. In Francesco Parisi Presicce, editor, *Sel. Papers 12th Int. Wsh. Algebraic Development Techniques*, volume 1376 of *Lect. Notes Comp. Sci.*, pages 349–364. Springer, Berlin, 1998.
16. Gianna Reggio and Luigi Repetto. CASL-CHART: A Combination of Statecharts and the Algebraic Specification Language CASL. In Teodor Rus, editor, *Proc. 8th Int. Conf. Algebraic Methodology and Software Technology*, volume 1816 of *Lect. Notes Comp. Sci.*, pages 243–272. Springer, Berlin, 2000.
17. Timm Schäfer, Alexander Knapp, and Stephan Merz. Model Checking UML State Machines and Collaborations. In Scott Stoller and Willem Visser, editors, *Proc. Wsh. Software Model Checking*, volume 55(3) of *Elect. Notes Theo. Comp. Sci.*, Paris, 2001. 13 pages.
18. Jos Warmer and Anneke Kleppe. *The Object Constraint Language*. Addison–Wesley, Reading, Mass., &c., 1999.
19. Heike Wehrheim. *Behavioural Subtyping in Object-Oriented Specification Formalisms*. Habilitationsschrift, Carl-von-Ossietzky Universität Oldenburg, 2002.

A Many-Sorted Algebras

Signatures. A (many-sorted) *signature* $\langle S, F\rangle$ consists of a set of *sort* symbols S and a set of *operations* F of the form $f : (s_i)_{1\leq i\leq k} \to s_0$ ($k \in \mathbb{N}$) with f an operation symbol and $s_0, s_1, \ldots, s_k \in S$. We require all sort symbols and all operation symbols to be distinct.

A *signature morphism* $\sigma : \langle S, F\rangle \to \langle S', F'\rangle$ is given by a map σ from the symbols of $\langle S, F\rangle$ to the symbols of $\langle S', F'\rangle$ such that a sort symbol is mapped to a sort symbol and an operation symbol is mapped to an operation symbol; and the following condition holds: Given an operation $f : (s_i)_{1\leq i\leq k} \to s_0 \in F$ the image $\sigma(f) : (\sigma(s_i))_{1\leq i\leq k} \to \sigma(s_0)$ is in F'.

The category *Sig* has as objects: signatures, and as morphisms: signature morphisms. The composition of signature morphisms is defined as function composition; the identity morphism on a signature is given by the identity on the signature's symbols.

Algebras. Given a (many-sorted) signature $\Sigma = \langle S, F\rangle$, a (many-sorted) *$\Sigma$-algebra* A consists of a family of *universes* $(s^A)_{s\in S}$ and a family of functions $(f^A)_{f\in F}$ such that $f^A : (s_i^A)_{1\leq i\leq n} \to s_0^A$ for $f : (s_i)_{1\leq i\leq n} \to s_0 \in F$.

A *Σ-algebra morphism* $\alpha : A \to A'$ for a signature $\Sigma = \langle S, F\rangle$ and algebras A and A' is given by a family of functions $(\alpha_s : s^A \to s^{A'})_{s\in S}$ such that $\alpha_{s_0}(f^A(u_1, \ldots u_n)) = f^{A'}(\alpha_{s_1}(u_1), \ldots, \alpha_{s_n}(u_n))$ for $f : (s_i)_{1\le i\le n} \to s_0 \in F$.

The category Σ-Alg has as objects: Σ-algebras, and as morphisms: Σ-algebra morphisms. The composition of Σ-algebra morphisms is defined as function composition; the identity morphism on a Σ-algebra is given by the family of identity functions on its universes.

A signature morphism $\sigma : \Sigma \to \Sigma'$ for signatures $\Sigma = \langle S, F\rangle$ and $\Sigma' = \langle S', F'\rangle$ induces a functor $\restriction_\sigma : \Sigma'$-$Alg \to \Sigma$-$Alg$ such that, for an $A' \in |\Sigma'\text{-}Alg|$, $s^{A'\restriction_\sigma} = \sigma(s)^{A'}$ for $s \in S$ and $f^{A'\restriction_\sigma} = \sigma(f)^{A'}$ for $f \in F$; and $(\alpha' : A' \to B')\restriction_\sigma = \alpha : A'\restriction_\sigma \to B'\restriction_\sigma$ where $(\alpha_s)_{s\in S} = (\alpha'_{\sigma(s)})_{s\in S}$.

The category Alg has as objects: the categories Σ-Alg for signatures Σ, and as morphisms: for every signature morphism $\sigma : \Sigma \to \Sigma'$ the reduct functor $\restriction_\sigma$. The composition of Alg-morphisms is defined as functor composition; the identity morphism on the category Σ-Alg is given by the reduct functor from the identity signature morphism $\mathrm{id} : \Sigma \to \Sigma$.

B Viewpoints

B.1 Structural Viewpoint

Syntax. A *structure diagram* $\langle C, A\rangle$ consists of a set of classes C and a set of associations A. Each *class* $\gamma \in C$ is given by its (unique) name; a set of *attributes* of the form $a : \tau$ with a a (unique) name and τ the name of a class in C; and a set of *methods* of the form $m : (\tau_i)_{1\le i\le k} \to \tau_0$ ($k \in \mathbb{N}$) with m a (unique) name and $\tau_0, \tau_1, \ldots, \tau_k$ names of classes in C. Each *association* $\alpha \in A$ is given by a pair of *association ends* of the form $\langle a_1 : \tau_1, a_2 : \tau_2\rangle$ with a_1, a_2 (unique) names and τ_1, τ_2 names of classes in C.

A *structure diagram morphism* $\sigma : \langle C, A\rangle \to \langle C', A'\rangle$ is given by a map σ from the names of $\langle C, A\rangle$ to the names of $\langle C', A'\rangle$ such that a class name is mapped to a class name, an attribute name is mapped to an attribute name, &c; and the following conditions hold: (1) Given a class $\gamma \in C$ with name n, attributes $a_1 : \tau_1, \ldots, a_k : \tau_k$, and methods $m_1 : (\tau_{1i})_{1\le i\le k_1} \to \tau_{10}, \ldots, m_l : (\tau_{li})_{1\le i\le k_l} \to \tau_{l0}$ the class $\gamma' \in C'$ denoted by $\sigma(n)$ has at least the attributes $\sigma(a_1) : \sigma(\tau_1), \ldots, \sigma(a_k) : \sigma(\tau_k)$ and at least the methods $\sigma(m_1) : (\sigma(\tau_{1i}))_{1\le i\le k_1} \to \sigma(\tau_{10}), \ldots, \sigma(m_l) : (\sigma(\tau_{li}))_{1\le i\le k_l} \to \sigma(\tau_{l0})$; (2) given an association $\langle a_1 : \tau_1, a_2 : \tau_2\rangle \in A$ there is an association $\langle \sigma(a_1) : \sigma(\tau_1), \sigma(a_2) : \sigma(\tau_2)\rangle \in A'$.

The category $StructDiag$ has as objects: structure diagrams, and as morphisms: structure diagram morphisms. The composition of structure diagram morphisms is defined as function composition; the identity morphism on a structure diagram is given by the identity on the structure diagram's names.

Signature. The *signature* of a structure diagram $\mathrm{Sig}(\langle C, A\rangle) = \langle S, F\rangle$ is defined as follows: (1) S contains a distinguished sort State. (2) Every class name in C gives rise to a sort symbol in S. (3) An attribute $a : \tau$ of a class with name γ gives rise to an operation $a : \mathrm{State} \times \gamma \to \tau$ in F. (4) A method $m : (\tau_i)_{1\le i\le k} \to \tau_0$ of a class with name γ gives rise to operations $m : \mathrm{State} \times \gamma \times (\tau_i)_{1\le i\le k} \to \tau_0$ and

$m\text{state} : \text{State} \times \gamma \times (\tau_i)_{1 \leq i \leq k} \to \text{State}$ in F. (5) An association $\langle a_1 : \tau_1, a_2 : \tau_2 \rangle$ in A gives rise to the operations $a_1 : \text{State} \times \tau_2 \to \tau_1$ and $a_2 : \text{State} \times \tau_1 \to \tau_2$ in F. (6) S and F are the least such sets under inclusion.

As each structure diagram morphism naturally induces a signature morphism, the signature mapping on objects of *StructDiag* can be extended to a functor $\text{Sig} : \mathit{StructDiag} \to \mathit{Sig}$.

Semantics. The *semantics* $\text{Mod}(\langle C, A \rangle)$ of a structure diagram $\langle C, A \rangle$ is given by $\text{Sig}(\langle C, A \rangle)$-*Alg*. The full sub-category of *Alg* induced by the image of Mod is called *StructAlg*. This semantics on objects of *StructDiag* is lifted to a functor $\text{Mod} : \mathit{StructDiag}^{\text{op}} \to \mathit{StructAlg}$ by setting $\text{Mod}(\sigma^{\text{op}} : \langle C', A' \rangle \to \langle C, A \rangle) = \restriction_{\text{Sig}(\sigma)} : \text{Mod}(\langle C', A' \rangle) \to \text{Mod}(\langle C, A \rangle)$.

B.2 Behavioural Viewpoint

Syntax. A *behaviour specification* $\langle C, \text{pre}, \text{post} \rangle$ consists of a set of classes C and mappings pre and post from the methods of classes from C to pre- and postcondition specifications. Each *class* $\gamma \in C$ is given by its (unique) name; a set of *attributes* of the form $a : \tau$ with a a (unique) name and τ the name of a class in C; and a set of *methods* of the form $m : (x_i : \tau_i)_{1 \leq i \leq k} \to \tau_0$ ($k \in \mathbb{N}$) with m a (unique) name, $x_1, \ldots, x_n$ (unique) parameter names, and $\tau_0, \tau_1, \ldots, \tau_k$ names of classes in C. A *pre-condition* of a method m of class $\gamma \in C$ is given by a boolean term involving the names of attributes of γ and the parameter names of m. A *postcondition* of a method m of class $\gamma \in C$ is given by boolean term involving the names of attributes of γ, references to the pre-state of attributes using the special notation `@pre`, and the special constant `result`.

A *behaviour specification morphism* $\beta : \langle C, \text{pre}, \text{post} \langle$ is given by a map β from the names of $\langle C, \text{pre}, \text{post} \rangle$ to the names of $\langle C', \text{pre}', \text{post}' \rangle$ such that a class name is mapped to a class name, an attribute name is mapped to an attribute name, &c; and the following conditions hold: (1) Given a class $\gamma \in C$ with name n, attributes $a_1 : \tau_1, \ldots, a_k : \tau_k$, and methods $m_1 : (x_{1i} : \tau_{1i})_{1 \leq i \leq k_1} \to \tau_{10}, \ldots, m_l : (x_{li} : \tau_{li})_{1 \leq i \leq k_l} \to \tau_{l0}$ the class $\gamma' \in C'$ denoted by $\sigma(n)$ has at least the attributes $\sigma(a_1) : \sigma(\tau_1), \ldots, \sigma(a_k) : \sigma(\tau_k)$ and at least the methods $\sigma(m_1) : (\sigma(x_{1i}) : \sigma(\tau_{1i}))_{1 \leq i \leq k_1} \to \sigma(\tau_{10}), \ldots, \sigma(m_l) : (\sigma(x_{li} : \sigma(\tau_{li}))_{1 \leq i \leq k_l} \to \sigma(\tau_{l0})$. (2) The homomorphic images of the pre- and postconditions of a method m of class γ w. r. t. β is a pre- and postcondition of $\beta(m)$, resp.

The category *BehSpec* has as objects: behaviour specifications, and as morphisms: behaviour specification morphisms. The composition of behaviour specification morphisms is defined as function composition; the identity morphism on a behaviour specification is given by the identity on the behaviour specification's names.

Signature. The *signature* of a behaviour specification $\text{Sig}(\langle C, \text{pre}, \text{post} \rangle) = \langle S, F \rangle$ is defined as follows: (1) S contains a distinguished sort State. (2) Every class name in C gives rise to a sort symbol in S. (3) An attribute $a : \tau$ of a class with name γ gives rise to an operation $a : \text{State} \times \gamma \to \tau$ in F. (4) A method $m : (\tau_i)_{1 \leq i \leq k} \to \tau_0$ of a class with name γ gives rise to operations $m : \text{State} \times \gamma \times (\tau_i)_{1 \leq i \leq k} \to \tau_0$ and

$m\text{state} : \text{State} \times \gamma \times (\tau_i)_{1 \leq i \leq k} \to \text{State}$ in F. (5) S and F are the least such sets under inclusion.

As each behaviour specification morphism naturally induces a signature morphism, the signature mapping on objects of *BehSpec* can be extended to a functor $\text{Sig} : \mathit{BehSpec} \to \mathit{Sig}$.

Semantics. The *semantics* $\text{Mod}(\langle C, \text{pre}, \text{post}\rangle)$ of a structure diagram $\langle C, \text{pre}, \text{post}\rangle$ is given by the full sub-category of $\text{Sig}(\langle C, \text{pre}, \text{post}\rangle)$-*Alg*-algebras A satisfying the following condition: The interpretation of a pre-condition of a method $m : (x_i : \tau_i)_{1 \leq i \leq k} \to \tau_0$ over an $s \in \text{State}^A$ and parameters $x_1 = v_1, \ldots, x_k = v_k$ implies the interpretation of the postcondition of m over s and $m\text{state}^A(s, v_1, \ldots, v_n)$ with $x_1 = v_1, \ldots, x_k = v_k$ and $\text{result}^A = m(s, v_1, \ldots, v_n)$. The full sub-category of Alg induced by the image of Mod is called BehAlg. This semantics on objects of *BehSpec* is lifted to a functor $\text{Mod} : \mathit{BehSpec}^{\text{op}} \to \mathit{BehAlg}$ by setting $\text{Mod}(\sigma^{\text{op}} : \langle C', \text{pre}', \text{post}'\rangle \to \langle C, \text{pre}, \text{post}\rangle) = \restriction_{\text{Sig}(\sigma)} : \text{Mod}(\langle C', \text{pre}', \text{post}'\rangle) \to \text{Mod}(\langle C, \text{pre}, \text{post}\rangle)$.

B.3 Instance Viewpoint

Syntax. An *object diagram* $\langle O, L\rangle$ is given by a set of objects O and a set of links L. An *object* $o \in O$ is given by a (unique) name; a type (name); and a set of *slots* of the form $a = v$ with a a (unique) name and v an object in O. Each *link* $\lambda \in L$ is given by a pair of *link ends* of the form $\langle a_1 = v_1, a_2 = v_2\rangle$ with a_1, a_2 (unique) names and v_1, v_2 objects in O. An object diagram has to be *well-typed*: If o and o' both show type τ then: (1) if $a = v$ is a slot of o with v an object with type τ there is a slot $a = v'$ of o' with v' an object with type τ; (2) if there is a link $\langle a_1 = o, a_2 = v_2\rangle$ then there is a link $\langle a_1 = o', a_2 = v_2'\rangle$; (3) if there is a link $\langle a_1 = v_1, a_2 = o\rangle$ then there is a link $\langle a_1 = v_1', a_2 = o'\rangle$.

An *object diagram morphism* $\iota : \langle O, L\rangle \to \langle O', L'\rangle$ is given by a map ι from O to O' such that the following conditions hold: (1) Given an object $o \in O$ with name n, type τ, and slots $a_1 = v_1, \ldots, a_k = v_k$, the object $\iota(o) \in O'$ has type τ and at least the slots $a_1 = \iota(v_1), \ldots, \iota(a_k) = \sigma(v_k)$; (2) given a link $\langle a_1 = v_1, a_2 = v_2\rangle \in L$ there is a link $\langle a_1 = \iota(v_1), a_2 : \iota(v_2)\rangle \in L'$.

The category *ObjDiag* has as objects: object diagrams, and as morphisms: object diagram morphisms. The composition of object diagram morphisms is defined as function composition; the identity morphism on an object diagram is given by the identity on the object diagram's objects.

Signature. The *signature* $\text{Sig}(\langle O, L\rangle) = \langle S, F\rangle$ of an object diagram $\langle O, L\rangle$ is defined as follows: (1) Every type name of an object in O gives rise to a sort symbol in S. (2) Every object o in O with type τ gives rise to a constant operation $o : \to \tau$ in F. (3) A slot $a = v$ of an object with type τ and v an object with type τ' gives rise to an operation $a : \tau \to \tau'$ in F. (4) A link $\langle a_1 = v_1, a_2 = v_2\rangle$ in L with v_1, v_2 objects with types τ_1, τ_2, resp., gives rise to the operations $a_1 : \tau_2 \to \tau_1$, $a_2 : \tau_1 \to \tau_2$ in F. (5) S and F are the least such sets under inclusion.

As each object diagram morphism naturally induces a signature morphism, the signature mapping on objects of *ObjDiag* can be extended to a functor $\text{Sig} : \mathit{ObjDiag} \to \mathit{Sig}$.

Semantics. The *semantics* $\mathrm{Mod}(\langle O, L\rangle)$ of an object diagram $\langle O, L\rangle$ is given by the full sub-category of $\mathrm{Sig}(\langle O, L\rangle)$-algebras A satisfying the following conditions: If $a = v$ is a slot in $\langle O, L\rangle$ then $a^A(o^A) = v^A$; if $\langle a_1 = v_1, a_2 = v_2\rangle$ is a link in $\langle O, L\rangle$ then $a_1^A(v_2^A) = v_1^A$ and $a_2^A(v_1^A) = v_2^A$. The full sub-category of Alg induced by the image of Mod is called $InstAlg$. This semantics on objects of $ObjDiag$ is lifted to a functor $\mathrm{Mod} : ObjDiag^{\mathrm{op}} \to InstAlg$ by setting $\mathrm{Mod}(\sigma^{\mathrm{op}} : \langle O', L'\rangle \to \langle O, L\rangle) = \restriction_{\mathrm{Sig}(\sigma)} : \mathrm{Mod}(\langle O', L'\rangle) \to \mathrm{Mod}(\langle O, L\rangle)$.

B.4 Interaction Viewpoint

Syntax. An *interaction diagram* $\langle O, M, \leq\rangle$ is given by a set of objects O, a set of messages M, and a partial order $\leq \subseteq M \times M$. Each object $o \in O$ is given by a (unique) name and a type (name). Each *message* $m \in M$ is given by a (unique) name; a *sender* object in O, a *receiver* object in O, and a message *label* name.

An *interaction diagram morphism* $\mu : \langle O, M, \leq\rangle \to \langle O', M', \leq'\rangle$ is given by a map μ from the names of $\langle O, M, \leq\rangle$ to the names of $\langle O', M', \leq'\rangle$ such that object names are mapped to object names, type names are mapped to type names, &c., and the following conditions hold: (1) A message $m \in M$ with sender $s \in O$, receiver $r \in O$, and label l is mapped to message $\mu(m) \in M'$ with sender $\mu(s) \in O'$, receiver $\mu(r) \in O'$, and label $\mu(l)$. (2) If $m \leq n$ for messages $m, n \in M$ then $\mu(m) \leq' \mu(n)$.

The category $InterDiag$ has as objects: interaction diagrams, and as morphisms: interaction diagram morphisms. The composition of interaction diagram morphisms is defined as function composition; the identity morphism on an interaction diagram is given by the identity on the interaction diagram's messages.

Signature. The *signature* $\mathrm{Sig}(\langle O, M, \leq\rangle) = \langle S, F\rangle$ of an interaction diagram $\langle O, M, \leq\rangle$ is defined as follows: (1) S contains distinguished sorts Obj, Label, Msg, and Boolean. (2) Every type name τ of an object in O gives rise to a sort symbol τ in S. (3) F contains distinguished operation symbols $\mathrm{sender} : \mathrm{Msg} \to \mathrm{Obj}$, $\mathrm{receiver} : \mathrm{Msg} \to \mathrm{Obj}$, $\mathrm{label} : \mathrm{Msg} \to \mathrm{Label}$, and $\leq : \mathrm{Msg} \times \mathrm{Msg} \to \mathrm{Boolean}$. (4) Every object $o \in O$ with type name τ gives rise to a constant operation symbol $o : \to \tau$. (5) Every message label l in M gives rise to a constant operation $l : \to \mathrm{Label}$. (6) Every message with identifier m gives rise to a constant operation symbol $m : \to \mathrm{Msg}$. (7) S and F are the least such sets under inclusion.

As each interaction diagram morphism naturally induces a signature morphism, the signature mapping on objects of $InterDiag$ can be extended to a functor $\mathrm{Sig} : InterDiag \to Sig$.

Semantics. The *semantics* $\mathrm{Mod}(\langle O, M, \leq\rangle)$ of an interaction diagram $\langle O, M, \leq\rangle$ is given by the full sub-category of $\mathrm{Sig}(\langle O, M, \leq\rangle)$-algebras A satisfying the following conditions: (1) The sort symbol Boolean is interpreted as the standard booleans, i.e. $\mathrm{Boolean}^A = \mathbb{B}$. (2) Obj is interpreted as the union of the interpretion of all sort symbols τ for object type names. (3) If m is a message in M with sender s, receiver r, and message label l, then $\mathrm{sender}^A(m^A) = s^A$ and $\mathrm{receiver}^A(m^A) = r^A$ and $\mathrm{label}^A(m^A) = l^A$. (4) If m and n are messages in M with $m \leq n$ then $\leq^A(m^A, n^A) = true$. The

full sub-category of Alg induced by the image of Mod is called $InterAlg$. This semantics on objects of $InterDiag$ is lifted to a functor $\mathrm{Mod} : InterDiag^{\mathrm{op}} \to InterAlg$ by setting $\mathrm{Mod}(\sigma^{\mathrm{op}} : \langle O', M', \leq' \rangle \to \langle O, M, \leq \rangle) = \restriction_{\mathrm{Sig}(\sigma)} : \mathrm{Mod}(\langle O', M', \leq' \rangle) \to \mathrm{Mod}(\langle O, M, \leq \rangle)$.

B.5 State Machine Viewpoint

Syntax. A *state machine diagram* $\langle C, \mu \rangle$ is given by a set of classes C and a mapping μ of classes to machines. Each *class* $\gamma \in C$ is given by its (unique) name; a set of *attributes* of the form $a : \tau$ with a a (unique) name and τ the name of a class in C; and a set of *methods* of the form $m : (\tau_i)_{1 \leq i \leq k} \to \tau_0$ ($k \in \mathbb{N}$) with m a (unique) name and $\tau_0, \tau_1, \ldots, \tau_k$ names of classes in C. A *machine* of class $\gamma \in C$ is given by a set S of (unique) *state* names, a set T of transitions, and an *initial state* $i \in S$. A *transition* of a machine of class γ is given by a source state $s_1 \in S$, an incoming event e from the method names of C, a guard as a boolean expression of the names of attributes of γ and the names of parameters of the event e, an effect consisting of a statement over the names of attributes of γ and the names of parameters of the event e and a subset of outgoing messages, and a target state $s_2 \in S$. An outgoing message of a machine of class γ is given by the name of an attribute of γ and the name of a method name of the class of this attribute and list of expressions over the names of attributes of γ and the parameters of e.

A *state machine diagram morphism* $\beta : \langle C, \mu \rangle \to \langle C', \mu' \rangle$ is given by a map β from the names in $\langle C, \mu \rangle$ to the names in $\langle C', \mu' \rangle$ such that class names are mapped to class names, attribute names are mapped to attribute names, &c., and the following condition holds: $\mu'(\beta(C)) = \beta(\mu(C))$ where $\beta(\mu(C))$ is the homomorphic extension of β to guards, statements, and expressions.

The category $MachDiag$ has as objects: state machine diagrams, and as morphisms: state machine diagram morphisms. The composition of state machine diagram morphisms is defined as function composition; the identity morphism on a state machine diagram is given by the identity on the state machine diagram's classes, attributes, and method names.

Signature. The *signature* $\mathrm{Sig}(\langle C, \mu \rangle) = \langle S, F \rangle$ of a state machine diagram $\langle C, \mu \rangle$ is defined as follows: (1) S contains distinguished sorts Obj, $\mathrm{Set}(\mathrm{Obj})$, Env, State, Msg, and $\mathrm{Set}(\mathrm{Msg})$. (2) Every class name $\gamma \in C$ gives rise to a sort symbol γ in S. (3) F contains a distinguished operation symbol $\mathrm{obj} : \mathrm{Env} \to \mathrm{Set}(\mathrm{Obj})$. (4) F contains a distinguished operation symbol $\mathrm{initial} : \to \mathrm{State}$. (5) F contains distinguished operation symbols $\mathrm{state} : \mathrm{Obj} \times \mathrm{State} \times \mathrm{Env} \to \mathrm{State}$, $\mathrm{env} : \mathrm{Obj} \times \mathrm{State} \times \mathrm{Env} \to \mathrm{Env}$, $\mathrm{msg} : \mathrm{Obj} \times \mathrm{State} \times \mathrm{Env} \to \mathrm{Msg}$. (6) An attribute $a : \tau$ of a class with name γ gives rise to an operation $a : \mathrm{Env} \times \gamma \to \tau$. (7) A method $(\tau_i)_{1 \leq i \leq k} \to \tau_0$ of a class with name γ gives rise to an operation $m : \gamma \times (\tau_i)_{1 \leq i \leq k} \to \mathrm{Msg}$. (7) S and F are the least such sets under inclusion.

As each state machine diagram morphism naturally induces a signature morphism, the signature mapping on objects of $MachDiag$ can be extended to a functor $\mathrm{Sig} : MachDiag \to Sig$.

Semantics. The *semantics* $\mathrm{Mod}(\langle C, \mu\rangle)$ of a state machine diagram $\langle C, \mu\rangle$ is given by the full sub-category of $\mathrm{Sig}(\langle C, \mu\rangle)$-algebras A satisfying the following conditions: (1) The sort symbol Obj is interpreted as the union of the interpretions of the sort symbols τ for classes. (2) The sort symbol $\mathrm{Set}(\mathrm{Obj})$ is interpreted as the standard subsets of the interpretion of Obj; the sort symbol $\mathrm{Set}(\mathrm{Msg})$ is interpreted as the standard subsets of the interpretion of Msg. (3) The operaton symbol env is interpreted as the function that given an object in the interpretion of sort τ, a state s_1 of the machine of class τ, and an environment e yields the environment resulting from executing the machine for class τ in one step. The operation symbols state and msg are to be interpreted similarly, but resulting in the state and the outgoing messages, respectively. The full sub-category of Alg induced by the image of Mod is called $MachAlg$. This semantics on objects of $MachDiag$ is lifted to a functor $\mathrm{Mod} : MachDiag^{\mathrm{op}} \to MachAlg$ by setting $\mathrm{Mod}(\sigma^{\mathrm{op}} : \langle C', \mu'\rangle \to \langle C, \mu\rangle) = \restriction_{\mathrm{Sig}(\sigma)} : \mathrm{Mod}(\langle C', \mu'\rangle) \to \mathrm{Mod}(\langle C, \mu\rangle)$.

Author Index