

Radu Calinescu
Ethan Jackson (Eds.)

LNCS 6662

Foundations of Computer Software

Modeling, Development,
and Verification of Adaptive Systems

16th Monterey Workshop 2010
Redmond, WA, USA, March/April 2010
Revised Selected Papers

 Springer

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Alfred Kobsa

University of California, Irvine, CA, USA

Friedemann Mattern

ETH Zurich, Switzerland

John C. Mitchell

Stanford University, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz

University of Bern, Switzerland

C. Pandu Rangan

Indian Institute of Technology, Madras, India

Bernhard Steffen

TU Dortmund University, Germany

Madhu Sudan

Microsoft Research, Cambridge, MA, USA

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max Planck Institute for Informatics, Saarbruecken, Germany

Radu Calinescu Ethan Jackson (Eds.)

Foundations of Computer Software

Modeling, Development,
and Verification of Adaptive Systems

16th Monterey Workshop 2010
Redmond, WA, USA, March 31 – April 2, 2010
Revised Selected Papers

Volume Editors

Radu Calinescu

Oxford University, Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
E-mail: Radu.Calinescu@comlab.ox.ac.uk

Ethan Jackson

Microsoft Research
One Microsoft Way, Redmond, WA 98052-6399, USA
E-mail: ejackson@microsoft.com

ISSN 0302-9743

e-ISSN 1611-3349

ISBN 978-3-642-21291-8

e-ISBN 978-3-642-21292-5

DOI 10.1007/978-3-642-21292-5

Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: Applied for

CR Subject Classification (1998): D.2, H.4, H.3, C.2, H.5, D.2.2, C.2.4

LNCS Sublibrary: SL 2 – Programming and Software Engineering

© Springer-Verlag Berlin Heidelberg 2011

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Message from the General Chairs

The 16th Monterey Workshop was held at the Microsoft Headquarters during March 31 – April 2, 2010 in Redmond, Washington.

A decade into the new millennium, the field of software engineering faces more challenges than ever to evolve with the increasing demands placed on software, as technology expands and becomes even more integrated into every facet of our lives. This pervasiveness increases not only the range of challenges that software applications must face but also the adaptability, flexibility, and robustness they need in order to reliably function in the chaotic jungle of the real world.

Accordingly, the 16th Monterey Workshop investigated an intriguing direction for potential innovation: mechanisms by which organisms cope with harsh, unfavorable, and variable conditions in the natural world. Distillation and formalization of common strategies of complex systems that persevere and function in severely stressful or unexpected situations can aid in the design of systems that must be able to weather similarly chaotic environments.

While these observed strategies provide a potential source of inspiration, unlike nature, software engineers are not limited to trial and error. We can formulate mathematical models and design principles to provide systematic avenues for realizing, analyzing, and improving software reliability. Specialization to meet the particular needs of software development and clever design based on insights gained from natural strategies may eventually surpass the robustness of biological systems. Multiple approaches are explored here, ranging from decentralization-based redundancy and improved verification of flexible systems with many configurations to self-adaptive, self-management, and self-correction capabilities.

You will find interest and inspiration in the work of this gathering of brilliant minds at Microsoft Headquarters. Touring Microsoft Research was a fantastic opportunity. The presentations on various frontier topics are extremely interesting. The papers in this volume represent the most important directions at the workshop, refined in response to workshop discussions and referee comments.

In particular, we note several presentations dealing with the notion of adaptation in software systems. It is amazing how this notion could be specialized and refined in various application domains such as autonomous space systems, re-configuration of modular robots, adaptation to application design, managements of unpredictable changes in specifications, and certification of reconfiguration. This subtopic was covered in many different ways during the workshop, without prior coordination of the invitees. This outcome is not surprising if we consider that ubiquitous systems are rising, with increasing rates of new requirements, new operating environments, subsystem failures, and hostile activity.

In some contexts adaptation is a necessity, for a variety of reasons. For example, autonomic robots in some space missions have to be self-adaptive because

there is no way to get an answer from earth, due to a 40-minute delay. In the context of reconfigurable modular robots, runtime adaptation is needed because there is no way to pre-compute the potential moves, due to a combinatorial explosion of possible starting states. Variability in modeling languages requires adaptability because the application-specific extensions needed are discovered only when we know the application. Lightweight formal methods supporting re-configuration in response to varying system loads are needed to derive a system change that will not violate any system constraints, to ensure success of the proposed adaptation before attempting any system modifications. In these and many other cases, details of the required adaptation depend on information that is not available at the time the system is designed.

We very much enjoyed sharing of the advancement of science and technology in the field of software engineering with the research community, following the culture and tradition of the Monterey Workshop, and would like to thank our fantastic Program Committee Chairs Radu Calinescu from Oxford and Ethan Jackson from Microsoft for assembling a fascinating workshop program.

On behalf of the Monterey Workshop Steering Committee, we would like to thank NSF, ONR, AFOSR, ARO, DARPA, and all of our European research sponsors for their support for the Monterey Workshops over the years, and ARO and Microsoft in particular for making this 16th Monterey Workshop possible. Many of the Monterey Workshop topics have subsequently blossomed into major research initiatives and widespread applications with great benefit to all. Here are the topics of Monterey Workshops from the past two decades:

- 0th: Research Review on Formal Methods in Software Engineering: Concurrent and Real-time Systems, Monterey, California, 1991
- 1st: Computer-Aided Prototyping: CAPSTAG, Monterey, California, 1992
- 2nd: Software Slicing, Merging and Integration, Monterey, California, 1993
- 3rd: Software Evolution, Monterey, California, 1994
- 4th: Specification-Based Software Architectures, Monterey, California, 1995
- 5th: Requirements Targeting Software and Systems Engineering, Bernried, Germany, 1997
- 6th: Engineering Automation for Computer-Based Systems, Monterey, California, 1998
- 7th: Modeling Software and System Structure in a Fast-Moving, Scenario, Santa Margherita Ligure, Italy, 2000
- 8th: Engineering Automation for Software-Intensive System Integration, Monterey, California, 2001
- 9th: Radical Innovations of Software and Systems Engineering in the Future, Venice, Italy, 2002
- 10th: Software Engineering for Embedded Systems: From Requirements to Implementation, Chicago, Illinois, 2003
- 11th: Software Engineering Tools: Compatibility and Integration, Vienna, Austria, 2004
- 12th: Realization of Reliable Systems on Top of Unreliable Networked Platforms, Irvine, California, 2005

- 13th: Composition of Embedded Systems: Scientific and Industrial Issues, Paris, France, 2006
- 14th: Innovations for Requirement Analysis: From Stakeholders' Needs to Formal Designs, Monterey, California, 2007
- 15th: Foundations of Computer Software, Future Trends and Techniques for Development, Budapest, Hungary, 2008
- 16th: Modeling, Development and Verification of Adaptive Systems, Redmond, Washington, 2010

March 2011

Luqi
Fabrice Kordon

Message from the Program Chairs

The 16th Monterey Workshop was held at Microsoft Research in Redmond, WA, at an exciting turning point in consumer technology. Cloud computing was becoming mainstream facilitated by a combination of advances in virtualization technology and concerns about the costs and environmental impact of maintaining an in-house IT infrastructure. For the first time, sales of smart phones were predicted to overtake sales of laptops. These two trends have since synergized to provide powerful mobile computing on an unprecedented scale.

Similar technological advances have led to a continual increase in the adoption of IT-based solutions in industrial safety-critical and business-critical applications in recent years. We anticipate that cyber-physical systems — systems that integrate computing and physical processes — will become increasingly common over the next decade.

This evolution focuses our attention on a number of key research challenges. How can we ensure information privacy and security? Can data-centers, clouds, and other large-scale distributed systems be made reliable enough to truly depend on them? Can we certify that software performs its intended functions, and can it adapt to withstand unanticipated component failures? During the workshop we listened to presentations by experienced researchers in the modeling, development and verification of adaptive computer systems, and we discussed these challenges from many angles. This proceedings volume gives both an outline of these discussions and an extension of the works presented at the workshop.

We would like to thank the participants for their insightful perspectives and lively discussion, which made this volume possible.

We would also like to thank Jim Larus of the Extreme Computing Group (XCG) for his overview of Microsoft's research in cloud computing. Similarly, we thank Desney Tan for presenting his group's research on next-generation user input devices, giving us a glimpse of what might come after the touch screen. Finally, we are grateful to Fabrice and Luqi for organizing the Monterey Workshop series.

March 2011

Radu Calinescu
Ethan Jackson

Table of Contents

Software Verification of Autonomic Systems Developed with ASSL	1
<i>Emil Vassev and Mike Hinchey</i>	
Modeling Language Variability	17
<i>Hans Grönniger and Bernhard Rumpe</i>	
An Approach for Effective Design Space Exploration	33
<i>Eunsuk Kang, Ethan Jackson, and Wolfram Schulte</i>	
Migration of Legacy Software towards Correct-by-Construction Timing Behavior	55
<i>Stefan Resmerita, Kenneth Butts, Patricia Derler, Andreas Naderlinger, and Wolfgang Pree</i>	
Towards IT Systems Capable of Managing Their Health	77
<i>Selvi Kadirvel and José A.B. Fortes</i>	
Self-reconfigurable Modular Robots and Their Symbolic Configuration Space	103
<i>Souheib Baarir, Lom-Messan Hillah, Fabrice Kordon, and Etienne Renault</i>	
Formal Methods @ Runtime	122
<i>Radu Calinescu and Shinji Kikuchi</i>	
Modular State Spaces for Prioritised Petri Nets	136
<i>Charles Lakos and Laure Petrucci</i>	
A Problem Frame-Based Approach to Evolvability: The Case of the Multi-translation	157
<i>Gianna Reggio, Egidio Astesiano, Filippo Ricca, and Maurizio Leotta</i>	
Towards a Framework for Modelling and Verification of Relay Interlocking Systems	176
<i>Anne E. Haxthausen</i>	
Trust Of, In, and among Adaptive Systems	193
<i>Douglas S. Lange</i>	

Software Certification: Is There a Case against Safety Cases?	206
<i>Alan Wassing, Tom Maibaum, Mark Lawford, and Hans Bherer</i>	
Testing Adaptive Probabilistic Software Components in Cyber Systems	228
<i>Luqi and Grant Jacoby</i>	
Author Index	239

Software Verification of Autonomic Systems Developed with ASSL

Emil Vassev¹ and Mike Hinchey²

Lero—the Irish Software Engineering Research Centre
^{1,2} University of Limerick, Limerick, Ireland
{Emil.Vassev, Mike.Hinchey}@lero.ie

Abstract. We discuss our experiences in building tools for software verification of autonomic systems developed with the Autonomic System Specification Language (ASSL). ASSL is a software framework that aims to assist developers of autonomic systems by providing a powerful combination of both notation and tools. One of the major objectives of the framework is to assure the correctness of the autonomic systems via inclusion of tools targeting consistency checking, model checking, and automatic test case generation. In this paper, we review our recent work on these tools.

Keywords: software verification, formal methods, ASSL, autonomic computing.

1 Introduction

The Autonomic System Specification Language (ASSL) [1, 2] is a formal method dedicated to the development of autonomic systems (ASs) [3]. Conceptually, ASSL assists developers with *formal specification*, *validation*, and *code generation* of such systems. Due to the synthesis approach of automatic code generation, ASSL guarantees consistency between a specification and the corresponding implementation. As part of the framework validation, ASSL has been successfully used to specify autonomic features and generate AS models for a variety of computer systems including prototypes of two NASA projects—the Autonomous Nano-Technology Swarm (ANTS) concept mission [4] and the Voyager mission [5]. Our experience with ASSL has demonstrated that although the framework is very efficient, errors can be easily introduced while specifying large systems. The first release of ASSL provides built-in consistency checking and functional testing as the only means of software verification. This helps developers easily discover syntax and consistency errors, but barely handles logical errors. To increase the framework’s software-verification capabilities, we have been investigating model checking [6] as the most effective approach to software verification for our purposes. In addition, in order to detect errors introduced not only in ASSL specifications, but also with supplementary coding, the automatic verification support provided by the ASSL tools is to be augmented by appropriate automatic generation of test cases. Both model checking and automatic test case generation are subjects of new research projects at Lero—the Irish Software Engineering Research Center. In this paper, we briefly present existing and new software-verification approaches for ASSL.

The rest of this paper is organized as follows: In Section 2, we briefly present the ASSL formal specification model. In Section 3, we present the basic consistency checking mechanism. Sections 4 and 5 present our approach to model checking and automatic test case generation with ASSL. Finally, Section 6 provides brief concluding remarks and a summary of future research goals.

2 ASSL

ASSL [1, 2] is based on a specification model exposed over hierarchically organized formalization tiers (see Table 1). This specification model provides both infrastructure elements and mechanisms needed by an AS (autonomic system). Each tier of the ASSL specification model is intended to describe different aspects of the AS in question, such as *service-level objectives, policies, interaction protocols, events, actions, autonomic elements*, etc. This helps to specify an AS at different levels of abstraction (imposed by the ASSL tiers) where the AS in question is composed of special autonomic elements (AEs) interacting over interaction protocols (IPs).

Table 1. ASSL multi-tier specification model

AS	AS Service-level Objectives	
	AS Self-management Policies	
	AS Architecture	
	AS Actions	
	AS Events	
	AS Metrics	
ASIP	AS Messages	
	AS Channels	
	AS Functions	
AE	AE Service-level Objectives	
	AE Self-management Policies	
	AE Friends	
	AEIP	AE Messages
		AE Channels
		AE Functions
		AE Managed Elements
	AE Recovery Protocols	
	AE Behavior Models	
	AE Outcomes	
	AE Actions	
AE Events		
AE Metrics		

As shown in Table 1, the ASSL specification model decomposes an AS in two directions: 1) into levels of functional abstraction; and 2) into functionally related *sub-tiers*. The first decomposition presents the system at three different tiers [1, 2]:

- 1) *a general and global AS perspective* – we define the general system rules (providing autonomic behavior), architecture, and global actions, events, and metrics applied in these rules;
- 2) *an interaction protocol (IP) perspective* – we define the means of communication between AEs within an AS;
- 3) *a unit-level perspective* – we define interacting sets of individual computing elements (AEs) with their own autonomic behavior rules, actions, events, metrics, etc.

The second decomposition presents the major tiers AS, ASIP and as composed of functionally related sub-tiers, where new AS properties emerge at each sub-tier. The AS Tier specifies an AS in terms of *service-level objectives* (AS SLOs), *self-management policies*, *architecture topology*, *actions*, *events*, and *metrics* (see Table 1). The AS SLOs are a high-level form of behavioral specification that helps developers establish system objectives such as performance. The *self-management policies* are driven by *events* and trigger the execution of *actions* driving an AS in critical situations. The metrics constitute a set of parameters and observables controllable by an AS. With the ASIP Tier, the ASSL framework helps developers specify an AS-level interaction protocol as a public communication interface expressed with special *communication channels*, *communication functions*, and *communication messages*. At the AE Tier, the ASSL formal model exposes specification constructs for the specification of the system’s AEs. Note that AEs are considered to be analogous to software agents able to manage their own behavior and their relationships with other AEs. An AE may also specify a private *AE interaction protocol* (AEIP) shared with special AE considered as “friends” (AE Friends tier).

It is important to mention that the ASSL tiers are intended to specify different aspects of the AS in question, but it is not necessary to employ all of them in order to develop an AS. Conceptually, it is sufficient to specify self-management policies only, because those provide self-management behavior at the level of AS (the AS tier) and at the level of AE (AE tier). These policies are specified within the AS/AE Self-management Policies sub-tier (the ASSL construct is `AS[AE]SELF_MANAGEMENT`) with special ASSL constructs termed *fluents* and *mappings* [1, 2]. A fluent is a state where an AS enters with fluent-activating events and exits with fluent-terminating events. A mapping connects fluents with particular actions to be undertaken. Usually, an ASSL specification is built around one or more self-management policies, which make that specification AS-driven. Self-management policies are driven by events and actions determined deterministically. The following ASSL code presents a sample specification of a self-healing policy.

```

ASSELF_MANAGEMENT {
  SELF_HEALING {
    FLUENT inLosingSpacecraft {
      INITIATED_BY { EVENTS.spaceCraftLost }
      TERMINATED_BY { EVENTS.earthNotified } }
    MAPPING {
      CONDITIONS { inLosingSpacecraft }
      DO_ACTIONS { ACTIONS.notifyEarth } }
  }
} // ASSELF_MANAGEMENT

```

As shown, fluents are expressed with *fluent-activating* and *fluent-terminating* events. In order to express mappings, conditions and actions are considered, where the former determine the latter in a deterministic manner.

Once a specification is complete, it can be validated with the ASSL built-in verification mechanisms (e.g., consistency checking) and a functional application skeleton can be generated automatically. The application skeletons generated with the ASSL framework are fully-operational multithreaded event-driven applications with embedded messaging.

3 Consistency Checking with ASSL

In general, we can group the ASSL tiers into groups of *declarative* (or *imperative*) and *operational* tiers [1, 2]. Whereas the former simply describe definitions in the AS under consideration, the latter not only describe definitions but also focus on the operational behavior of that AS. The ASSL framework evaluates an AS specification formally to construct a special *declarative specification tree* needed to perform both consistency checking and code generation. The declarative specification tree is created by the framework when parsing an AS specification and contains the hierarchical tier structure of that specification. Each specified tier/sub-tier is presented as a *tier instance*. Consistency checking (see Fig. 1) is a framework mechanism for verifying specifications by performing exhaustive traversing of the declarative specification tree. In general, the framework performs two kinds of consistency-checking: 1) *light* – checks for type consistency, ambiguous definitions, etc.; and 2) *heavy* – checks whether the specification model conforms to special *correctness properties*.

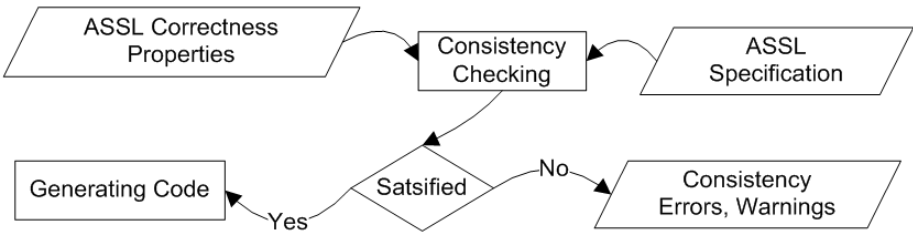


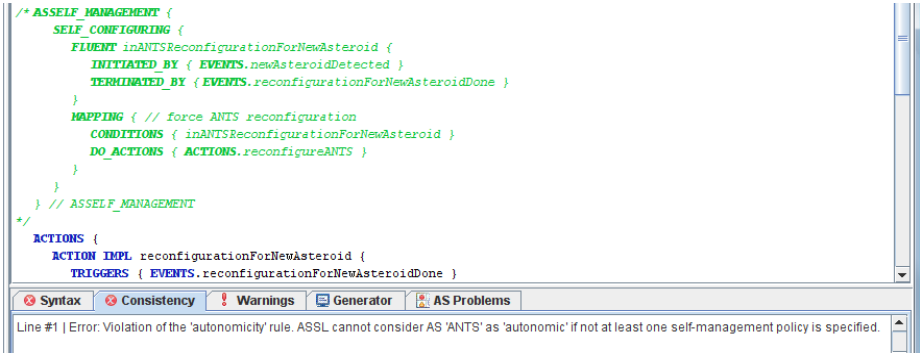
Fig. 1. Consistency Checking with ASSL

The correctness properties are *ASSL semantic definitions* [1, 2] defined per tier. Although, they are expressed in First-Order Linear Temporal Logic (FOLTL)¹ [6], currently ASSL does not incorporate a FOLTL engine, and thus, the consistency checking mechanism implements the correctness properties as Java statements. Here, the FOLTL operators \forall (forall) and \exists (exists) work over sets of ASSL tier instances. It is important to mention that the consistency checking mechanism generates *consistency errors* and *consistency warnings*. Warnings are specific situations where the

¹ In general, FOLTL can be seen as a quantified version of linear temporal logic. FOLTL is obtained by taking propositional linear temporal logic and adding a first order language to it.

specification does not contradict the correctness properties, but rather introduces uncertainty as to how the code generator will handle it.

As mentioned above, a variety of predefined correctness properties are subject of consistency checking. One of those correctness properties is the so-called autonomicity rule [1, 2]. According to that rule, every autonomic system specified with ASSL must have specified at least one self-management policy. Fig. 2 shows an error reported by the ASSL’s consistency checker, because the processed ASSL specification violates the autonomicity rule (the entire `ASSELF_MANAGEMENT` sub-tier comprising the self-management policies is commented).



```

/* ASSELF_MANAGEMENT {
  SELF_CONFIGURING {
    FLUENT inANTSReconfigurationForNewAsteroid {
      INITIATED_BY { EVENTS.newAsteroidDetected }
      TERMINATED_BY { EVENTS.reconfigurationForNewAsteroidDone }
    }
    MAPPING { // force ANTS reconfiguration
      CONDITIONS { inANTSReconfigurationForNewAsteroid }
      DO_ACTIONS { ACTIONS.reconfigureANTS }
    }
  } // ASSELF_MANAGEMENT
}
*/

ACTIONS {
  ACTION_IMPL reconfigurationForNewAsteroid {
    TRIGGERS { EVENTS.reconfigurationForNewAsteroidDone }
  }
}

```

Line #1 | Error: Violation of the 'autonomicity' rule. ASSL cannot consider AS 'ANTS' as 'autonomic' if not at least one self-management policy is specified.

Fig. 2. Checking for “Autonomicity” with the Consistency Checker

4 Built-in Model-Checking Mechanism for ASSL

In general, model checking advocates formal verification whereby software programs are automatically checked for specific flaws by considering correctness properties expressed in *temporal logic* [6]. In this endeavor, three model-checking mechanisms for ASSL have been considered: 1) a built-in model checker [7]; 2) a mechanism for mapping ASSL specifications to formal notation with provided tool support for model checking [8]; and 3) a post-implementation model checker based on the Java Path-Finder [9] tool developed at NASA Ames. Whereas the first two model-checking methods check ASSL specifications, the third one is to verify the generated Java code. Note that despite careful specification and the existence of ASSL-level model checking, it is theoretically possible to generate ASs that contain fatal errors (e.g., deadlocks). This is mainly due to the state-explosion problem, which we discuss in Section 4.2. Moreover, with the post-implementation model checker we may verify not only the newly-generated code but also all consecutively updated versions of the same. Thus, the ASSL model-checking mechanisms are intended to verify both the ASSL specifications and the corresponding AS implementations.

In this paper, we report our experience in developing the built-in model checker [7]. In this approach, an ASSL specification is translated into a *state-transition graph*, over which model checking is performed to verify whether an ASSL specification satisfies *correctness properties*. Here, the model-checking problem is: given the AS A

and its ASSL specification a , determine in the AS's state graph g (called ASG) whether the behavior of A , expressed with the correctness properties p , meets the specification a . An ASG formally stems from the concept of Kripke Structure [6]. The latter is basically a graph having the reachable states of an ASSL-specified system as nodes and the state transitions of the system as edges. In addition, to allow for formal verification, each system state must be labeled with properties (called atomic propositions AP) that hold in that state and each state transition must be associated with one or more state transition operations Op . The notion of state in ASSL is related to the ASSL specification constructs called *ASSL tier instances* [1, 2] (specified tiers and sub-tiers). The ASSL operational semantics [1, 2] considers a state-transition model where *tier instances* can be in different *tier states*, e.g., instances of the SLO (Service-Level Objectives) tier can be evaluated as *satisfied* or *not satisfied*. Here, an ASSL-developed AS transits from one state to another when a particular tier instance *evolves* from a tier state to another tier state. Here, transition operations Op cause tier instances to evolve.

4.1 Building the Autonomic System Graph

In order to build the ASSL model checker, we had to do some preliminary theoretical work to prepare the program structures holding an ASG. Here, we had to define:

- 1) the reference state model for ASSL-specified ASs, which appeared to be a product machine that consists of *high-level tier states* composed of multilevel *nested tier states*, and the global system state is a product of all nested states (we had to identify an initial state and all the possible tier states S);
- 2) a set of all atomic propositions AP , which denote the properties of individual states S , and present the S - AP relationship as tuples of the form (S_n, AP_1, \dots, AP_n) ;
- 3) all possible transition relations R as tuples of the form (S_1, Op, S_2) .

Next, we had to implement structures holding the S - AP and R tuples. Note that those are recorded in two flat files (one per tuple type) and are loaded into the implemented program structures at the time of ASSL loading. This helps the model-checker tool cope with future extensions to ASSL. To implement the tuple structures, we used a distinct token class per tuple type (S - AP and R) and used vectors of tuple tokens. In addition, a generic algorithm is implemented to traverse those vectors and return a sub-vector of tuple tokens refined by *state*, by *operation*, or by *atomic proposition*. Thus, at runtime, the model-checking tool can obtain all the atomic propositions and related transition operations for a particular state. Here,

- tier states S are recorded with tier instance name and state name;
Example: **tier** { *SLO* } **name** { *performance* } **state** { *unsatisfied* }
- transition operations Op are recorded with their ASSL predefined names [1];
- atomic propositions AP are recorded with “if” and “then” sections and optional “temporal” operators (a temporal logic operator).

Example: **if** { *event prompted* } **then** { **tempOperator** { *eventually* } *fluent initiated* }.

In the next step, we had to develop a mechanism constructing the ASG from an ASSL specification. Here, the ASG is constructed by the ASSL framework by using a

special *declarative specification tree* created by the framework when parsing an AS specification [1, 2]. The declarative specification tree contains the hierarchical tier structure of the actual specification. Thus, enriched with the tier states \mathcal{S} , it can be used to derive the composite multilevel structure of the ASG by taking into consideration that all the tier instances run concurrently as state machines. Thus, the tier states \mathcal{S} are derived from the declarative specification tree and enriched with the appropriate atomic propositions AP . The latter are retrieved per state.

In addition, the so-called *operational evaluation* [1, 2] performed on the ASSL specification is used to derive all the transition relations $R(S_1, Op, S_2)$ needed to connect the states \mathcal{S} and thus, to construct the ASG. Here, an ASG is composed of nodes that can be presented formally as a tuple (s, R, AP) where: s is the tier state; R is a set of transition relations connecting the state s to other states via system operations; AP is a set of atomic propositions held in s . Similar to the declarative specification tree, the generated ASG is hierarchical, i.e., composed of multilevel composite tier states. Note that the generated ASG is stored in a flat file, which helps us trace the graph. Fig. 3 depicts the transformation of the declarative specification tree into an ASG, where the latter is presented at the highest possible level of abstraction comprising a single composite state “AS Active”, which is a product machine consisting of product states.

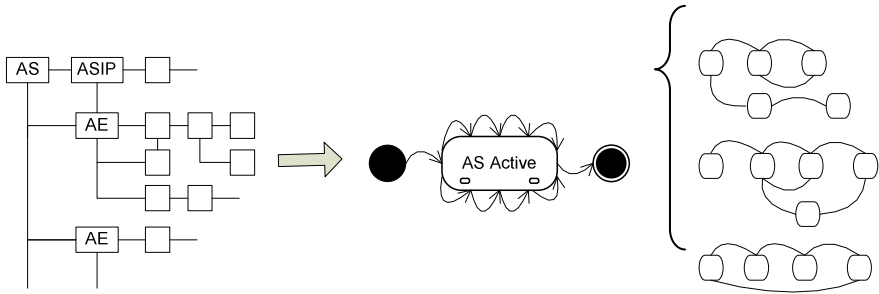


Fig. 3. Transformation of the Declarative Specification Tree into an ASG

4.2 Building the Model-Checking Engine

Next, we had to implement the model checking engine that should work over the following algorithm: *given that Φ is a correctness property expressed in a temporal logic formula, determine whether the “AS Active” tier state (see Fig. 3) satisfies Φ , which implies that all possible compositions of nested tier states satisfy Φ .*

Thus, the model-checking engine traverses all the possible paths in an ASG to check whether special correctness properties Φ (expressed in a temporal logic) are satisfied. In case such a property is not satisfied, the ASSL framework produces a counterexample. The latter is an execution path of the ASG for which the desired correctness property is not true.

At the time of writing, the model-checking engine is still under development. We are currently examining two possible solutions: 1) developing our own engine; or 2) integrating an already existing engine that can process the generated ASG file. Engines of current interest are SPIN [10] and GEAR [11]. In all approaches though, we

need to consider the so-called *state-explosion problem*. In general, the size of an ASG is at least exponential in the number of ASSL tier instances running concurrently in the system (recall that an ASG is a product machine). We are currently working on two possible solutions to that problem—*abstraction* and *prioritized tiers*. The first solution is to use composite tier states to abstract their nested tier states. Thus, given an original state graph G (derived from an ASSL specification) an abstraction is obtained by suppressing low-level tier states yielding a simpler and likely smaller state graph G_a . This reduces the total amount of states to be considered but is likely to introduce a sort of conservative view of the system where the abstraction ensures only that correctness of G_a implies correctness of G . The other possible solution is to prioritize ASSL tiers by giving their tier states a special probability weight ρw . This can be used as a state-reduction factor to derive probability graphs $G_{\rho w}$ with a specific level of probability weight, e.g., $\rho w > 0,5$. However, this approach is likely to introduce probability to the model-checking results, which correlates with the probability level of the graph $G_{\rho w}$.

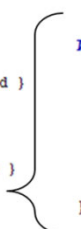
4.3 Checking Liveness Properties

This section demonstrates how the ASSL built-in model-checking mechanism can perform formal verification to check *liveness* properties of an AS specified and generated with ASSL. Our example is the ASSL specification model for the NASA Voyager Mission [5]. In this case study, we specified the Voyager II spacecraft and the antennas on Earth as AEs (autonomic elements) that follow their encoded autonomic behavior to process space pictures, and communicate those via predefined ASSL messages. In this section, we use a sample from this specification to demonstrate how a liveness property such as “*a picture taken by the Voyager spacecraft will eventually result in sending a message to antennas on Earth*” can be checked with the ASSL model-checking mechanism. Note that the ASSL specification model for the NASA Voyager Mission is relatively large (over 1000 lines of specification code). Thus, we do not present the entire specification but a specification sample. For more details on that specification, please refer to [5].

```

POLICY IMAGE_PROCESSING {
  ....
  FLUENT inProcessingPicturePixels {
    INITIATED_BY { EVENTS.pictureTaken }
    TERMINATED_BY { EVENTS.pictureProcessed }
  }
  ....
  MAPPING {
    CONDITIONS { inProcessingPicturePixels }
    DO_ACTIONS { ACTIONS.processPicture }
  }
}

```



```

ACTION processPicture {
  ....
  DOES {
    ....
    call AEIP.FUNCTIONS.sendBeginSessionMsgs
    ....
  }
}

```

Fig. 4. The IMAGE_PROCESSING policy

Fig. 4 presents a partial ASSL specification of the IMAGE_PROCESSING self-management policy of the Voyager AE. Here the `pictureTaken` event will be prompted when a picture has been taken. This event initiates the `inProcessingPicturePixels` fluent. The same fluent is mapped to a `processPicture` action, which will be executed once the fluent

gets initiated. As it is specified, the `processPicture` action prompts the execution of the `sendBeginSessionMsgs` communication function (see Fig. 4), which puts a special message \mathbf{x} on a special communication channel [5] (message \mathbf{x} is sent over that channel). Note that the specification of both the `pictureTaken` event and the `sendBeginSessionMsgs` function is not presented here. As we have already mentioned in Section 4.1, the ASSL model-checking mechanism builds the ASG (autonomic system graph) from the ASSL specification. Here both the *declarative specification tree* and the *ASSL operational semantics* [1, 2] are used to derive tier states \mathcal{S} and transition relations \mathcal{R} , and to associate those tier states via the ASSL transition operations Op . Next the labeling function $L(s)$ (integrated in the model-checking mechanism) labels each tier state s with appropriate atomic propositions AP .

Fig. 5 presents a partial ASSL ASG of the sub-tiers of the Voyager AE. These sub-tiers are derived from the declarative specification tree constructed for the Voyager AE. Note that this ASG is a result of our analytical approach and for reasons of clarity it is simplified, i.e., not all the possible tier states are presented here.

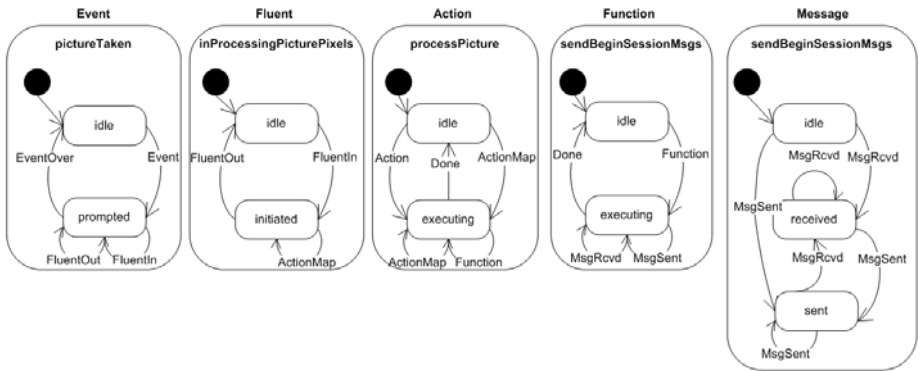


Fig. 5. State machines of the Voyager AE sub-tiers

As shown, each sub-tier instance forms a distinct *state machine* (basic machine) within the AE state machine and the AE state machine is a *Cartesian product* of the state machines of its sub-tiers. It is important to mention that by taking the Cartesian product of a set of basic sub-tier machines, we form a product machine consisting of product states. The latter are tuples of concurrent basic sub-tier states. Moreover, in the AE product machine, the ASSL state-transition operations Op are considered product transitions that move from one product state to another. Note that the states in the state machine of the whole AS product machine can be obtained by the Cartesian product of all the AE product machines. Thus, by considering the sub-tier state machines we construct the Voyager AE product machine (see Fig. 6). Note that this is again a simplified model where not all the possible product states are shown.

Fig. 6 presents the AE product states as large circles embedding the sub-tier states (depicted as smaller circles). Here we use the following aliases: e states for *Event state machine*; f states for *Fluent state machine*; a states for *Action state machine*; y states for *Communication Function state machine*; x states for *Message state machine*.

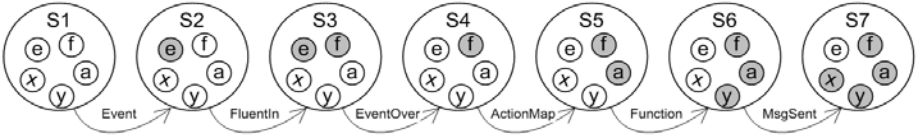


Fig. 6. Voyager AE product machine

Moreover, white circles present "idle" state and gray circles present the corresponding "active" state of the sub-tier state machine under consideration (such as: prompted for events, initiated for fluents, etc.; see Fig. 5).

Therefore, the formal presentation $(S; Op; R; S_0; AP; L)$ (see Section 4.1) of the Voyager AE ASG is:

- $S = \{S_1; S_2; S_3; S_4; S_5; S_6; S_7\}$
- $Op = \{Event; FluentIn; EventOver; ActionMap; Function; MsgSent\}$
- $R = \{(S_1; S_2; Event); (S_2; S_3; FluentIn); (S_3; S_4; EventOver); (S_4; S_5; ActionMap); (S_5; S_6; Function); (S_6; S_7; MsgSent)\}$
- $S_0 = S_1$ (initial state)
- $AP = \{ \text{event } pictureTaken \text{ occurs, event } pictureTaken \text{ terminates, action } processPicture \text{ starts, fluent } inProcessingPicturePixels \text{ initiates, function } sendBeginSessionMsgs \text{ starts, sends message } x \}$
- $L(S)$:
 - $L(S_1) = \{ \text{event } pictureTaken \text{ occurs } \};$
 - $L(S_2) = \{ \text{fluent } inProcessingPicturePixels \text{ initiates } \};$
 - $L(S_3) = \{ \text{event } pictureTaken \text{ terminates } \};$
 - $L(S_4) = \{ \text{action } processPicture \text{ starts } \};$
 - $L(S_5) = \{ \text{function } sendBeginSessionMsgs \text{ starts } \};$
 - $L(S_6) = \{ \text{sends message } x \};$

Moreover, we consider the following correctness properties applicable to our case:

- *If an event occurs eventually a fluent initiates.*
- *If an event occurs next eventually it terminates.*
- *If a fluent initiates next actions start.*
- *If an action starts eventually a function starts.*
- *If a function starts eventually it sends a message.*

The ASSL model-checking mechanism uses the correctness property formulae to check if these are held over product states considering the atomic propositions AP true for every state. Thus, the ASSL framework is able to trace the state path shown in Fig. 6 and to validate the *liveness property* stated above. Note that in this example, we intentionally presented a limited set of atomic propositions AP and correctness properties. The former are derivable, that is, deduced from the operational evaluation of the ASSL specification. Moreover, the Voyager AE product machine presents only product states relevant to our case study.

5 Automatic Test Case Generation with ASSL

To allow post-implementation software verification with the ASSL framework, we are currently developing a novel *test-generator tool* based on change-impact analysis that will help the ASSL framework automatically generate test suites for self-managing policies. Conceptually, the test generator tool accepts as input an ASSL specification (see Section 2) comprising sets of policies Π that need to be tested and generates a set of test cases T as tuples $T \{P_{ex}, A \{I, R\}\}$ comprising an execution path P_{ex} and test attributes A . The latter is a tuple comprising needed inputs I and optional replacement ASSL constructs R . The replacement ASSL constructs are automatically or semi-automatically specified and generated as supplementary software stubs to ensure the execution of P_{ex} .

Table 2 presents a `privateMessageInsecure` replacement event that is intended to replace the original `privateMessageInsecure` event. As shown, the replacement event guarantees that this event will occur in the system because: 1) it does not have a **GUARDS** clause that prevents the event from firing if special conditions are not met; and 2) its activation (see the **ACTIVATION** clause in Table 2) is time-ensured; i.e., it does not depend on external factors.

Table 2. Original and replacement ASSL events

Original Event	Replacement Event
<pre> EVENT privateMessageInsecure { GUARDS { NOT METRICS.thereIsInsecureMsg } ACTIVATION { CHANGED { METRICS.thereIsInsecureMsg } } } </pre>	<pre> EVENT privateMessageInsecure { ACTIVATION { PERIOD { 1 min } } } </pre>

5.1 Test Generation Methodology

5.1.1 Policy Execution Paths

Formally, from a policy execution perspective, an ASSL-specified self-management policy π may be presented as a tuple:

$$\pi \{F, A\}$$

where F presents the fluents driving the policy in question and A presents the actions that eventually will be undertaken while the policy is active. Here, for each fluent $f \in F$ we have:

$$f \{Ea, Af, Et\}$$

where Ea and Et are the sets of fluent-activating and fluent-terminating events respectively and $Af \subset A$ is the set of actions to be executed by f . Further, an event:

$e \in Ea \cup Et$ is a tuple $e \{grd, act\}$

where grd is the **GUARDS** clause and act is the **ACTIVATION** clause of the event e . Finally, an action $a \in A$ is a tuple:

$a \{grd, ens, Etr, Eer\}$

where grd and ens are the action's **GUARDS** and **ENSURES** clauses (state post-conditions that must be met after the action execution [1, 2]) respectively, and Etr and Eer are sets of events triggered by the action a in case of normal and erroneous action execution.

The execution of a policy π is activation and termination of the policy's fluents. Thus, to trace the policy execution, we must consider the execution paths of all the policy's fluents F . The execution path of a fluent is a sequence of the form:

$\{Ea, Af, Et\}$

The number of execution paths of a fluent with n activation events Ea , m termination events Et , and k actions A is a product:

$m \times n \times v(k)$

where the function $v(k)$ gives the variations in the execution of A . This function takes into account the action's formal attributes: grd , ens , Etr , and Eer , together with their internal dependencies and ASSL formal semantics [1, 2] as following:

- Etr and Eer are mutually exclusive, i.e., both cannot co-exist in same execution path;
- if ens is not met (denoted as $!ens$), then Eer is mandatory;
- if grd is not met (denoted as $!grd$), then the action a is not executed (denoted as $!a$).

Note that to simplify the problem, in this formal model we consider events as activated or not activated, thus helping us generalize over the event's clauses **GUARDS** and **ACTIVATION**. To illustrate the formal model, we present a simple example of a fluent

$f \{Ea, Af, Et\}$

where $n = 1$, $m = 1$, $k = 2$, and:

$Ea = \{ea1\}$

$Et = \{et1\}$

$Af = \{a1; a2\}$

$a1 = \{grd1; ens1; Etr1; Eer1\}$

$a2 = \{grd2; ens2; Etr2; Eer2\}$

Here, the possible execution paths of the fluent f are:

$Pex1 = \{ea1, a1\{grd1, ens1, Etr1\}, a2\{grd2, ens2, Etr2\}, et1\};$
 $Pex2 = \{ea1, a1\{grd1, ens1, Etr1\}, !a2\{!grd2\}, et1\};$
 $Pex3 = \{ea1, a1\{grd1, ens1, Etr1\}, a2\{grd2, !ens2, Eer2\}, et1\};$
 $Pex4 = \{ea1, a1\{grd1, ens1, Etr1\}, a2\{grd2, ens2, Eer2\}, et1\};$
 $Pex5 = \{ea1, !a1\{!grd1\}, a2\{grd2, ens2, Etr2\}, et1\};$
 $Pex6 = \{ea1, !a1\{!grd1\}, a2\{!grd2\}, et1\};$
 $Pex7 = \{ea1, !a1\{!grd1\}, a2\{grd2, !ens2, Eer2\}, et1\};$
 $Pex8 = \{ea1, !a1\{!grd1\}, a2\{grd2, ens2, Eer2\}, et1\};$
 $Pex9 = \{ea1, a1\{grd1, !ens1, Eer1\}, a2\{grd2, ens2, Etr2\}, et1\};$
 $Pex10 = \{ea1, a1\{grd1, !ens1, Eer1\}, !a2\{!grd2\}, et1\};$
 $Pex11 = \{ea1, a1\{grd1, !ens1, Eer1\}, a2\{grd2, !ens2, Eer2\}, et1\};$
 $Pex12 = \{ea1, a1\{grd1, !ens1, Eer1\}, a2\{grd2, ens2, Eer2\}, et1\};$
 $Pex13 = \{ea1, a1\{grd1, ens1, Eer1\}, a2\{grd2, ens2, Etr2\}, et1\};$
 $Pex14 = \{ea1, a1\{grd1, ens1, Eer1\}, !a2\{!grd2\}, et1\};$
 $Pex15 = \{ea1, a1\{grd1, ens1, Eer1\}, a2\{grd2, !ens2, Eer2\}, et1\};$
 $Pex16 = \{ea1, a1\{grd1, ens1, Eer1\}, a2\{grd2, ens2, Eer2\}, et1\};$

5.1.2 ASSL Test Generator

With the ASSL Test Generator we are aiming at a novel tool based on change-impact analysis that helps the ASSL framework automatically generate high-quality test suites for self-management policies.

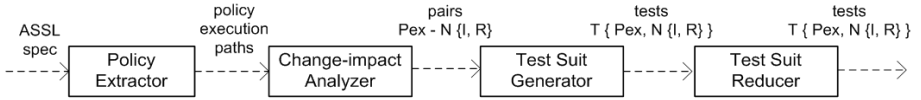


Fig. 7. Operational view of the ASSL Test Generator

As shown in Fig. 7, the test generator tool consists of four major components: *policy extractor*, *change-impact analyzer*, *test suit generator*, and *test suit reducer*. The key notion of the tool is to synthesize two or more execution paths of the same policy in such a way that test coverage targets (e.g., certain policies, rules, or conditions) are covered by the synthesized execution paths. The change-impact analysis component can then determine for each execution path the needed test attributes N such as inputs I and optional replacement constructs R in the form of ASSL events, ASSL actions, and **ACTIVATION**, **GUARDS**, and **ENSURES** clauses, needed to be employed by an execution path in order to ensure the same.

Based on the determined test attributes and execution paths, the tool generates tests T . Often the number of generated tests is large (recall that the number of fluent execution paths is a product of the number of events and actions employed by a fluent) and it is not feasible for developers to manually inspect their responses. To mitigate this issue, the final step of the test generator tool reduces the number of generated tests by selecting tests based on policy structural coverage.

5.1.3 Change-Impact Analysis

The goal of change-impact analysis is to determine what should be changed in the events and actions employed by a particular fluent execution path P_{ex} in order to ensure the same. In general, ASSL facilitates change-impact analysis because ASSL specifications allow:

- 1) extraction of information from the model to see where a change must occur in order to force one or more execution paths;
- 2) calculation of the change impact on the other parts of the model for any proposed change.

Here, of major importance the evaluation of how the execution of a fluent will be affected by a change in a particular event (**GUARDS** or **ACTIVATION** clause) or action (**GUARDS** or **ENSURES** clause). Note that at the time of writing, we are working on the change-impact analysis heuristic algorithm. Our initial results have demonstrated that this algorithm should involve the following logical steps.

- A. Evaluate what the conditions that must be met to have a specific fluent execution path ensured are:
 - a. Evaluate the events employed by a specific fluent:
 - 1) For each event analyze the pre-conditions that must be met (**GUARDS** clause) and the activation conditions (**ACTIVATION** clause);
 - 2) Evaluate if a particular event drives (activates or terminates) multiple fluents.
 - b. Evaluate the actions employed by a specific fluent:
 - 1) For each action analyze the pre- and post-conditions that must be met (**GUARDS** and **ENSURES** clause) and the events that are triggered by the action (**TRIGGERS** and **ONERR_TRIGGERS** clauses);
 - 2) Evaluate if the action itself executes other ASSL actions, or other executable constructs that may have impact on events such as ASSL *interaction functions* and ASSL *managed element functions* (both are sub-tiers in the ASSL specification model [1, 2]).
 - c. Generate a test case that meets the fluent execution path's conditions. Replacement constructs must be generated when the original ones cannot ensure the path execution. For example, if an event cannot be triggered due to conditions that must be met new replacement event may be generated that simulates the old one.
- B. Evaluate what the impact of having two or more fluent executing simultaneously is and what the conditions that must be met for that are. Generate test cases.
- C. Evaluate the policies involved in the tested execution path for the presence of chained fluents (the termination of a fluent activates another one, and so on). Find the conditions that must be met for that. Generate test cases.

In addition, it is important to evaluate the impact of modifying an existing construct and that of replacing the same construct with a completely new one. Another aspect that must be addressed by the change-impact analysis is the tradeoffs stemming from disabling **GUARDS** and **ENSURES** clauses. Note that such clauses act as special *behavior constraints* and are usually specified to ensure that certain conditions are met

before processing (or terminating) actions or events. Therefore, by disabling (removing) those constraints (see Table 2), we may ensure certain execution paths, but the impact of such a change needs to be also analyzed in the context of tradeoffs coming with the *unconstrained behavior*.

6 Conclusion and Future Work

We have presented software verification mechanisms for ASs (autonomic systems) developed with the ASSL framework. The family of software-verification framework tools includes: *consistency checker*, *model checker*, and *test case generator*. Currently, the ASSL consistency checker is the only fully implemented tool. It automatically checks ASSL specifications for consistency errors and some design flaws. The latter are verified against special consistency rules implemented as semantic definitions.

We have also presented our experiences to-date in developing the model checker and test case generator tools for ASSL. To implement the model checker, we developed program structures and algorithms that help an ASSL specification be transformed into a state-transition graph composed of special tier states with associated atomic propositions and transition relations connecting those states. We are currently developing a model-checking engine that works on the state transition graph. In addition, possible solutions to the so-called state-explosion problem are considered.

The test case generator tool aims at automatic generation of test suites for self-management policies. A test case is generated with a policy-execution path and test attributes that come in the form of inputs and special replacement ASSL constructs ensuring the execution of a tested policy. The test attributes are determined by change-impact analysis of the effect of a change in particular events or particular actions employed by an execution path. It is our understanding that such a testing mechanism is going to have a great impact on the development of prototype models for current and future space-exploration missions. Properly tested prototypes, eventually, will lead to the construction of more reliable spacecraft systems. Note that traditional methods, such as analyzing each requirement and developing test cases to verify the correctness of ASSL-implemented ASs, are not effective, because they require complete understanding of the overall complex system's self-management behavior.

Our plans for future work are mainly concerned with further development of the model checker and test-case generator tools for ASSL. Further, we plan to generate test cases for a number of self-managing policies developed for ANTS to determine the effectiveness of this approach as a test-covering and test-generation strategy. Moreover, it is our intention to build an animation tool for ASSL that will help to visualize counterexamples and trace erroneous execution paths.

Acknowledgment

This work was supported in part by Science Foundation Ireland grant 03/CE2/I303_1 to Lero—the Irish Software Engineering Research Centre.

References

1. Vassev, E.: Towards a Framework for Specification and Code Generation of Autonomic Systems. PhD Thesis, Computer Science and Software Engineering Department, Concordia University, Quebec, Canada (2008)
2. Vassev, E.: ASSL: Autonomic System Specification Language - A Framework for Specification and Code Generation of Autonomic Systems. LAP Lambert Academic Publishing (2009)
3. Murch, R.: Autonomic Computing: On Demand Series. IBM Press (2004)
4. Vassev, E., Hinchey, M., Paquet, J.: Towards an ASSL Specification Model for NASA Swarm-Based Exploration Missions. In: Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC 2008), pp. 1652–1657. ACM, New York (2008)
5. Vassev, E., Hinchey, M.: Modeling the Image-processing Behavior of the NASA Voyager Mission with ASSL. In: Proceedings of the Third IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT 2009), pp. 246–253. IEEE Computer Society, Los Alamitos (2009)
6. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
7. Vassev, E., Hinchey, M., Quigley, A.: Model Checking for Autonomic Systems Specified with ASSL. In: Proceedings of the First NASA Formal Methods Symposium (NFM 2009), NASA, pp. 16–25 (2009)
8. Bakera, M., Wagner, C., Margaria, T., Vassev, E., Hinchey, M., Steffen, B.: Component-oriented Behavior Extraction for Autonomic System Design. In: Proceedings of the First NASA Formal Methods Symposium (NFM 2009), NASA, pp. 66–75 (2009)
9. Visser, W., Havelund, K., Brat, G., Park, S.-J.: Model Checking Programs. In: Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE 2000). IEEE Computer Society, Los Alamitos (2000)
10. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, Boston (2003)
11. Bakera, M., Renner, C.: GEAR - Game-based, Easy and Reverse Model Checking (2008), <http://jabcs.cs.tu-dortmund.de/modelchecking/>

Modeling Language Variability

Hans Grönniger and Bernhard Rumpe

Software Engineering
RWTH Aachen University, Germany

<http://www.se-rwth.de>

Abstract. A systematic way of defining variants of a modeling language is useful for adapting the language to domain or project specific needs. Variants can be obtained by adapting the syntax or semantics of the language. In this paper, we take a formal approach to define modeling language variability and show how this helps to reason about language variants, models, and their semantics formally. We introduce the notion of *semantic language refinement* meaning that one semantics variant is implied by another. Leaving open all variation points that a modeling language offers yields the notion of the *inner semantics* of that language. Properties of the modeling language which do not depend on the selection of specific variants are called *invariant language properties* with respect to a variation point. These properties consequently follow from the inner semantics of a model or language.

1 Introduction

It has often been stressed that software is one of the most important drivers for innovation in many branches of industry. Developers are faced with the challenge to produce high quality, increasingly complex solutions in a short period of time.

Model-based software development is regarded as one instrument to cope with the challenges. Standard modeling languages like UML [OMG09] or domain specific languages (DSLs) are employed to increase the level of abstraction and automation while at the same time lowering the complexity. Especially in the context of robust, reliable systems development, the modeling languages used have to be defined precisely to allow for rigorous analysis of models and correct code generation.

The precise definition of a modeling language involves syntax and semantics [HR04]. Formal semantics is advantageous because it helps to avoid misunderstandings between people and may enable interoperability between tools. But even if a formal modeling language exists, a new class of systems like highly robust and reliable systems or a specific application domain may require adaptation of the language. A language may be changed to incorporate new language constructs, to disallow others for methodological or safety reasons, or to be semantically adjusted to a specific platform. This variability of a modeling language is subject of the paper.

We provide a formal account on language variability based on our classification in [CGR09]. On the one hand, the formalization brings light into how a

language can be adapted to specific requirements. On the other hand, it serves as a basis to define language variants formally. This allows us to reason about language (especially semantic) variants. The theoretical work is also equipped with tool support. Complete language definitions including all aspects of syntax and semantics and their variants are handled using the tools MontiCore [KRV08] and Isabelle/HOL [NPW02]. MontiCore is a framework for the development of modular (domain-specific) modeling languages while Isabelle/HOL is a theorem prover with higher-order logic and suitable to encode various language aspects.

The paper is structured as follows. The basic constituents (syntax, semantics) of a modeling language that may be subject to variability are introduced in Section 2. In Section 3, a formal characterization of language variants and a method to define variants is presented. As an example application, we outline how semantic variants can be compared formally in Section 4. In this section, we also introduce the concepts of semantic language refinement, inner semantics, and invariant language properties. Section 5 sketches the available tool support. In Section 6, we discuss related work. Section 7 concludes the paper.

2 Language Constituents

A precise definition of a modeling language consists of the following elements, see also [HR04, CGR09].

Concrete Syntax. The concrete syntax is the representation of the model with which a user interacts. This may be a graphical or textual notation or a mixture of both. We denote the set of all models of a modeling language in concrete syntax by \mathcal{CS} .

Abstract Syntax. The abstract syntax represents the structural essence of a language [Wi197]. For a textual syntax this may be given as abstract syntax trees generated by a parser. In case of graphical models, metamodels (e.g., defined in MOF [OMG06a]) are typically used. The set of all models of a modeling language in abstract syntax is denoted by \mathcal{AS} .

Additionally, a set of well-formedness rules or context conditions is defined to rule out certain models based on syntactic criteria. A typical example is that, in an automaton language, sources and targets of transitions have to exist so there are no dangling start or end points. But also the question of whether a model, e.g., a class diagram containing OCL constraints, is well-typed is addressed on the syntactical level. We assume a predicate

$$\text{wellformed} : \mathcal{AS} \rightarrow \text{bool}$$

to decide if a model is well-formed. The set of all well-formed models \mathcal{AS}^{wf} of a language hence is

$$\mathcal{AS}^{\text{wf}} = \{m \in \mathcal{AS} \mid \text{wellformed } m\}$$

We may also define additional constraints that rule out models for methodological or safety reasons, potentially restricting the expressiveness of the language. A more detailed explanation will be given in the next section.

A model in concrete syntax is associated with (or mapped to) a model in abstract syntax. Since typically not all models from \mathcal{CS} are well-formed, parsing is a partial mapping from concrete to abstract syntax:

$$p : \mathcal{CS} \rightarrow \mathcal{AS}^{\text{wf}}$$

Reduced Abstract Syntax. It is often advisable to reduce the number of language constructs for a simplification of semantic considerations. This is possible for each language construct that can be expressed by others (such as \forall by $\neg\exists\neg$ in predicate logic). This reduces set \mathcal{AS}^{wf} to a subset $\mathcal{AS}^{\text{red}} \subseteq \mathcal{AS}^{\text{wf}}$ with a syntactic transformation t to convert models into the reduced abstract syntax, i.e.,

$$t : \mathcal{AS}^{\text{wf}} \rightarrow \mathcal{AS}^{\text{red}} \text{ with } \mathcal{AS}^{\text{wf}} \supseteq \mathcal{AS}^{\text{red}}$$

Semantic Domain. By mapping models to elements of a semantic domain \mathcal{S} , the models obtain their meanings. The semantic domain is required to be well-known and understood and it should be based on a well-defined mathematical theory.

Our approach to semantics uses the system model [\[BCGR09a\]](#), [\[BCGR09b\]](#) which characterizes the structure, behavior, and interaction of objects in object-based systems. Thus, our focus is on semantics of object-based modeling languages. However, the variability mechanisms still apply if another semantic domain is used. The system model definitions are built on simple mathematical concepts like sets, relations, and functions. It is important to note that one element in the system model represents a single, complete object-based system. This means that the meaning of a model is directly represented as properties of possible implementations. The system model is underspecified to allow, for example, freedom of implementation when mapping a model to executable code.

For later reference, we introduce but a few system model concepts. Generally, elements of object-based systems are introduced as elements of underspecified universes leaving open the exact structure or number of elements. There is, for each system $s \in \text{SystemModel}$, a set of class names (or just classes, for short) UCLASS_s . In the following, we leave out the index s but a specific system is assumed implicitly if not stated otherwise. A class C_1 may be in a subset relation to a class C_2 which is denoted as $(C_1, C_2) \in \text{sub} \subseteq \text{UCLASS} \times \text{UCLASS}$. There is also a set of operation names (method signatures) UOPN . With function $\text{classOf} : \text{UOPN} \rightarrow \text{UCLASS}$ the defining class for an operation is obtained. Function nameOf determines the name of the operation, function params yields the set of all possible parameter assignments and function resType gives the return type of an operation. Types are elements of a universe UTYPE and there is a carrier set of values from universe UVAL associated with each type: $\text{CAR} : \text{UTYPE} \rightarrow \wp(\text{UVAL})$. $\wp(X)$ denotes the set of all subsets of X (power set).

Semantic Mapping. The semantic mapping sem finally relates models of the reduced abstract syntax to elements of the semantic domain. Characteristic of our loose approach is a set-valued or predicative semantic mapping of the form

$$\text{sem} : \mathcal{AS}^{\text{red}} \rightarrow \wp(\mathcal{S})$$

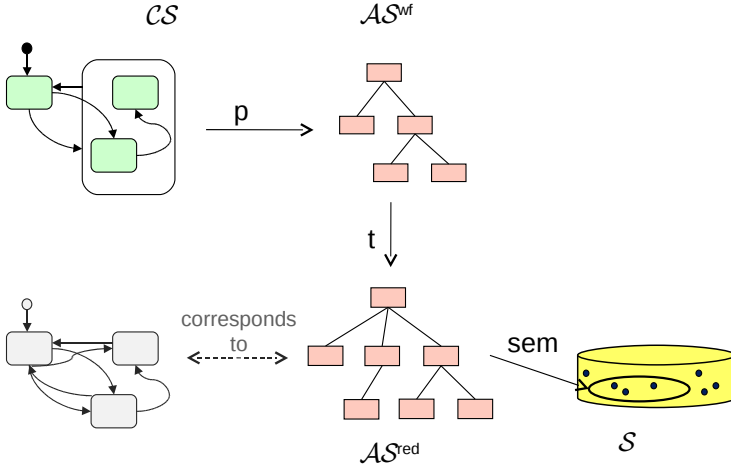


Fig. 1. From syntax to semantics for a Statechart model

The semantics of a model m is therefore the set $\text{sem}(m)$ of elements in the domain \mathcal{S} . If the system model is used for \mathcal{S} , then the model’s meaning is the set of all possible realizations.

Using the system model as a single semantic domain and the set-valued semantic mapping enable a straightforward treatment of composition and refinement of possibly incomplete and underspecified models of various modeling languages [Rum96]. For example, the integrated semantics of models m_1, \dots, m_n from possibly different languages is given as

$$\text{sem}_1(m_1) \cap \dots \cap \text{sem}_n(m_n)$$

In the same way, a model m' is a refinement of model m , exactly if

$$\text{sem}(m') \subseteq \text{sem}(m)$$

The whole chain from syntax to semantics is illustrated in Fig. 1. The example shows a hierarchical automaton (Statechart) in concrete syntax. Its abstract syntax is transformed into a conceptually reduced abstract syntax. For example, the automaton is flattened and the concept of hierarchy can be eliminated in the abstract syntax. Note that the (abstract) syntax of the resulting automaton will be more verbose compared to the original version. With the help of the semantic mapping, the automaton is mapped into the system model. Its semantics is given as a set of systems in the system model. These systems have to obey the properties introduced by the model. Hence, in the semantic mapping, we have to define ways to associate Statechart states with concepts in the system model (like classes, attributes, etc.). Additionally, we need means to encode the enabledness of transitions and their effect when actually executed.

3 Language Variants

A modeling language should be defined precisely but should not be completely fixed. Sustaining a certain degree of flexibility regarding a language’s syntax or semantics allows for adapting it to project or domain specific needs, or to enable modeling of new classes of systems. This idea has also been incorporated in the definition of UML where the informal semantics is equipped with semantic variation points subject to specific interpretation. At present, the UML standard itself regards semantic variation points as “less precisely defined dimensions” [OMG09]. We take a formal approach to define the possible variability in a language definition thereby substantiating our classification in [CGR09]. Afterwards, we present an intuitive way to document language variants.

3.1 Classification of Language Variability

In the previous section, we defined the constituents of a modeling language and their relations. A model in concrete syntax is translated into its abstract representation which then is (optionally) transformed into a conceptually simplified version. Based on this, the semantic mapping associates sets of elements of the semantic domain with the model. To summarize, we have the sequence

$$\mathcal{CS} \xrightarrow{p} \mathcal{AS}^{\text{wf}} \xrightarrow{t} \mathcal{AS}^{\text{red}} \xrightarrow{\text{sem}} \varphi(S)$$

In this section, we discuss means to define variants of a modeling language by adapting one or more elements of the above sequence.

Presentation Variability. A modeling language may offer *presentation options*, a term also coined in the UML standard. Presentation options allow for representing models differently in concrete syntax without changing a model’s abstract syntax. Formally, a language contains presentation options, if

$$\exists m_1, m_2 \in \mathcal{CS} : m_1 \neq m_2 \wedge p(m_1) = p(m_2)$$

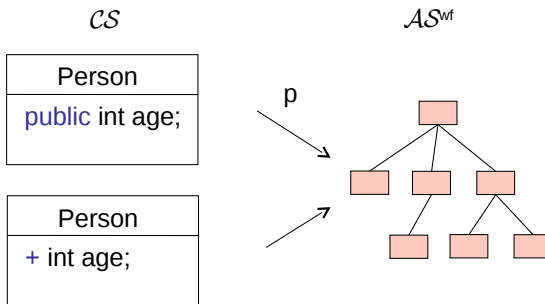


Fig. 2. Presentation option: Modifier representation in a class diagram

As shown in Fig. 2, for example, we have different ways to represent a public class modifier in UML: We can use the keyword `public` but equivalently the symbol `+`. The resulting abstract syntax, however, stays the same. Variants of presentation options result in changes of \mathcal{CS} and p , say \mathcal{CS}_v and p_v , by introducing, eliminating or changing existing ones. Models contained in both variants still have the same abstract syntax:

$$\forall m \in \mathcal{CS}_v \cap \mathcal{CS} : p_v(m) = p(m)$$

Additionally, every model can be expressed without choosing the presentation option variant:

$$\forall m_1 \in \text{dom}(p_v) : \exists m_2 \in \text{dom}(p) : p_v(m_1) = p(m_2)$$

Another form of presentation variability is what we call *abbreviations* or *extended constructs*: The syntax may contain certain constructs that help to enhance readability and comfort but which can be eliminated by some syntactic transformation t without losing expressiveness of the language. All models which do not use extended constructs remain identical under t , i.e.,

$$\forall m \in \mathcal{AS}^{\text{red}} : t(m) = m$$

The models that actually get transformed are contained in $\mathcal{AS}^{\text{wf}} \setminus \mathcal{AS}^{\text{red}}$. Variability in abbreviations means adapting \mathcal{AS}^{wf} and t , to $\mathcal{AS}_v^{\text{wf}}$ and t_v say. Consider, for example, a reduced abstract syntax for Statecharts $\mathcal{AS}^{\text{red}}$ which contains flat automata only (see Fig. 1). Hierarchy can be added to or removed from Statecharts without changing expressiveness [Rum04], but we obtain a larger set of expressible models when adding hierarchy, i.e., $\mathcal{AS}_v^{\text{wf}} \supseteq \mathcal{AS}^{\text{red}}$. Models that do not contain an extended construct variant (e.g., hierarchy) are transformed equally under t_v :

$$\forall m \in \text{dom}(t_v) \cap \text{dom}(t) : t_v(m) = t(m)$$

And we can still represent each model without the abbreviation:

$$\forall m_1 \in \text{dom}(t_v) : \exists m_2 \in \text{dom}(t) : t_v(m_1) = t(m_2)$$

As abbreviations do not show up in the reduced abstract syntax, semantics of these constructs is defined in two steps, the first one being the transformation to $\mathcal{AS}^{\text{red}}$ for which semantics is defined via the semantic mapping sem . Summarizing, variants of presentation options have an effect on the concrete syntax. Variants of abbreviations have an effect on the full abstract syntax. Both do not change the reduced abstract syntax and are called presentation variability.

Syntactic Variability. We now consider language variants that also have an impact on the reduced abstract syntax $\mathcal{AS}^{\text{red}}$. The syntax of a language may allow the use of *stereotypes*. A set of defined stereotypes (e.g., as part of a profile in case

of UML) is a syntactic variant of the language. We assume a function variant $\text{allowedStereotypes}_v$ that checks if only the chosen stereotypes are used, i.e.,

$$\mathcal{AS}_v^{\text{red}} = \{m \in \mathcal{AS}^{\text{red}} \mid \text{allowedStereotypes}_v(m)\}$$

An example for this kind of variability are priorities of transitions in a Statechart. A stereotype `<<prio:outer>>` attached to a hierarchical Statechart model would override the default priority rule that the innermost enabled transition is taken if there are multiple transitions with the same trigger enabled in the same step, see also [\[Rum04\]](#).

Another form of syntactic variability is given by so called *language parameters*, also termed language embedding in [\[KRV08\]](#). Consider again, for example, the language of Statecharts in which transitions may be guarded by a precondition. The language in which this condition is expressed is not specified. A natural candidate language would be OCL [\[OMG06b\]](#) but we may allow any other constraint language or a variant thereof that is suitable for the intended application. Hence, a syntax can be equipped with parameters $\mathcal{AS}^{\text{red}}(p_1, \dots, p_n)$. Variants can then be specified by assigning concrete languages to the parameters p_1, \dots, p_n .

As a last form of syntactic variability, we consider general *language constraints*. A language is further constrained to disallow certain models syntactically. It may be the case that this results in a less expressive language. Formally, a variant $\mathcal{AS}_v^{\text{red}}$ is given by models which fulfill further constraints stated, for example, in the predicate constr_v :

$$\mathcal{AS}_v^{\text{red}} = \{m \in \mathcal{AS}^{\text{red}} \mid \text{constr}_v(m)\}$$

The expressiveness of the language is preserved if

$$\forall m_1 \in \mathcal{AS}_v^{\text{red}} : \exists m_2 \in \mathcal{AS}^{\text{red}} : \text{sem}(m_1) = \text{sem}(m_2)$$

It is, for example, the goal of modeling or programming guidelines [\[Mat07, MIS\]](#) to restrict the use of certain (e.g., unsafe) language constructs while preserving the expressiveness. Restricting the expressiveness might be useful in situations in which a target platform may not be powerful enough to implement the models.

Semantic Variability. While UML only uses the term semantic variation point, we further subdivide semantic variability into *semantic mapping variability* and *semantic domain variability*. A helpful analogy might be to see the variability of the semantic mapping similar to configuration options of a code generator while variability of the semantic domain has its analogy with properties of an underlying run-time system or target platform.

By selecting variants for the semantic domain \mathcal{S} , we obtain an adapted domain \mathcal{S}_v in which elements have certain additional properties, for example, encoded in a predicate prop_v :

$$\mathcal{S}_v = \{s \in \mathcal{S} \mid \text{prop}_v(s)\}$$

Regarding semantic domain variability, the system model already contains explicit variability in form of extensions through optional definitions. It provides, for instance, different notions of type-safe method overriding or optional constraints to allow single inheritance only.

As an example for semantic domain variability, we show two variants for type-safe overriding of operations in a subclass. The first variant contains the well-known formalization of co-variant extension of parameters and contra-variant restriction of return values for operations in the subclass [Mey97]. In the system model, this can be expressed by a subset relation on the sets of all possible parameters and all possible return values, respectively:

$$\begin{aligned} \forall op_1 \in \text{UOPN}, c \in \text{UCLASS} : c \text{ sub classOf}(op_1) &\implies \\ \exists op_2 \in \text{UOPN} : \text{classOf}(op_2) = c \wedge & \\ \text{nameOf}(op_1) = \text{nameOf}(op_2) \wedge & \\ \text{params}(op_1) \subseteq \text{params}(op_2) \wedge & \\ \text{CAR}(\text{resType}(op_1)) \supseteq \text{CAR}(\text{resType}(op_2)) & \end{aligned}$$

The second variant is stricter as it does not allow a modification of the operation's signature in terms of possible values for parameters and the return type:

$$\begin{aligned} \forall op_1 \in \text{UOPN}, c \in \text{UCLASS} : c \text{ sub classOf}(op_1) &\implies \\ \exists op_2 \in \text{UOPN} : \text{classOf}(op_2) = c \wedge & \\ \text{nameOf}(op_1) = \text{nameOf}(op_2) \wedge & \\ \text{params}(op_1) = \text{params}(op_2) \wedge & \\ \text{CAR}(\text{resType}(op_1)) = \text{CAR}(\text{resType}(op_2)) & \end{aligned}$$

Variants of a semantic mapping arise as alternative definitions of (parts of) the semantic mapping, for example

$$\text{sem}_{v_1}, \text{sem}_{v_2} : \mathcal{AS}^{\text{red}} \rightarrow \wp(\mathcal{S})$$

Considering a Statecharts semantics again, a mapping variant could be the different choices of representing Statecharts states (syntax) as, for example, a simple enumeration in a class or using the state pattern [GHJV95].

Note that semantic variability is transparent to the modeler. But it may be necessary to allow the modeler to select one or the other interpretation of a construct. We propose to model these interpretation choices as syntactic variability by providing corresponding stereotypes. A modeler can then select the semantics of certain constructs by using appropriate stereotypes. With this approach, we transfer semantic variation points to syntactic ones.

3.2 Documentation of Language Variability

We propose to model variation points and variants in a language by feature diagrams [CE00]. Fig. 3 contains a feature diagram representing a generic structure

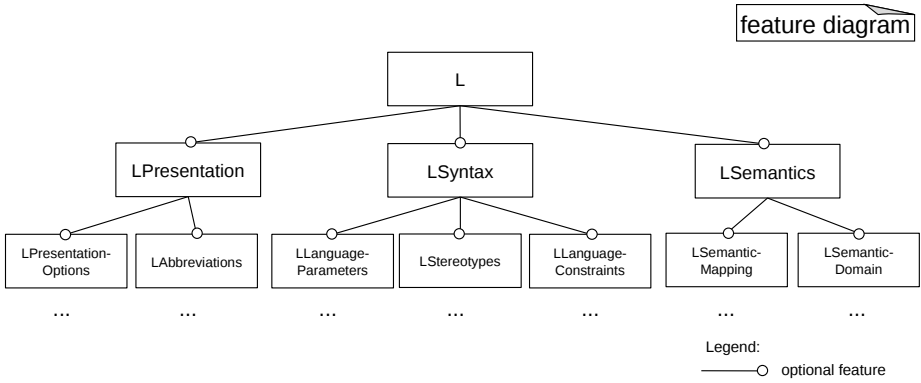


Fig. 3. Template to document variability of a language L

to model variants of a language L. We do not show concrete variants which depend on a specific language and which would be inserted under the corresponding nodes.

A supplement description of the variability can be given to explain their raison d’être and to point to formal definitions of the variants or other documentation. Since our main focus currently is on UML-like modeling languages, we refrained from expressing the variability in UML itself. This would certainly be possible but might be more confusing since UML models would be used also on the language definition level.

In Fig. 4 we present an excerpt of the feature diagram of a Statechart language containing the variants discussed previously in this paper. The choice of a feature

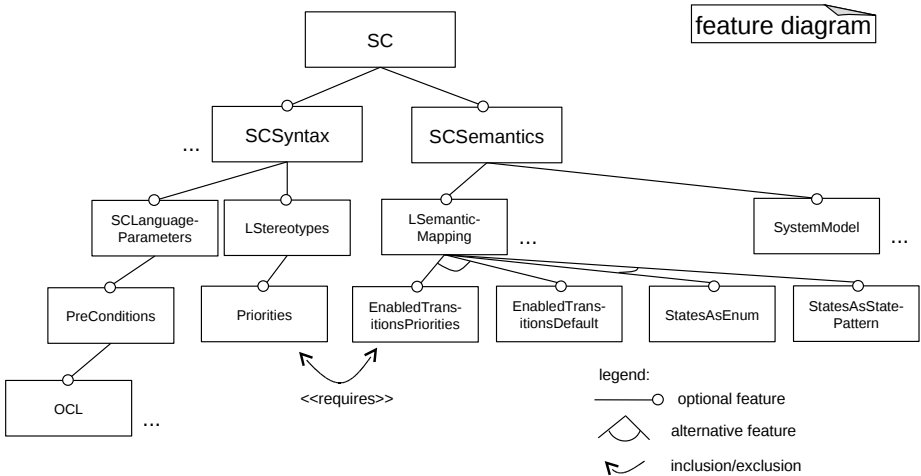


Fig. 4. Feature diagram

is always optional but there may be exclusive alternatives. It is, for example, not reasonable to choose both mappings for the decision of which transitions are enabled. Likewise we can only use one specific interpretation of Statechart states. There is an inclusion constraint for priority-based selection of transitions since priorities refer to syntactical as well as semantic variants at the same time (encode the priorities as stereotypes and select a semantic mapping that can handle them).

4 Comparison of Semantic Variants

Using our formal notion of language variants, it is possible to compare language variants formally and derive properties of the (relationship between) variants. Consider two semantic variants of the same language, e.g.,

$$\begin{aligned} \text{sem}_{v1} : \mathcal{AS}^{\text{red}} &\rightarrow \wp(\mathcal{S}_{v1}) \\ \text{sem}_{v2} : \mathcal{AS}^{\text{red}} &\rightarrow \wp(\mathcal{S}_{v2}) \end{aligned}$$

An interesting property is if variant $v2$ is a *semantic language refinement* of the semantic variant $v1$. Note that we discuss language refinement here and do not talk about refinement of models or the modeled system.

We define that language variant $v2$ is a semantic language refinement of variant $v1$ exactly if for all models the sets generated by the respective semantic mapping are in a subset relation, i.e.,

$$\forall m \in \mathcal{AS}^{\text{red}} : \text{sem}_{v1}(m) \supseteq \text{sem}_{v2}(m)$$

This implies that all properties ϕ of a model m which hold in variant $v1$ are preserved in variant $v2$:

$$\forall s \in \text{sem}_{v1}(m) : \phi(s) \implies \forall s \in \text{sem}_{v2}(m) : \phi(s)$$

Semantic language refinement is an important property if we consider for example tool integration. Assume that one tool for formal analysis uses (and correctly implements) language variant $v2$. Another tool for code generation correctly implements variant $v1$. If we show that variant $v2$ is a language refinement of $v1$ then we can be sure that analysis results obtained by the analysis tool are preserved in the second tool for code generation.

Let sem_{v1} be the Statechart semantics where the realization of states (either as an enumeration in a class or using the state pattern) is left open. Obviously, a semantic variant in which one of the alternatives is selected is always a subset of sem_{v1} for any model. A property ϕ which holds for sem_{v1} hence also holds under sem_{v2} . Since we did not select a specific variant in sem_{v1} , we say that ϕ is an *invariant property* with respect to the variation point on the interpretation of Statechart states. The property may be globally invariant (valid for all models) or locally invariant (for at least a single model). Not choosing any specific variant for any semantic variation point yields the notion of *inner semantics* of a modeling language. Properties shown for the inner semantics are intrinsic language properties and are agnostic to variant selection.

5 Tool Support

We have developed tool support in order to a) specify a machine-readable, checkable semantics that can directly be used for verification purposes, and b) to better control and quality check the different artifacts by using standard tools, e.g., version control. Fig. 5 gives an overview of the approach when defining the semantics of a language with tool support. First, the (domain specific) modeling language concepts are specified using a MontiCore grammar. MontiCore [KRV08] is a framework for the textual definition of languages based on an extended context-free grammar format. This format enables a modular development of the syntax of a language by providing modularity concepts like language inheritance and language parameters/embedding. MontiCore has an integrated, consistent definition of concrete and abstract syntax which also provides meta-modeling concepts like associations and inheritance [KRV07]. Framework functionality helps developers also to define well-formedness rules and, for example, the implementation of generators. To provide the semantics developer with maximum flexibility but also with some machine-checking (i.e., type checking) of the semantics and the potential for real verification applications, we use the theorem prover Isabelle/HOL for

- the formalization of the system model as a hierarchy of theories, including its semantic domain variants. This step has to be done once. All following language definitions which should be based on the system model can re-use this effort.
- the representation of the abstract syntax of the language as a deep embedding, including its syntactic variants. The translation of a MontiCore grammar to Isabelle/HOL abstract syntax data types is automated. Only manual configuration of variants is needed. We decided to follow a deep embedding approach (explicit representation of the abstract syntax in Isabelle/HOL) because this allows us also to reason about syntactical entities and the semantic mapping. With a shallow embedding (encoding properties of systems of the system model directly for a given model) this would not be possible.
- the semantic mapping which maps the generated abstract syntax to predicates over systems of the formalized system model, including its semantic mapping variants.
- specification of context conditions that may be helpful when doing verification on well-formed models, including variants in context conditions.

Details of the approach can also be found in [GRR09] and [Grö10]. We give a small example in which we prove in Isabelle/HOL that the stronger variant for type-safe operations refines the weaker variant. For this, we first need the encoding of the two predicates in Isabelle/HOL. The first one is given in Fig. 6. The figure shows an excerpt of an Isabelle/HOL theory which defines the predicate `valid-TypeSafeOps` as a function for systems in the system model (l. 1). In general, each definition in Isabelle/HOL is parameterized with the system model. As with the index s used earlier this means that the predicate is valid or not for a given system `sm`. For example, `UCLASS sm` (l. 4) is the specific universe

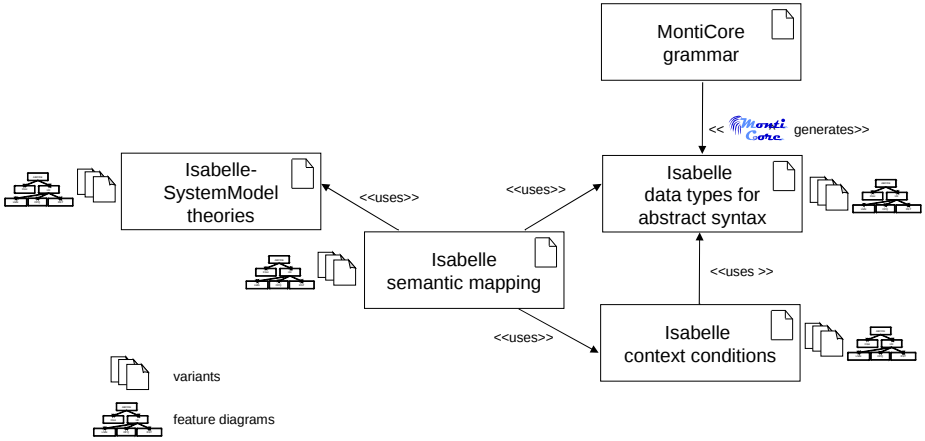


Fig. 5. Overview of approach to define a modeling language including its variants

```

TypeSafeOps
1 fun valid-TypeSafeOps :: "SystemModel  $\Rightarrow$  bool"
2 where
3   "valid-TypeSafeOps sm = (
4      $\forall$  op1  $\in$  UOPN sm .  $\forall$  C  $\in$  UCLASS sm .
5     (sub sm C (classOf sm op1))  $\longrightarrow$ 
6     ( $\exists$  op2  $\in$  UOPN sm .
7       classOf sm op2 = C  $\wedge$ 
8       nameOf op1 = nameOf op2  $\wedge$ 
9       params sm op1  $\subseteq$  params sm op2  $\wedge$ 
10      CAR sm (resType sm op2)  $\subseteq$  CAR sm (resType sm op1)
11    )"
12

```

Fig. 6. Part of a theory of the system model in Isabelle/HOL encoding the predicate of type-safe overriding of operations

of class names of the system **sm**. Similarly, other universes and functions are parameterized with the concrete system **sm**. Apart from slight notational differences, the predicate is a direct translation of the predicate given in Sect. 3.1. The predicate for the stronger variant is similar, only the subset relation is replaced by equality. We do not give the whole definition but the predicate is called `valid-TypeSafeOpsStrict`.

To prove the refinement from `TypeSafeOps` to `TypeSafeOpsStrict` we have to show that the set of systems with the second property is a subset of the set of systems with the first property. The actual proof is now given in Fig. 7. In Isabelle/HOL, this is a lemma which is given the name `TypeSafeOpsImplStrict` (l. 1). Applying the predicate definitions (ll. 4,5) the proof can be finished automatically by Isabelle (l. 6).

```

RefinedTypeSafeOps
1 lemma TypeSafeOpsImplStrict :
2   "{sm | sm . valid-TypeSafeOpsStrict sm}
3     ⊆ {sm | sm . valid-TypeSafeOps sm}"
4 apply (unfold valid-TypeSafeOps.simps)
5 apply (unfold valid-TypeSafeOpsStrict.simps)
6 by best

```

Fig. 7. Part of a theory containing the proof that the strict variant refines the weaker variant

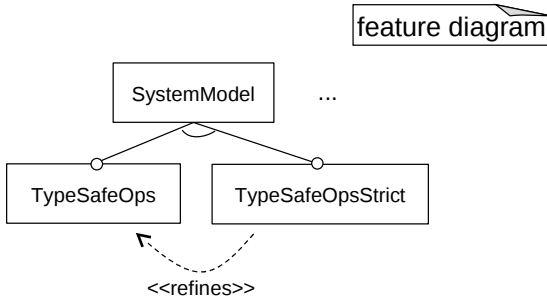


Fig. 8. Documentation of refinement relationship between variants in the feature diagram

Having shown that a variant refines another, we propose to update the feature diagram accordingly. Fig. 8 contains an excerpt of the feature diagram of the system model containing the information that `TypeSafeOpsStrict` is a semantic domain refinement of variant `TypeSafeOps`. As before, we assume that additional information, documentation may be attached to the feature diagram. This could be a link to the actual Isabelle/HOL proof that establishes the refinement relation. Other examples using the tool-supported approach can be found in [Grö10] and [GRR09].

6 Related Work

Presentation and semantic variants are also covered informally in the UML standard [OMG09]. We state precisely what kinds of variability may be found in a modeling language and document variants using feature diagrams.

Feature diagrams are also used in [Völ08] to define a family of architecture description languages. Formal semantics is not addressed. In the area of semantics, semantic variability is covered to some extent.

Template semantics [NAD03] as well as templatable metamodels [CMTG07] can be used to describe semantics with variation points. None of the mentioned work discusses the possibility to compare language variants. [TA06] examines

different variants of formal Statecharts semantics but does not address formal relationships between the variants.

Informal comparisons of Statecharts variants can, for example, be found in [Bee94, CD07].

Tool support to define modeling languages, including their formal semantics, is for example described in [CSAJ05]. This work presents semantic anchoring which means to transform the abstract syntax of a language into the abstract syntax of a language with known, formal semantics, for example Abstract State Machines (ASMs). [KM08] contains an Alloy-based approach that also allows to handle complete language definitions - from syntax, well-formedness of models to operational semantics. Mainly focusing on operational semantics these approaches have problems with underspecification and are not capable of integrating multiple languages into one common semantic domain easily.

7 Conclusion

We have formally described the constituents of a modeling language and how they can be varied to obtain modeling language variants. As an example application of precise modeling language variants, we have introduced the notion of semantic language refinement. Given two semantics variants of a language this notion defines if it is safe to use the one instead of the other variant. Additionally, we introduced the concept of inner semantics of a language, meaning to leave open all available variation points, and the notion of invariant properties with respect to a variation point. We have furthermore sketched the available tool support for complete language definitions with variability and how it can be applied to verify relationships between semantic variants.

Future work is concerned with investigating other relationships between language variants. Additionally, this work needs to be applied to, for example, the UML, or to various domain specific languages and needs to be explored in practice.

References

- [BCGR09a] Broy, M., Cengarle, M.V., Grönniger, H., Rumpe, B.: Considerations and Rationale for a UML System Model. In: Lano, K. (ed.) UML 2 Semantics and Applications. John Wiley & Sons, Chichester (2009)
- [BCGR09b] Broy, M., Cengarle, M.V., Grönniger, H., Rumpe, B.: Definition of the System Model. In: Lano, K. (ed.) UML 2 Semantics and Applications. John Wiley & Sons, Chichester (2009)
- [Bee94] von der Beeck, M.: A Comparison of Statecharts Variants. In: Langmaack, H., de Roeper, W.-P., Vytöpil, J. (eds.) FTRTFT 1994 and ProCoS 1994. LNCS, vol. 863, pp. 128–148. Springer, Heidelberg (1994)
- [CD07] Crane, M.L., Dingel, J.: UML vs. classical vs. rhapsody statecharts: not all models are created equal. *Software and System Modeling* 6(4), 415–435 (2007)

- [CE00] Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Reading (2000)
- [CGR09] Cengarle, M.V., Grönniger, H., Rumpe, B.: Variability within Modeling Language Definitions. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 670–684. Springer, Heidelberg (2009)
- [CMTG07] Cuccuru, A., Mraidha, C., Terrier, F., Gérard, S.: Enhancing UML Extensions with Operational Semantics. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 271–285. Springer, Heidelberg (2007)
- [CSAJ05] Chen, K., Sztipanovits, J., Abdelwalhed, S., Jackson, E.: Semantic Anchoring with Model Transformations. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 115–129. Springer, Heidelberg (2005)
- [GHJV95] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
- [Grö10] Grönniger, H.: Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten. Phd thesis, RWTH Aachen (2010) (in German)
- [GRR09] Grönniger, H., Ringert, J.O., Rumpe, B.: System Model-Based Definition of Modeling Language Semantics. In: Lee, D., Lopes, A., Poetzsch-Heffter, A. (eds.) FMOODS 2009. LNCS, vol. 5522, pp. 152–166. Springer, Heidelberg (2009)
- [HR04] Harel, D., Rumpe, B.: Meaningful Modeling: What’s the Semantics of “Semantics”? IEEE Computer 37(10), 64–72 (2004)
- [KM08] Kelsen, P., Ma, Q.: A Lightweight Approach for Defining the Formal Semantics of a Modeling Language. In: Busch, C., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 690–704. Springer, Heidelberg (2008)
- [KRV07] Krahn, H., Rumpe, B., Völkel, S.: Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 286–300. Springer, Heidelberg (2007)
- [KRV08] Krahn, H., Rumpe, B., Völkel, S.: MontiCore: Modular Development of Textual Domain Specific Languages. In: Objects, Components, Models and Patterns, TOOLS EUROPE 2008 (Proceedings). Lecture Notes in Business Information Processing, vol. 11, pp. 297–315. Springer, Heidelberg (2008)
- [Mat07] MathWorks Automotive Advisory Board (MAAB). Control Algorithm Modeling Guidelines Using Matlab, Simulink, and Stateflow – Version 2.1 (July 2007), <http://www.mathworks.com/automotive/standards/maab.html>
- [Mey97] Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice-Hall, Englewood Cliffs (1997)
- [MIS] MISRA C Website, <http://www.misra-c2.com/>
- [NAD03] Niu, J., Atlee, J.M., Day, N.A.: Template Semantics for Model-Based Notations. IEEE Trans. Software Eng. 29(10), 866–882 (2003)
- [NPW02] Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL – A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)

- [OMG06a] Object Management Group. Meta Object Facility Version 2.0 (2006-01-01) (January 2006), <http://www.omg.org/spec/MOF/2.0>
- [OMG06b] Object Management Group. Object Constraint Language Version 2.0 (2006-05-01) (May 2006), <http://www.omg.org/spec/OCL/2.0>
- [OMG09] Object Management Group. Unified Modeling Language: Superstructure Version 2.2 (2009-02-02) (February 2009), <http://www.omg.org/spec/UML/2.2>
- [Rum96] Rumpe, B.: Formale Methodik des Entwurfs verteilter objektorientierter Systeme. Doktorarbeit, Technische Universität München (1996)
- [Rum04] Rumpe, B.: Modellierung mit UML. Springer, Heidelberg (2004)
- [TA06] Taleghani, A., Atlee, J.M.: Semantic Variations Among UML StateMachines. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MODELS 2006. LNCS, vol. 4199, pp. 245–259. Springer, Heidelberg (2006)
- [Völ08] Völter, M.: A Family of Languages for Architecture Description. In: 8th OOPSLA Workshop on Domain-Specific Modeling (DSM) 2008 (Proceedings), pp. 86–93. University of Alabama, Birmingham (2008)
- [Wil97] Wile, D.S.: Toward a calculus for abstract syntax trees. In: Proceedings of the IFIP TC 2 WG 2.1 International Workshop on Algorithmic languages and Calculi, Alsace, France, pp. 324–353. Chapman & Hall, Ltd., Boca Raton (1997)

An Approach for Effective Design Space Exploration

Eunsuk Kang¹, Ethan Jackson², and Wolfram Schulte²

¹ Massachusetts Institute of Technology, Cambridge, MA, USA
eskang@mit.edu

² Microsoft Research, Redmond, WA, USA
{ejackson,schulte}@microsoft.com

Abstract. *Design space exploration* (DSE) refers to the activity of exploring design alternatives prior to implementation. The power to operate on the space of potential design candidates renders DSE useful for many engineering tasks, including rapid prototyping, optimization, and system integration. The main challenge in DSE arises from the sheer size of the design space that must be explored. Typically, a large system has millions, if not billions, of possibilities, and so enumerating every point in the design space is prohibitive. In this paper, we present a method for systematically exploring the design space in a cost-effective manner. The key idea is that many of the design candidates may be considered equivalent as far as the user is concerned, and so only a small subset of the space needs to be explored. Our approach takes the user-defined notion of equivalence, and generates *symmetry breaking predicates* to ensure that the underlying exploration engine does not sample multiple equivalent design candidates. We describe how the method is integrated into our DSE framework, *FORMULA*, which uses an SMT solver to solve a set of global design constraints and search for valid design instances.

1 Introduction

Design space exploration (DSE) refers to the activity of discovering and evaluating design alternatives during system development. It has many uses including:

- **Rapid prototyping:** DSE is used to generate a set of prototypes prior to implementation. Simulating and profiling of these prototypes can increase understanding of the impact of design decisions while taking complex system dynamics into account.
- **Optimization:** When metrics are available for comparing one design to another, DSE can be used to perform optimization, eliminating inferior designs and collecting a set of final candidates that are further studied.
- **System integration:** System integration requires the assembly and configuration of multiple components into a working whole. DSE can be used to find legal assemblies and configurations that satisfy a set of global design constraints.

DSE must be performed carefully because of the sheer number of design alternatives to be explored. A large complex system may admit millions, if not billions of design alternatives; in some cases, the design space may be infinite. A manual, ad-hoc approach to DSE is tedious, error-prone, and does not scale. An effective DSE framework must consist of the following ingredients:

- **Representation:** A suitable representation of the design space is essential. The representation should be formal, so that it can be subject to automated analysis and exploration techniques. A complex system may have a large number of design constraints that must be satisfied by every valid design solution. These constraints may involve arithmetic operations, Boolean expressions, and data type constraints over infinite domains. The representation should be expressive enough to capture these types of complex constraints.
- **Analysis:** A DSE framework must be equipped with machine-assisted techniques for discovering potential candidates, and checking them against the design constraints to ensure that they are actually valid design solutions. The framework must also be able to tackle the challenge of solving a large number of complex constraints at reasonable computational costs.
- **Exploration method:** Even after an optimization procedure rules out all inferior designs, the user may end up with the task of exploring a large number of design candidates. Enumerating them one-by-one in an ad-hoc fashion is not desirable. As far as the user is concerned, some of the solutions may be considered equivalent, and the user may be interested in examining only the ones that are distinctive from each other. The framework must provide a method for navigating to interesting solutions.

In previous papers [15,16], we proposed our DSE framework, called *FORMULA*, and discussed its representation of the design space and the underlying analysis engine, which is based on the *Z3* SMT solver [10]. In this paper, we describe the method in *FORMULA* for sampling a set of interesting design solutions. We say a solution is *interesting* if it is considered distinct from any other solution that has already been explored, under the user-defined notion of equivalence. Formally, two solutions are considered equivalent if their mathematical representations are *isomorphic* to each other. We show how we allow the user to define an equivalence relation that groups all isomorphic solutions into a single equivalence class. Our approach applies *symmetry breaking predicates* [8] to ensure that *FORMULA* returns exactly one solution from each equivalence class, thereby avoiding uninteresting designs from being presented to the user.

This paper is structured in the following way. We will begin by presenting a motivating example (Section 2). We will present background information on *FORMULA*—its representation of the design space and the SMT-based analysis engine for solving design constraints (Section 3). We will outline a method for exploring the design space in a way that guarantees only distinct design candidates to be found (Section 4). Then, we will discuss an experiment demonstrating the effectiveness of our approach (Section 5). Finally, we will conclude with discussions of related work (Section 6) and future directions (Section 7).

2 Motivating Example

We begin with an example that is simple but challenging for existing DSE methods. This example is borrowed from *platform mapping* problems found in automotive embedded systems [19,26]. Given a set of software *tasks* and *devices*, the goal is to map each task onto a device in such a way that a certain set of design constraints are satisfied.

We formally describe the problem as follows. Let T be a set of named tasks. A *conflict graph* $C = (T, E_C)$ is a labeled undirected graph over tasks. An edge $(t_1, t_2) \in E_C$ indicates that tasks t_1 and t_2 are in conflict and should not be executed on the same device. Let D be a set of named devices. Then, a *distributed network* $N = (D, E_N, cap)$ is a triple where (D, E_N) is a labeled *directed* graph. For each edge $(d_1, d_2) \in E_N$, there is a directed communication channel from d_1 to d_2 . The notation $in(d)$ indicates the set of incoming communication channels of device d , and $out(d)$ its outgoing channels. Every channel has a strictly positive *capacity* as assigned by the function $cap : E_N \rightarrow \mathbf{Z}_+$. Finally, tasks are bound to devices by the function $bind : T \rightarrow D$. The structures C , N , and $bind$ provide a representation for instances of the design space—i.e. possible configurations of tasks, devices, and mappings between them.

Every valid design of the system must satisfy the following design constraints:

1. A pair of conflicting tasks cannot be mapped onto the same device: $\forall t_1, t_2 \in E_C \cdot bind(t_1) \neq bind(t_2)$.
2. A single device can provide at maximum two ingoing and/or outgoing channels: $\forall d \in D \cdot |in(d)| \leq 2 \wedge |out(d)| \leq 2$.
3. Each device with both input and output channels must have balanced capacities:

$$\forall d \in D \cdot in(d) \neq \emptyset \wedge out(d) \neq \emptyset \Rightarrow \sum_{i \in in(d)} cap(i) = \sum_{o \in out(d)} cap(o)$$

Constraint (1) is equivalent to a graph coloring problem, and requires reasoning about the global topology of the system. Constraint (2) is a forbidden sub-graph problem. Constraint (3) requires arithmetic reasoning and is guarded by a Boolean constraint.

Let us assume the engineer has chosen a set of tasks and identified conflicts appropriately. Then, the possible design alternatives arise from variations in network topologies, capacities, and task bindings. Figure 1 shows one possible configuration of the system. Note that this instance satisfies all of the design constraints. If, for example, the capacity of the channel from device B were to be altered from 1 to 2, then the modified instance would fail to satisfy constraint (3), and no longer qualify as a legal design candidate.

From the perspective of the engineer, the “best” design might be one that utilizes the communication channels most efficiently. However, this utilization depends on the code executed by the tasks, the scheduling strategy of the underlying operating system, the communication protocols implementing channels, and a number of other factors. An optimization problem cannot be formulated easily at this high-level, and so rapid prototyping combined with simulation is the approach that is often taken to evaluate design alternatives [19].

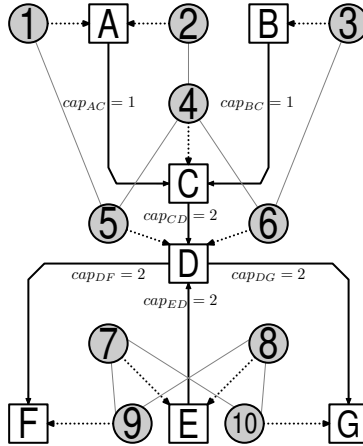


Fig. 1. A design instance of the platform mapping problem, representing one possible configuration of the system. Circles represent tasks, and squares represent devices. Every edge between a pair of devices is labeled with the channel capacity. This instance contains ten tasks, with conflicts indicated by gray lines. A dotted line from a task to a device indicates that the task has been mapped to the device.

The requirement on the DSE engine is to enumerate design alternatives that satisfy global constraints. However, this example is challenging for several reasons. First, instantiating a single solution requires solving non-trivial constraints, such as arithmetic and relational constraints. Second, the number of design alternatives is, in principle, infinite, because no bounds were placed on the channel capacities, the number of devices, or the domain of their labellings; certainly, the entire space cannot be explicitly enumerated. Third, labels on the devices are also a design parameter, and thus, each solution in the design space will have a large (possibly infinite) number of counterparts that differ only by labeling of devices. These counterparts may be of no interest to the user, and so the exploration method must be able to eliminate these equivalent solutions. As we shall show, this last criterion is particularly challenging to achieve.

3 Background on FORMULA

FORMULA is our modeling framework for formally specifying domain-specific languages [17]. From the DSL perspective, the representation and constraints of a design space form a domain-specific abstraction; DSLs are ideal for capturing such abstractions. Additionally, the DSL metaphor allows complex design spaces to be built from smaller ones using DSL composition operators. In this section we introduce just enough of FORMULA to encode our motivating example; please see [16] for a more detailed discussion.

```

1.   domain Functionality
2.   {
3.     Task      ::= (id: Basic).
4.     [Closed]
5.     Conflict  ::= (t1: Task, t2: Task).
6.   }
7.   model ThreeTasks of Functionality
8.   {
9.     Task(1)
10.    Task(2)
11.    Task(3)
12.    Conflict(Task(1), Task(2))
13.    Conflict(Task(2), Task(3))
14.   }

```

Fig. 2. Examples of a domain and a model specified in FORMULA. The `Functionality` domain encodes the abstraction related to tasks and conflicts. The model contains three tasks with two conflicts among the tasks.

3.1 Representation

A domain block encapsulates the data types and constraints of a DSL, as shown in Figure 2. A data type is either the name of a sort (a set of constants, e.g. `String`), a record constructor, or an arbitrary union of other data types. Line 3 declares a constructor called `Task`, which takes an `id` argument of type `Basic` (which corresponds to the set of all constants). Line 5 declares a constructor for denoting conflicts between tasks, which requires two arguments of type `Task`. FORMULA data types are algebraic: Two data instances are the same if and only if they were built from the same sequence of constructors and constants. This formalism captures inductive data types with type constraints. A model is a set of record instances built using the constructors of a domain that satisfy domain constraints (lines 7-14). The declaration `model ThreeTasks of Functionality` is a *claim* that the model satisfies constraints; the claim is verified by FORMULA.

Some domain constraints are quite common; e.g. conflict edges form a relation over tasks: $E_C \subseteq T \times T$. FORMULA provides built-in support for common constraints via annotations on data type declarations. The `[Closed]` annotation applied to the `Conflict` constructor is an example. Let $\llbracket C \rrbracket$ be the set of all well-typed records that can be constructed by C . If M is a set of records, then $M(C) = M \cap \llbracket C \rrbracket$ is the set of C -records in M . For example $M(\text{Task})$ and $M(\text{Conflict})$ is the set of all tasks/conflicts respectively. The *closed* annotation requires every model M to satisfy $\{(t1, t2) \mid \text{Conflict}(t1, t2) \in M(\text{Conflict})\} \subseteq M(\text{Task}) \times M(\text{Task})$.

In general, a rich constraint language is needed to specify domain constraints. Many modeling tools use the *object constraint language* (OCL) for this purpose. However, the intricacies of OCL complicate automated analysis of arbitrary OCL constraints [21]. For this reason, we choose *constraint logic programming* (CLP)

```

1.  domain Distribution
2.  {
3.    Device ::= (id: Basic).
4.    [PartialFunction(src, dst -> cap)]
5.    Channel ::= (src: Device, dst: Device, cap: PosInteger).
6.
7.    bigFanIn  :=d is Device, count(Channel(⊖, d, ⊖)) > 2.
8.    bigFanOut :=d is Device, count(Channel(d, ⊖, ⊖)) > 2.
9.    clog      :=d is Device,
10.         sum(Channel(⊖, d, ⊖), 2) != sum(Channel(d, ⊖, ⊖), 2).
11.    conforms  :=!(bigFanIn | bigFanOut | clog).
12.  }

```

Fig. 3. A domain representing the distribution of devices through channels

for the core constraint language of FORMULA. CLP is well studied, has an unambiguous execution semantics, and can be converted into first-order logic. In fact, FORMULA converts all built-in constraint annotations into logic programs.

Figure 3 shows an abstraction for the distributed network of devices through channels, which requires more complex constraints to specify. A `Channel` is a partial function from a pair of `Devices` to a positive integer. Line 7 defines a *query* for checking whether an input model M has a `Device` with too many incoming `Channels`. For each binding of the variable `d` to a `Device`, the `count` operator counts the number of distinct `Channels` terminating on `d` (the underscores are “don’t care” variables). If there is any binding of `d` with more than two incoming `Channels`, then the Boolean variable `bigFanIn` evaluates to true. The `bigFanOut` query (line 8) performs the same check for outgoing `Channels`. The `clog` query checks if the communication network is unbalanced by summing the capacities on incoming/outgoing `Channels`. The second argument of the `sum` operator is the zero-indexed field within the record that is summed.

Every FORMULA domain has a query called `conforms`. By definition, an input model satisfies domain constraints only if `conforms` evaluates to true. The design space associated with a domain is the set of models satisfying its `conforms` query. In the example from Figure 2, the `conforms` query has not been explicitly defined by the user; in this case, FORMULA will implicitly define the query as a conjunction of compiler generated constraints (e.g. `[Closed]`). In Figure 3, the `Distribution` domain explicitly requires that none of `bigFanIn`, `bigFanOut`, and `clog` should evaluate to true. The entire `conforms` query for `Distribution` also contains compiler generated constraints due to the annotation `[PartialFunction]`.

The DSL approach supports modular and compositional specification of abstractions. The `Architecture` domain (Figure 4) is an extension of the product of the `Functionality` and `Distribution` domains. These composition operations allow the `Architecture` domain to use the data structures of `Functionality` and `Distribution` while provably ensuring that all constraints are enforced the same way [16]. `Architecture` also adds a new data structure `Binding` and requires that `Bindings` must respect task conflicts (lines 18-19). Again, the complete `conforms` of


```

13.  domain Architecture extends Functionality, Distribution
14.  {
15.    [Function]
16.    Binding ::= (t: Task, d: Device).
17.
18.    conflict  := Binding(t1, d), Binding(t2, d), Conflict(t1, t2).
19.    conforms  := !conflict.
20.  }

```

Fig. 4. A domain as a composition of Functionality and Distribution

Architecture contains constraints imported from the other domains. In summary, the models conforming to **Architecture** are exactly those legal systems described in Section 2. The DSL approach allows the user to encode the interesting degrees of design freedom via formal and composable abstractions. Specifically, FORMULA utilizes algebraic data types and CLP to accomplish this.

3.2 Solving for Instances

In order to find non-trivial solutions to design spaces, FORMULA specifications are translated into the SMT solver Z3. Let $D.q$ be a query q defined in domain D , then the translation procedure must produce a first-order formula $\varphi[X]$ with the following property: Finite models (sets of records) satisfying $D.q$ are in correspondence with satisfying instances of $\varphi[X]$, where X denotes the vector of variables appearing in φ . A satisfying instance is a mapping of variables to values $\{x_1 \mapsto v_1, \dots, x_n \mapsto v_n\}$; a reverse translation converts satisfying instances into FORMULA models.

SMT solvers represent a significant step in automated theorem proving by soundly combining decision procedures for different theories while using efficient SAT-based backtracking techniques to drive the search process. For example, the clog query (lines 9-10, Figure 3) imparts the following fragment into φ :

$$\begin{aligned}
 & test_{Device}(d) \wedge test_{Channel}(in1) \wedge test_{Channel}(in2) \wedge \\
 & sel_{Channel,1}(in1) = d \wedge sel_{Channel,1}(in2) = d \wedge in1 \neq in2 \wedge \quad (1) \\
 & x = 2Int(sel_{Channel,2}(in1)) + 2Int(sel_{Channel,2}(in2)) \dots
 \end{aligned}$$

This fragment sums the incoming channel capacities for a device d with two distinct incoming channels. SAT techniques provide a strategy for satisfying sub-formulas, and specific decision procedures actually solve the sub-formulas. In this example, two decision procedures are required: (1) term algebras (TA) for inductive data types and (2) linear arithmetic for summing channels. The first line of the formula uses TA to test that the variables d , $in1$, and $in2$ have the appropriate record structure. The second line extracts the second components of the channels $in1$ and $in2$ using TA *selectors*; the equalities here invoke unification and the *occurs check*. The third line extracts the channel capacities, coerces them to integers using the function $2Int$, and calculates their sum via the linear arithmetic decision procedure.

This example illustrates the power of SMT, but also shows that the translation process from a high level specification to SMT is non-trivial, since most SMT solvers support only the existential fragment of first-order logic. In our approach, universal quantifiers are eliminated by *symbolically executing* a specification over a set of symbolic inputs and emitting all interesting branches of the logic program as a quantifier free formula. The symbolic execution loop is implemented outside of the theorem prover, and it takes as input a finite set of records with variables where constants would otherwise be. For example, symbolic execution on the following set:

$$S = \left\{ \begin{array}{l} Task(x_1), Task(x_2), Task(x_3), \\ Device(x_4), Device(x_5), \\ Conflict(x_6, x_7), \\ Channel(x_8, x_9, x_{10}), \\ Binding(x_{11}, x_{12}), Binding(x_{13}, x_{14}), Binding(x_{15}, x_{16}) \end{array} \right\} \quad (2)$$

produces a formula φ capturing all the possible ways that zero to three Tasks, zero to two Devices, etc... can satisfy design constraints. When the DSE procedure does not know cardinality bounds for all record types, then it repeatedly attempts larger and larger symbolic sets as input to the symbolic execution engine. Even though each symbolic input set has a finite number of records, the resulting SMT formula may still have an infinite number of solutions, because variables occurring in φ range over infinite domains. We refer to the original finite set used to produce φ via symbolic execution as the *generator set* of φ .

4 Design Space Exploration Method

After symbolic execution, elements of the design space can be enumerated by repeatedly querying the SMT solver. This procedure is not sufficient for rapidly exploring diverse solutions, because the solver does not know which solutions are considered similar. Also, solving strategies are optimized to find any *next* solution, and not necessarily solutions that are highly distinct. In this section, we describe a technique for grouping related solutions based on isomorphisms over algebraic data types.

4.1 Projection-Based Equivalence Partitioning

Let Σ be the set of constants that might appear in the field of some record. Let \mathcal{C} be the set of all constructors of a domain D . A *term homomorphism* ϕ is a function over constants lifted onto records. If $c(r_1, \dots, r_n)$ is a record built by applying constructor c to records r_1, \dots, r_n , then $\phi(c(r_1, \dots, r_n))$ returns a new record that is equal to $c(\phi(r_1), \dots, \phi(r_n))$. Term homomorphisms preserve the structure of records, but change the constants appearing in their fields. If M is a model (i.e. a set of records), then $\phi(M)$ is a model formed by applying ϕ to each record in M . Homomorphisms induce a preorder on models: $M' \preceq M$ if

$\exists \phi \cdot \phi(M) = M'$. Two models M, M' are *isomorphic* if $M' \preceq M$ and $M \preceq M'$; this can be written as $M \sim M'$.

Isomorphic models are equivalent up to relabeling of the constant values appearing in their records. Our approach groups all isomorphic solutions into a single equivalence class, and finds only one representative per equivalence class. We take this one step further, allowing isomorphisms to be considered on some subsets of data types, thereby further decreasing the number of distinct equivalence classes. We call $\Pi \subseteq \mathcal{C}$ a *projection* on the records of domain D . If M is a model, then $\Pi(M)$ discards records not in Π :

$$\Pi(M) = \{r \mid r = c(r_1, \dots, r_n) \wedge r \in M \wedge c \in \Pi\} \quad (3)$$

Given a projection Π , then $M' \preceq_{\Pi} M$ if $\exists \phi \cdot \phi(\Pi(M)) = \Pi(M')$. Again, two models are in the same equivalence class if and only if $M \preceq_{\Pi} M'$ and $M' \preceq_{\Pi} M$.

FORMULA provides the user with an interface for specifying a projection. Returning to the motivating example, suppose the user wants to see only those solutions with distinct channel topologies. In the case, the user may specify $\Pi = \{Channel\}$, and every solution returned by FORMULA will have a distinct communication topology.

This approach must be integrated with the solver, so it knows to return non-isomorphic solutions. Communicating isomorphism-based equivalence classes to solvers can be accomplished using *symmetry breaking predicates* [8].

4.2 Exploration Algorithms

Encoding equivalence classes into the SMT solver using symmetry breaking predicates can ensure that every new solution is non-isomorphic to previous ones. However, this alone does not diversify the exploration throughout the design space; in other words, FORMULA may consecutively return non-isomorphic but structurally similar solutions within a small portion of the space. Ideally, we want the solver to “jump around” various parts of the design space, sampling a wide variety of non-isomorphic solutions. In this section, we describe an algorithm that explores the design space for a particular generator set G , and show how we employ randomization to incrementally construct a diverse set of non-isomorphic solutions.

Naïve Exploration Algorithm. We begin by describing a simple candidate algorithm *Explore* (Figure 5) to build intuition. The algorithm accepts as inputs the generator set G , the formula φ generated from G , which encodes the design constraints, and a user-specified projection Π . The algorithm randomly samples an equivalence class in the design space, and then checks if that equivalence class contains a model satisfying φ . A sample s is a symbolic set of records under the projection Π . For example, given the generator set from Equation 2 in Section 3.2, and $\Pi = \{Binding\}$, one possible sample is:

$$\{Binding(x_1, x_2), Binding(x_3, x_2), Binding(x_4, x_5)\}. \quad (4)$$

```

Explore( $G, \varphi, \Pi$ )
1:  $solutions := \{\}$ 
2:  $sampled := \{\}$ 
3: while True do
4:    $s := SampleClass(G, \Pi)$ 
5:   for all  $p$  in  $sampled$  do
6:     if  $TestIsomorphism(s, p)$  then
7:       goto Line 3
8:     end if
9:   end for
10:   $sampled := sampled \cup \{s\}$ 
11:   $soln := FindModel(s \wedge distinct(s) \wedge \varphi)$ 
12:  if  $soln \neq NULL$  then
13:     $solutions := solutions \cup \{soln\}$ 
14:  end if
15:  if  $CheckExhaustive(sampled)$  then
16:    return  $solutions$ 
17:  end if
18: end while

```

Fig. 5. Naïve exploration algorithm

Note that the first two **Binding** terms contain the same variable for the second argument to the constructor (x_2). This sample represents the set of all design instances in which two of the tasks are mapped to the same device. Figure 6 provides a graphical illustration of the sample.

The basic algorithm consists of a single while loop. In each iteration, an equivalence class s in the design space is sampled based on the projection (line 4), and is discarded if it is isomorphic to any of the samples that *Explore* has visited (lines 5-9). This check avoids isomorphic solutions from being collected.

If the sample passes this check, then the SMT solver attempts to construct a solution to the new sample that satisfies all of the design constraints (line 11). In this procedure, all the variables appearing in the sample are constrained to be distinct ($distinct(s)$), ensuring that the solver does not return a homomorphic image of s . If the solver succeeds in finding a satisfying instance, it stores the instance into the set $solutions$. If not, it goes back to the beginning of the loop and attempts another sample.

The exploration loop terminates when *Explore* has visited all equivalence classes in the design space (line 15)¹. The termination condition is based on a property of the random sampling procedure (*SampleClass*): The probability that all classes are visited can be made arbitrarily close to 1 with a finite number of iterations.

How well does this algorithm work? Our goal is to collect a set of non-isomorphic solutions at a reasonable amount of computational cost. For this

¹ The exhaustive list of equivalence classes can be computed using a variant of Polya's enumeration theorem [28].

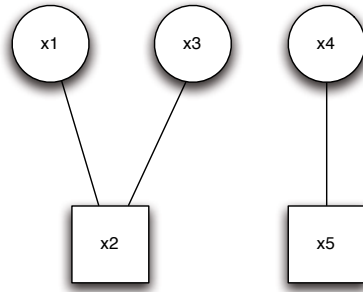


Fig. 6. A graphical representation of a sample for the platform mapping problem. Circles present tasks, and boxes represent devices. This particular sample represents the equivalence class of design solutions where two of the tasks are mapped to the same device.

purpose, we define the cost as the number of times that the algorithm invokes the SMT solver. Then, the success rate of the algorithm is the ratio of the number of non-isomorphic solutions to the number of calls to the SMT solver. A solving run that fails to find a satisfying instance is wasteful. But whether or not the SMT solver succeeds depends on the equivalence class that is picked by the sampling procedure. If the design space is loosely constrained, and a large number of equivalence classes contain models that satisfy constraints, then this simple algorithm should perform well. However, this assumption is often not true; the design space may be highly constrained, and random sampling may frequently pick samples that do not contain a satisfying instance. In practice, the algorithm performs poorly for a complex system that contains a large number of design constraints. This leads to a need for an algorithm that is able to avoid those parts of the design space that contain only invalid design instances.

Improved Algorithm. In Figure 7, we present an alternative algorithm *ExploreII*, which avoids the observed problem with the naïve algorithm. Two key ideas distinguish the new algorithm. First, we allow the solver to check a possibly exponential number of equivalence classes per invocation. This is accomplished by removing distinctness constraints on variables in a random sample; the solver is now free to equate variables in the sample when searching for a satisfying instance. Secondly, *ExploreII* incrementally learns the regions of the design space that contain only invalid designs and then avoids examining these designs.

The outline of the new algorithm is as follows. It keeps track of two sets of samples, *valid* and *blocked*, whose purposes will be explained in the following paragraphs. Like the previous algorithm, *ExploreII* consists of a single while loop, which begins by sampling a symbolic set s in the design space (line 5). However, unlike the previous algorithm, the variables in this sample are no longer constrained to be distinct; some of them may be equated if the solver decides to assign the same constant to them. As a result of the relaxation, s is no longer constrained to represent a single equivalence class, but can also represent *homomorphic* images of s spanning many equivalence classes. Consider Figure 8. The

```

ExploreII( $G, \varphi, \Pi$ )
1:  $solutions := \{\}$ 
2:  $valid := \{\}$ 
3:  $blocked := \{\}$ 
4: while  $True$  do
5:    $s := SampleClass(G, \Pi)$ 
6:   for all  $p$  in  $blocked$  do
7:     if  $TestHomomorphism(p, s)$  then
8:       goto Line 4
9:     end if
10:  end for
11:   $C := \{\}$ 
12:  for all  $q$  in  $valid$  do
13:     $C := C \cup ComputeHomomorphism(s, q)$ 
14:  end for
15:   $soln := FindModel(s \wedge \neg C \wedge \varphi)$ 
16:  if  $soln \neq NULL$  then
17:     $valid := valid \cup \{Simplify(s, soln)\}$ 
18:     $solutions := solutions \cup \{soln\}$ 
19:  else
20:    if  $CheckMostGeneral(s)$  then
21:      return  $solutions$ 
22:    end if
23:     $blocked := blocked \cup \{s\}$ 
24:  end if
25: end while

```

Fig. 7. Improved exploration algorithm

sample on the left hand side represents the equivalence class of design instances where two of the three tasks are mapped to the same device. By equating $x1$ and $x3$, we obtain a homomorphic image of the original sample; this image represents the equivalence class where each instance maps each task to exactly one device.

If a sample does not contain any instance that satisfies the given design constraints, then every homomorphic image of the sample will also be unsatisfiable. Thus, a new sample that is a homomorphic image of any element in $blocked$ is deemed invalid or redundant, and immediately discarded to save the solver from doing wasteful work (lines 6-10).

Since a sample may admit solutions in multiple equivalence classes, *ExploreII* must prevent the solver from returning a solution that it has already found. The procedure $ComputeHomomorphism(s, q)$ computes a homomorphism (if it exists) from s to q in the form of (dis)equalities over the variables in s and q . At the end of the loop on lines 12-14, C contains the set of all homomorphisms from s to the elements in $valid$. The negation of the disjunction of the constraints in C (represented by $\neg C$), prevents the solver from equating variables in s in a way that would map s into one of the equivalence classes in $valid$. In other words, $\neg C$ is the symmetry breaking predicate that guarantees that the solver does not

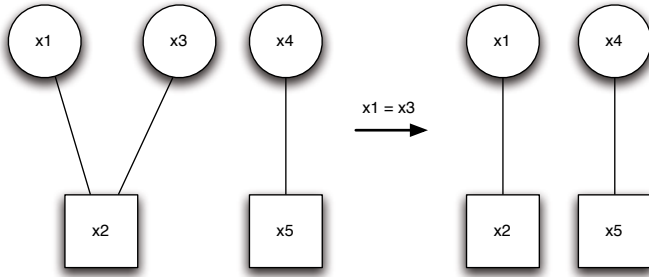


Fig. 8. Equality between $x1$ and $x3$ maps the sample on the left to its homomorphic image on the right. The resulting sample represents the set of instances in which each of the two tasks is mapped to exactly one device.

search the part of the design space that it has already visited. If a solution exists, $Simplify(s, soln)$ derives a set of equalities between variables in s from $soln$, and uses them to reduce s into a canonical representation of the equivalence class that contains $soln$ (line 17).

When no solution exists, $ExploreII$ attempts to learn the characteristics of the failed sample. The unsatisfiability of $s \wedge \neg C$ implies that any homomorphic images of s besides those in C cannot satisfy the design constraints. Hence, $ExploreII$ can safely reject any subsequent sample that is a homomorphic image of s , because every such image will either be unsatisfiable, or isomorphic to an element in $valid$. This knowledge can cause an exponentially large region of the design space to be avoided, but still allows random sampling over the good regions of the space.

The termination condition of $ExploreII$ follows readily from the incremental aspect of the learning approach. We consider a sample to be *most general* when it is equal to the generating set G . The most general sample can be homomorphically mapped into any of the equivalence classes in the design space. If this sample becomes unsatisfiable, then this implies no more solutions are left to be discovered in the design space. Hence, the algorithm can be terminated when the most general sample is added to the blocked list (lines 20-21).

Since each sample represents a larger number of equivalence classes than it did in the naïve algorithm, the solver has more opportunities to find a satisfying design instance. Combined with the learning of failed samples, $ExploreII$ attempts to overcome the difficulty of finding a satisfying instance in a highly constrained design space.

5 Evaluation

5.1 Experimental Setup

In this section, we evaluate the ability of the two algorithms to rapidly return solutions that are spread across the design space. We do not focus on the runtime

performance of the SMT solver, since this has been well-studied [10]. In order to visualize the entire design space, we use a small generator set to produce a particular φ . In addition, the channel capacities have been moved outside of the `Channel` constructor in order to further reduce the number of equivalence classes. Let us assume that the generator set has the following set of distinct terms:

$$\begin{aligned} |\text{Task}| &= 3 & |\text{Device}| &= 3 & |\text{Conflict}| &= 1 \\ |\text{Channel}| &= 2 & |\text{Binding}| &= 3 \end{aligned} \quad (5)$$

The projection used is $\Pi = \{\text{Binding}, \text{Channel}\}$, and thus there are a total of 10 variables (one variable per each constructor argument in each distinct term) that determine membership in a particular equivalence class. The theoretical maximum number of equivalence classes is the number of partitions of ten variables, i.e. 115,975. However, due to symmetries in the sets of records, the actual number of equivalence classes for this example is 11,233.

We ran the two exploration algorithms, *Explore* and *ExploreII*, and observed the outcome of each invocation to the SMT solver. For every experimental run, we bounded the maximum number of the invocations to 100. Figure 9 shows plots for the experimental runs. Each one of the six plots is a graphical representation of the design space. Parts that are colored in red represent portions of the design space that were explored by FORMULA but did not admit a satisfying instance; ones in green are samples from which the solver was able to find a satisfying instance. The plots (a)–(c) represent the design space for the platform mapping example from Section 2. The plots (d)–(f) represent a relaxed design space where a number of design constraints from the platform mapping problem have been removed ; since this design space is less constrained, these plots exhibit a larger number of green samples than the plots (a)–(c). The plots (a) and (d) show results after the naïve algorithm was run.

The plots (b), (c), (e), and (f) were produced with the improved algorithm, but with a varying amount of randomness in the sampling procedure. When *ExploreII* is performed without randomization in sampling, the task of searching the design space for a solution is handed off entirely to the SMT solver. As a result, in each invocation, the solver is guaranteed to return a solution, if any exists. On the other hand, with randomization, there is a probability that the picked sample emits no solutions at all. Therefore, in some invocations, the solver may fail to return a solution. This is a trade-off for achieving diversity in the exploration; randomization may lead to unsuccessful invocations of the solver, but can help avoid clustering of the solutions that tends to appear when no randomization is used. We describe this trade-off in more detail in Section 5.3.

Let us first introduce background notations that are necessary to explain these plots. Let $[M] = \{M' \mid M \sim M'\}$ be the equivalence class represented by model M , and let $\Lambda = \{[M_1], [M_2], \dots, [M_n]\}$ be the set of all equivalence classes, where the i^{th} class is represented by M_i . Then equivalence classes are partially ordered according to $[M_1] \leq [M_2] \Leftrightarrow M_1 \preceq_{\Pi} M_2$ (i.e. every member of the smaller equivalence class, $[M_1]$, is a homomorphic image of some member of the larger class, $[M_2]$). We plot the design space by dividing it into regions R_1, \dots, R_n such that: (1) For every $[M] \in R_i$ and $[M'] \in R_j$ it holds that

$[M] \not\leq [M']$ and $[M'] \not\leq [M]$, for $i \neq j$. (2) Within a region R_i , there exists a greatest equivalence class $[M]^\top: \forall [M'] \in R_i, [M'] \leq [M]^\top$. These regions occur naturally due to models with zero occurrences of some record types, and can be identified uniquely by their greatest class. In our experiment the regions are:

$$\begin{aligned} R_0 &= \emptyset, \\ R_1 &= \{Channel(x_1, x_2), Channel(x_3, x_4)\}, \\ R_3 &= \{Binding(y_1, y_2), Binding(y_3, y_4), Binding(y_5, y_6)\}, \\ R_4 &= R_1 \cup R_3 \end{aligned} \tag{6}$$

These regions are labeled in the figures using the short hand \emptyset , $\{f_0\}$, $\{f_1\}$, and $\{f_0, f_1\}$. The numbers at the top of each plot give the number of equivalence classes within regions.

Every equivalence class within a region is assigned a cell at some position along the y-axis. This position respects the \leq order on equivalence classes. Since the number of classes per region grows rapidly, we shrink the cell size and split the y-axis into a number of columns per region. This setup means the plots exhibit two important properties: (1) The number and internal complexity of record instances increases from left to right and bottom to top. (2) Record instances that are homomorphically similar are physically nearby, except when a column is broken and wrapped into the next column.

5.2 Randomization

Our exploration algorithm should behave well over various types of design spaces, and there are two important factors to take into account. First, under ideal circumstances, equivalence classes should be sampled uniformly across the design space. Second, sampling must be able to adapt to design spaces that are highly constrained and therefore, contain only a few valid solutions. These goals may be contradictory, in which case a reasonable balanced should be achievable.

The first goal is costly to achieve because it requires a canonical representation for all non-isomorphic homomorphic images of the symbolic set used to generate φ . This representation cannot be explicitly constructed, as it grows too quickly in size. Instead, an effective random sampling procedure must generate equivalence classes cheaply, but without introducing too much bias in the sampling process. Though the full statistical analysis of this problem is outside the scope of the paper, we describe the intuition behind our solution.

Given a set of variables X , it is easy to generate a random partition of the variables. The structure of the partitions of X closely follows the *integer partitions* of $|X|$. An integer partition of n is a collection of integers that sum to n . Generating the integer partitions of $|X|$ in lexicographic order is also straightforward. Each integer partition serves as a template for building a random partition of X . For example, if there are three variables $X = \{x, y, z\}$, then the partition $[2, 1]$ means pick two distinct variables to equate, and then pick one variable (which is equated to itself). The number of possible partitions of X that fit a template grows exponentially with respect to the template's lexicographic order in the integer partitions of $|X|$ and then decreases exponentially.

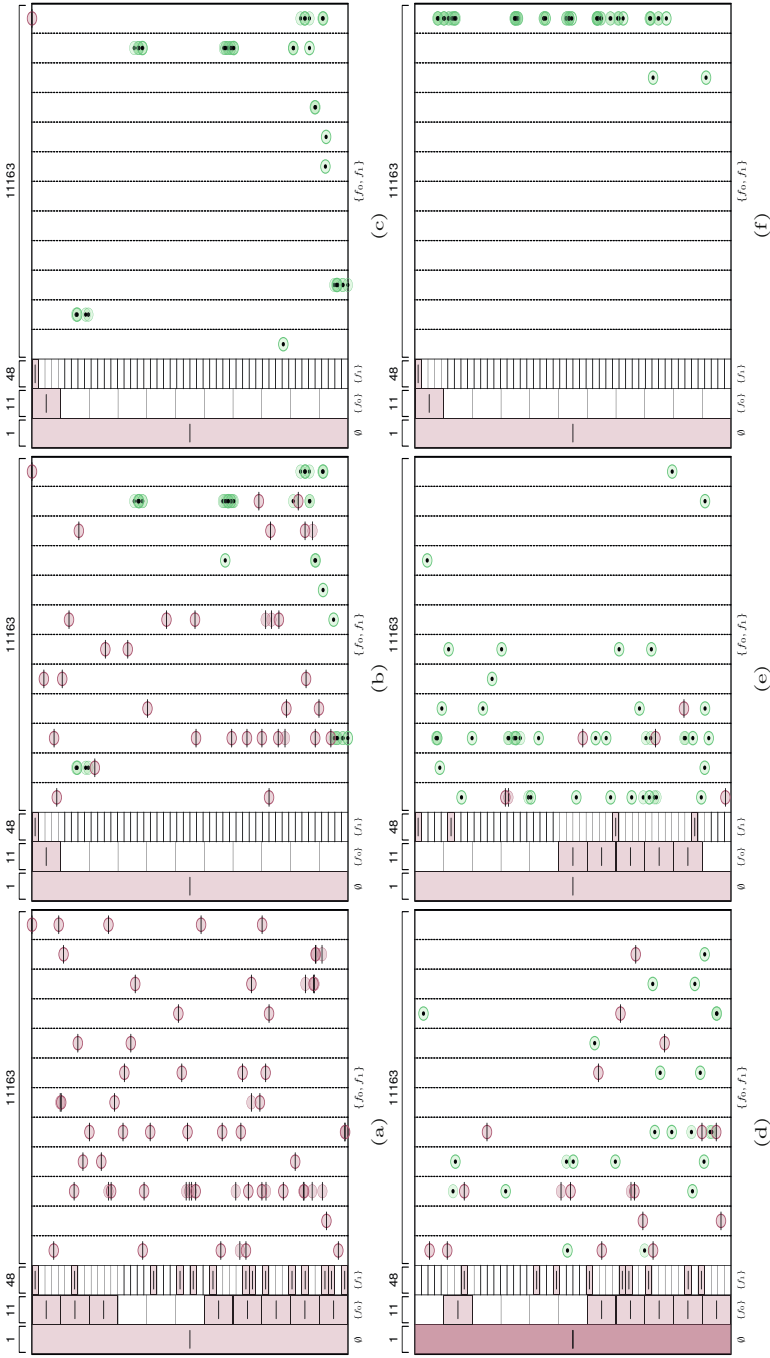


Fig. 9. Experiment results from running the exploration algorithms on the platform mapping problem. Each plot represents the design space; green dots mark explored samples that admitted a satisfying instance, and red dashes mark unsatisfiable samples. The plots (a)–(c) are results from the platform mapping example with all of the three design constraints described in Section 2; the plots (d)–(f) explore a relaxed design space. The plots (a) and (d) were generated using the naïve algorithm; the plots (b) and (e) using the improved algorithm (*ExploreII*) with $p_{gen} = 0.5$; the plots (c) and (f) using the improved algorithm with $p_{gen} = 1.0$.

We capture this behavior by fitting a normal distribution over the integer partitions of $|X|$. A random sample is constructed by first picking an integer partition as a template, and then randomly equating variables according to this template. The partitions are applied to the generator set to get a representative for an equivalence class. In our experiments, this approach removed an exponentially strong bias towards larger and more complex equivalence classes. The plots (a) and (d) in Figure 9 show the naïve random sampling algorithm with this correction applied. Our data suggests that this is a cost effective approach to sample design spaces with varying forms of symmetry in their generator sets.

5.3 Highly Constrained Design Spaces

Another difficulty arises when the design space is highly constrained. The plot (c) in Figure 9 shows all the solutions for the platform mapping under the generator set described in equation (5). In this case, there are only 41 non-isomorphic solutions out of 11,233, and these solutions are also highly clustered. By virtue of the solution set, no degree of random sampling can avoid the inherent clustering found here.

In order to address these cases, we add tuning probability p_{gen} to the *Sample-Class* method. The method selects the generating set G with probability p_{gen} as the sample to search or a random sample with probability $1 - p_{gen}$ (using the technique described in Section 5.2). Recall, from the discussion of *ExploreII*, that the generating set is the most general sample, and so when the solver is invoked on this sample, a either a new solution is returned or the sampling process terminates. The plot (c) was generated by setting $p_{gen} = 1$, thereby enumerating a new solution with every invocation of the solver. The plot (a) was generated by running the naïve random algorithm on the same problem. Though the distribution of points is fairly random, none of these points hit a satisfiable equivalence class. Finally, the plot (b) was generated by setting $p_{gen} = 0.5$. Here we see the algorithm alternating between randomized and solver-driven exploration patterns. Our initial results suggest that 0.5 provides a reasonable default trade-off between finding diverse solutions and quickly finding some solutions.

However, $p_{gen} = 1$ is not a reasonable solution to DSE. We relaxed the constraints on the motivating example by removing conflict constraints and the relational/functional constraints on **Channel** and **Binding**. Under this relaxation there are 4298 equivalence classes with solutions, and these classes are more evenly distributed about the space. The plot (f) shows sampling the relaxed space with $p_{gen} = 1$. In this case, the solutions are highly clustered due to the solver’s backtracking strategy. The plot (d) uses naïve random sampling and is even able to find some solutions. Finally, the plot (e) uses $p_{gen} = 0.5$ and finds a solution for almost every sample but does not exhibit the clustering behavior that is observed in the plot (f).

6 Related Work

6.1 DSE Frameworks

DESERT [26] is a framework that is closely related to FORMULA, with a goal of exploring design alternatives at the architectural level. Unlike FORMULA, design alternatives must be expressed as hierarchy of AND-OR choices with Boolean constraints describing interaction of design choices. DESERT encodes the design space and constraints symbolically, using BDDs [5]. However, the exploration in this framework is largely manual; the user specifies a constraint that should be true and DESERT prunes the design space accordingly. The user can also export points in the space to a modeling tool called the Generic Modeling Environment (GME).

CoBaSa [22] is a tool for automating the assembly of commercial off-the-shelf (COTS) components. It compiles system requirements and constraints among components into a pseudo-Boolean satisfiability (PBSAT) problem, which is tackled by a constraint solver. However, they focus on generating a solution that satisfies a large number of constraints, and do not focus on the exploration of the design space.

The technique developed by Hu et al. [14] collapses a multi-dimensional design space into a 3-dimensional space, after which the user selects portions of the space to explore with Cartesian co-ordinates. Their approach is similar to ours in that we both incorporate the user’s feedback into the exploration. Their goal is not to enumerate distinct designs, but to find ones that are optimal with a particular fitness metric.

Kakita et al. [18] developed an algebraic approach for DSE of dataflow systems. In this approach, each point in the design space is defined as a dataflow graph. Graph rewrite rules are applied to an initial graph iteratively to generate a set of alternative designs that preserve the scheduling constraints of the original design. The authors tackle the problem of the exponential growth in the design space by representing regularity in structures with compact recurrence relations. This approach has been implemented in the METROPOLIS framework [1].

There have been a great number of works that focus on the goal of finding a set of globally optimal solutions, many of which are surveyed in [13]. We believe that our exploratory approach is complementary to theirs. In an early phase of the design, where the overall architecture of the system has not been clearly defined, coming up with optimization functions is difficult. Hence, it is desirable for the designer to sample and experiment with a diverse set of alternatives. In addition, even if objective functions can be defined, there may be a large number of optimal solutions, which would then be examined and evaluated per-basis.

6.2 Exploration Techniques

Tabu search [12] is a technique in combinatorial optimization that uses a memory structure to keep track of solutions that it has already visited in the search space. Our approach is similar to Tabu search in that it also store points that it has

explored. In embedded system design, several groups [4,11,32] have evaluated Tabu search with respect to its effectiveness in finding globally optimal solutions. As far as we are aware of, Tabu search has not been used to exhaustively explore the design space.

Planning in AI solves the problem of finding a path between a pair of source and destination points on a search space. There is a wealth of literature on algorithms and heuristics to prune parts of the space that need not be explored. Some of these techniques, such as simulated annealing [31] and rapidly-exploring random trees [20], use random sampling to increase diversity during the exploration. Our random sampling approach was inspired by them.

Partitioning a large search space into equivalence classes has been employed in other areas of software engineering. In testing, the space of the test input can be partitioned based on a certain notion of equivalence, and only a single test case from each class needs to be executed [25]. In model checking [6], *state abstraction* can be used to group related states together, thereby reducing the size of the space that the model checker needs to visit. As far as we are aware of, our work is the first one to apply the partitioning method to explore distinct solutions in a design space.

In software product lines, researchers have studied techniques to explore and analyze the space of product configurations based on feature models [2,3]. These works focus on managing variability in features, whereas constraints in FORMULA describe non-functional, architectural properties such as scheduling and security requirements. Constraint solvers use symmetry breaking predicates to avoid searching through solutions that are isomorphic to each other [8]. These predicates operate on low-level representations such as Boolean propositions, and do not take into account high-level domain knowledge.

6.3 DSL Specification Languages

A number of tools exist for specifying DSLs, with various degrees of automated analysis. We have already mentioned the work of DESERT [26]. Additionally, the Atlas Model Management Architecture (AMMA) [29] uses the OMG's meta-object facility (MOF) [27] as the DSL specification language. Abstract state machines (ASMs) are used to define the behavioral semantics of DSLs. These tools are built on top of the Eclipse Modeling Framework (EMF).

The KerMeta [24] framework provides a MOF compliant specification language. At the time of writing, KerMeta provides static type analysis and runtime checking of pre/post conditions. Additional formal methods are provided by exporting to other tools. The KerMeta language is inspired by the programming language Eiffel and is object-oriented in nature. It provides its own imperative language for specifying DSL behavioral semantics. The AToM³ [9] framework is integrated with the Maude theorem prover [7]. AToM³ focuses on behavioral and transformational DSL semantics. It uses the term rewriting formalism of Maude to evaluate LTL queries on models.

7 Discussion and Conclusion

In conclusion, we have presented an approach for exploring the design space of complex systems in our framework, FORMULA. Our framework combines results from domain-specific languages, symbolic execution, and automated theorem proving in order to quickly move from a specification of a design space to a set of distinctive solutions. Our exploration algorithms go further than finding non-isomorphic solutions; they attempt to quickly visit a diverse set of solutions across the design space. We have presented initial data to show the efficacy of our approach.

In the improved exploration algorithm (*ExploreII*), the predicate $\neg C$ can be considered to be a *complete* symmetry breaking predicate [8]—complete because it *guarantees* that the solver will not find any instance that is isomorphic to the ones that have already been discovered. But computing a full symmetry breaking predicate is generally expensive. In our case, the computation of a homomorphism from a sample to the ones in *valid* can be costly; no polynomial algorithms are known for this problem. An alternative approach is to use a *partial* symmetry breaking predicate, which provides a weaker guarantee but is much cheaper to compute [8,23,30]. We are currently investigating a way to integrate this partial approach into our framework.

We are also interested in studying other mechanisms to differentiate solutions in design spaces. In case studies for embedded systems, we encountered scenarios where a partial order over records would also be useful mechanism to distinguish solutions. This also raises interesting theoretical and practical questions on how to combine various differentiation mechanisms, and how to fairly sample and efficiently encode more general equivalence classes into an SMT solver.

The opportunities to combine DSE with rapid prototyping and optimization appear promising. The DSL approach in general, and FORMULA in particular, allow behavioral semantics to be assigned to DSLs. Thus, it is possible to automatically simulate and profile designs as they are sampled. Either the user or a utility function could rank solutions, and then these results could be used to further prune the design space. In this instance, pruning would mean adding more constraints to the SMT formula φ to refine the design space.

References

1. Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A.L.: Metropolis: An integrated electronic system design environment. *IEEE Computer* 36(4), 45–52 (2003)
2. Batory, D.: Feature models, grammars, and propositional formulas. In: Obbink, H., Pohl, K. (eds.) *SPLC 2005*. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
3. Benavides, D., Martín-Arroyo, P.T., Cortés, A.R.: Automated reasoning on feature models. In: Pastor, Ó., Falcão e Cunha, J. (eds.) *CAiSE 2005*. LNCS, vol. 3520, pp. 491–503. Springer, Heidelberg (2005)
4. Bland, J.A., Dawson, G.P.: Tabu search and design optimization. *Computer-Aided Design* 23(3), 195–201 (1991)

5. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers* 35(8), 677–691 (1986)
6. Clarke, E.M., Grumberg, O., Peled, D.: *Model Checking*. MIT Press, Cambridge (1999)
7. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Bevilacqua, V., Talcott, C.L. (eds.): *All About Maude - A High-Performance Logical Framework*. LNCS, vol. 4350. Springer, Heidelberg (2007)
8. Crawford, J.M., Ginsberg, M.L., Luks, E.M., Roy, A.: Symmetry-breaking predicates for search problems. In: *Principles of Knowledge Representation and Reasoning*, pp. 148–159 (1996)
9. de Lara, J., Vangheluwe, H.: Atom³: A tool for multi-formalism and meta-modelling. In: Kutsche, R.-D., Weber, H. (eds.) *FASE 2002*. LNCS, vol. 2306, pp. 174–188. Springer, Heidelberg (2002)
10. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) *TACAS 2008*. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
11. Eles, P., Peng, Z., Kuchcinski, K., Daboli, A.: System level hardware/software partitioning based on simulated annealing and tabu search. In: *Design Automation for Embedded Systems*, vol. 2, pp. 5–32. Kluwer Academic Publishers, Dordrecht (1997)
12. Glover, F., Laguna, M.: *Tabu Search*. Kluwer Academic Publishers, Dordrecht (1998)
13. Gries, M.: Methods for evaluating and covering the design space during early design development. *Integration* 38(2), 131–183 (2004)
14. Hu, X., Greenwood, G.W., Ravichandran, S., Quan, G.: A framework for user assisted design space exploration. In: *DAC*, pp. 414–419 (1999)
15. Jackson, E.K., Kang, E., Dahlweid, M., Santen, D.S.T.: Components, platforms and possibilities: Towards generic automation for mda. In: *Embedded Software* (2010)
16. Jackson, E.K., Seifert, D., Dahlweid, M., Santen, T., Bjørner, N., Schulte, W.: Specifying and composing non-functional requirements in model-based development. In: *Software Composition*, pp. 72–89 (2009)
17. Jackson, E.K., Sztipanovits, J.: Formalizing the structural semantics of domain-specific modeling languages. In: *Software and Systems Modeling* (2008)
18. Kakita, S., Watanabe, Y., Densmore, D., Davare, A., Sangiovanni-Vincentelli, A.L.: Functional model exploration for multimedia applications via algebraic operators. In: *ACSD*, pp. 229–238 (2006)
19. Kanajan, S., Zeng, H., Pinello, C., Sangiovanni-Vincentelli, A.L.: Exploring trade-off's between centralized versus decentralized automotive architectures using a virtual integration environment. In: *DATE*, pp. 548–553 (2006)
20. LaValle, S.M., Kuffner Jr., J.J.: Randomized kinodynamic planning. I. *J. Robot Res.* 20(5), 378–400 (2001)
21. Mandel, L., Cengarle, M.V.: On the expressive power of OCL. In: Woodcock, J.C.P., Davies, J. (eds.) *FM 1999*. LNCS, vol. 1708, pp. 854–874. Springer, Heidelberg (1999)
22. Manolios, P., Vroon, D., Subramanian, G.: Automating component-based system assembly. In: *ISSTA*, pp. 61–72 (2007)
23. McDonald, I., Smith, B.M.: Partial symmetry breaking. In: Van Hentenryck, P. (ed.) *CP 2002*. LNCS, vol. 2470, pp. 431–445. Springer, Heidelberg (2002)

24. Muller, P.-A., Fleurey, F., Jézéquel, J.-M.: Weaving Executability into Object-Oriented Meta-languages. In: 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS), pp. 264–278 (2005)
25. Myer, G.J.: *The Art of Software Testing*. Wiley, Chichester (2004)
26. Neema, S., Sztipanovits, J., Karsai, G., Butts, K.: Constraint-based design-space exploration and model synthesis. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 290–305. Springer, Heidelberg (2003)
27. Object Management Group. *Meta Object Facility (MOF) Core Specification Version 2.0* (2006)
28. Polya, G., Read, R.C.: *Combinatorial Enumeration of Groups, Graphs, and Chemical Compounds*. Springer, New York (1987)
29. Ruscio, D.D., Jouault, F., Kurtev, I., Bézivin, J., Pierantonio, A.: Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs. RR 06.02 (April 2006)
30. Shlyakhter, I.: Generating effective symmetry-breaking predicates for search problems. *Discrete Applied Mathematics* 155(12), 1539–1548 (2007)
31. van Laarhoven, P.J.: *Simulated Annealing: Theory and Applications*. Springer, Heidelberg (1987)
32. Wiangtong, T., Cheung, P.Y.K., Luk, W.: Comparing three heuristic search methods for functional partitioning in hardware-software codesign. In: *Design Automation for Embedded Systems*, vol. 6, pp. 425–449. Kluwer Academic Publishers, Dordrecht (2002)

Migration of Legacy Software Towards Correct-by-Construction Timing Behavior

Stefan Resmerita¹, Kenneth Butts², Patricia Derler¹, Andreas Naderlinger¹,
and Wolfgang Pree¹

¹ C. Doppler Laboratory Embedded Software Systems, Univ. Salzburg
Jakob-Haringer-Str. 2, 5020 Salzburg, Austria
firstname.lastname@cs.uni-salzburg.at

² Toyota Technical Center
1555 Woodridge, Ann Arbor, Michigan 48105
ken.butts@tema.toyota.com

Abstract. This paper presents an approach for incrementally adjusting the timing behavior of legacy real-time software according to explicit timing specifications expressed in the Timing Definition Language (TDL). The main goals of such a migration are ensuring predictability of the timing behavior, and enabling adaptivity of the system. The latter is particularly important for embedded control systems which adapt their computational load in accordance to parameters of the physical environment in which they operate.

Our approach entails a minimal instrumentation of the original code combined with an automatically generated runtime system, which ensures that the executions of designated periodic computations in the legacy software satisfy the logical execution time specifications of the TDL model. The presented approach has been applied to a complex legacy controller system in the automotive domain.

1 Introduction

Various approaches have been recently proposed to design and implement adaptive embedded systems. For real-time embedded software, it has been argued e.g. in [1] that so-called semantics-preserving execution environments stemming from Giotto [2] and from synchronous programming [3] can prove to be a key ingredient for adaptivity. These approaches are intended to be used in modern methodologies for embedded system design such as Model-Driven Engineering (MDE) [4] and Platform-Based Design [5], which advocate a top-down approach for application development. The development process starts from high level models, which are incrementally refined to software models and then to implementations on execution platforms.

While the benefits of these approaches are well-understood, their full adoption in the established embedded industry is rather slow. One of the main factors responsible for this is the large base of legacy applications, which have been traditionally developed at the programming language level, are usually highly

optimized and thoroughly tested. MDE is thus employed only partially, typically for developing new functionality up to the software model, which is then manually merged with the existing legacy code.

Several timing specification languages and tools such as the Hierarchical Timing Language (HTL) [6] and the Timing Definition Language (TDL) [7] follow the Giotto model, providing semantics-preserving execution environments based on the Logical Execution Time (LET) abstraction. They serve the general argument that execution time of software should be captured in high-level models [8]. HTL was employed to describe adaptive real-time systems in [1]. While LET-based programming disciplines assume the classical MDE top-down approach, they are particularly amenable to a bottom-up application to legacy software, due to the separation of concerns provided by the LET paradigm, where timing is separated from functionality. This facilitates the enforcement of timing requirements on legacy software in a systematic and minimally interventive way. It also addresses intellectual property concerns, requiring no information about what the legacy code does. However, availability of the legacy source code and platform configuration information is assumed.

In this paper we describe how to apply TDL modeling to typical legacy controller systems. We propose an instrumentation-based approach, with minimal intervention in the legacy code and platform configuration. To achieve this, we had to reconcile the top-down approach of TDL with the constraints imposed by the legacy system. Two main aspects required trade-offs in this respect: (1) Event-triggered computations, which in TDL are assumed to have lower priorities than time-triggered tasks, while the legacy application has higher priority events, and (2) the TDL runtime system, which originally implements a virtual machine called E-Machine and compiles the timing specification into code for this E-Machine, called E-code, which has proved to be quite large for complex legacy applications. Issue (1) was addressed by a careful scheduling analysis, considering information about minimum inter-arrival times of high-priority events. Problem (2) was resolved by employing an application-specific runtime system, called TDL-Machine, which was code-generated from the TDL model and from application-specific information.

The approach described in this paper was applied to a complex industrial legacy application in an incremental manner. The desired timing behavior was tested by software-in-the-loop and hardware-in-the-loop simulations.

2 Background

This section briefly presents the Timing Definition Language (TDL), as well as common characteristics of legacy software that challenge some of the assumptions made in LET-based programming disciplines such as TDL.

2.1 The Timing Definition Language (TDL)

TDL allows the LET-based specification of timing properties of hard real-time applications. The LET of a computational unit, or task, represents a fixed logical

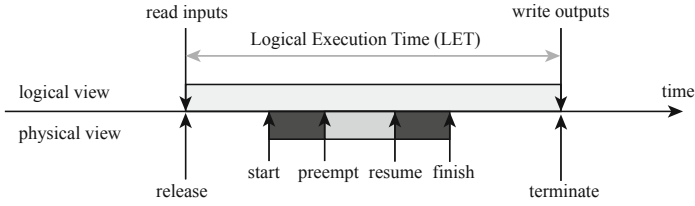


Fig. 1. The Logical Execution Time (LET)

duration between the time instant when the task becomes ready for execution and the instant when the execution finishes. A task's LET is specified at the model level, independently of the task's functionality. When deploying the model on a platform, the LET specification is satisfied if the total physical execution time of the task is within the LET interval for every task invocation, and an appropriate runtime system ensures that task inputs are read at the beginning of the LET interval (the release time) and task outputs are made available at the end of the LET interval (the termination time). This is illustrated in Figure 1. Between release and termination points, the output values are those calculated in the previous execution. Default or specified initial values are used in the first execution of a task.

Tasks can receive information from the environment via sensors and act on the environment via actuators. A task has input ports, output ports, and state ports. State ports keep state information between different executions of the same task.

TDL is mainly used for specification of periodic task executions under the LET model. Timing specifications are declared in TDL by means of the *mode* construct. A mode represents a set of concurrent activities: task invocations, actuator updates, and mode switches, which are periodically executed. A mode activity has a specified execution rate and may be carried out conditionally. The LET of a task is expressed as the mode period divided by the frequency of the task invocation. The schedule of activities within a mode period is determined statically and it is carried out at runtime by a dedicated component called the *TDL-Machine*. This performs the following operations at every time step of the schedule:

- Update output ports of tasks whose LET end at the current time step. At time 0, the ports are initialized rather than updated.
- Update actuators.
- Test for mode switches. If a mode switch is enabled, switch to the target mode.
- Update input ports of the tasks whose LET start at the current time step.
- Trigger the execution of the tasks whose LET start at the current time step.

```

1  module Sender {
2      sensor int s1 uses getS1;
3      actuator int a1 uses setA1;
4      public task inc {
5          input int i;
6          output int o := 10;
7          uses incImpl(i,o);
8      }
9      start mode main [period=5ms] {
10         task [freq=1] inc(s1); //LET = 5ms (=period/freq)
11         actuator [freq=1] a1 := inc.o;
12         mode [freq=1] if exitMain(s1) then freeze;
13     }
14     mode freeze [period=1000ms] {}
15 }

```

Listing 1.1. A TDL example

TDL allows also declarations of sporadic activities related to executions of event-triggered tasks. Such tasks are declared *asynchronous* in TDL, meaning that no timing constraints are specified for them. TDL can be used to express only data dependencies related to asynchronous tasks.

TDL provides a top level structuring unit called a module, which groups sensors, actuators, tasks, and modes that belong together. The module concept serves multiple purposes: (1) a module provides a name space and an export/import mechanism and thereby supports decomposition of large systems, (2) modules allow the parallel composition of real-time applications, (3) modules serve as units of loading, that is, a runtime system may support dynamic loading and unloading of modules, and (4) modules are the natural choice as unit of distribution because dataflow within a module (cohesion) will most probably be much larger than dataflow across module boundaries (adhesion).

An example of a TDL program is shown in Listing 1.1 and a graphical representation of this program is shown in Figure 2. In the example, module Sender contains a sensor variable *s1* and an actuator variable *a1*. The value of *s1* is updated by executing the (platform specific) driver *getS1* and the value of *a1* is sent to the physical actuator by using the platform-specific driver *setA1*. Every module has exactly one start mode, indicated by preceding the mode declaration with the keyword *start*. The declaration of the output port of task *inc* specifies an initial value of 10. The task is invoked in mode *main* of the Sender module, where its input port is connected to the sensor *s1*. In the same mode, actuator *a1* is updated with the value of the task's output port. The timing behavior of the mode activities is specified by means of individual frequencies within their common mode period. For example, with a frequency of 1, task *inc* is defined to have a LET of 5 ms. A more detailed description of TDL features can be found in [9].

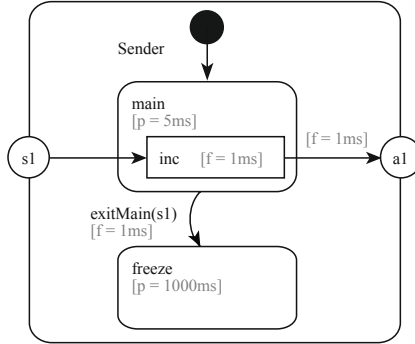


Fig. 2. Representation of the TDL example from Listing [1.1](#)

A TDL application consists of a set of time-triggered tasks and a runtime system called TDL-Machine, which performs all mode operations according to the TDL specifications. A platform specific implementation of the TDL-Machine can be generated from the specifications [\[10\]](#).

2.2 Aspects of Legacy Controller Systems

It is common for an embedded controller software system to contain both time-triggered and event-triggered computations. Some event-triggered tasks may require fast reaction times, and thus may have higher priorities than time-triggered tasks.

Legacy operating systems require tasks to be split into execution units, also called tasks. We refer to such a task as a *platform task*. Typically the number of tasks is restricted. For example, OSEK/VDX [\[11\]](#) or AUTOSAR OS [\[12\]](#) suggest a maximum of 8 to 16 tasks. Complex systems often comprise more tasks thus a common design practice is to group the time-triggered computations into a small number of time-triggered platform tasks, which are triggered by a high priority task (also called sequencer or dispatcher task [\[13\]](#)) that is itself triggered from a periodic interrupt that defines the base period in the system. That high priority task dispatches the time-triggered platform tasks at multiples of the base period, using system services for task activation. In addition, each time-triggered task may internally perform computations at multiples of the task's period.

Another common characteristic of legacy embedded code is heavy usage of shared memory communication (global variables) between various components in the system. Moreover, communication with the physical environment is done by memory-mapped I/O devices. Thus, reading from a sensor means accessing a (read-only) global variable, while actuating means writing into a global variable.

3 Modeling Timing Behavior of Legacy Controllers with TDL

The main goal of imposing a LET-based execution and data-transfer semantics on an existing application is to eliminate unpredictable behaviors due to variations in execution times. An important refactoring requirement in this respect is minimal modification of the legacy system, including the application code and its configuration on the platform. Thus, code changes are done only by adding the TDL-Machine as a separate component and by inserting calls to the TDL-Machine functions at well defined, top-level places in the original application. No line of the legacy code is modified. Also, all the parameters of the legacy configuration remain unchanged (same platform tasks, periods, and priorities). Additional resources of the operating system may be necessary to trigger executions of TDL tasks, as described in the sequel. Moreover, an additional platform task may be required for the TDL-Machine.

Modeling the timing behavior of legacy software with TDL must reconcile the assumptions made on the implementation of TDL tasks and the ability of the runtime system to control executions with the characteristics of the legacy applications mentioned in Section 2.2. TDL requires that inputs and outputs of TDL tasks be passed to the implementation functions by means of function arguments, while the legacy code uses mostly global variables. Platform tasks are activated according to the legacy configuration, which must remain unchanged, so the TDL runtime system does not have full control over triggering of TDL tasks. Nevertheless, one has to make sure that the TDL semantics is preserved.

Complex legacy applications contain periodic computations with periods that differ by several orders of magnitude. For example, a computation may have a period of 5 milliseconds, while another one may have a period of 3 seconds. Since each periodic computation is mapped to a TDL task, the number of operations of the TDL-Machine in a hyperperiod of the system (the least common multiple of all the periods) may be quite large. This makes the usage of the E-Machine approach originally proposed in Giotto [2] and later used in [14] and [15] impractical due to memory constraints, since the E-code defines all operations in a hyperperiod. Thus, we chose to generate directly from the TDL specification a TDL-Machine specific to the particular legacy system, rather than generating E-code and using a generic implementation of an E-Machine.

3.1 Mapping the Legacy Architecture to TDL Constructs

A TDL task can be mapped to any function of the legacy code, which is referred to as the implementation function of the task. Since TDL tasks are assumed to be independent, a TDL task cannot be mapped to a function that is called from the implementation function of another TDL task. Thus, different TDL tasks may have only data dependencies, not execution control dependencies. This assumption works for common legacy systems where multiple controllers are run on the same execution platform by having different control functions called from the same legacy platform task function. This is also naturally applicable to implementations of model-based designs employing data-flow models.

In general, TDL tasks are included in platform tasks, in the sense that a platform task may contain implementation functions of more than one TDL task, while a TDL task cannot be mapped to more than one platform task. If a TDL task is mapped to a platform task, the platform task function is the implementation function of the TDL task. A TDL task can be:

- Synchronous, also called time-triggered, if it corresponds to a periodic computation and it has a LET specification. The period of a synchronous TDL task is the same as the period of execution of the implementation function of the TDL task, which is a multiple of the period of execution of the platform task that contains the implementation function. The LET intervals are established together with application engineers, such that the system is schedulable.
- Asynchronous, also called event-triggered, if it corresponds to an event-triggered computation, in which case it has no LET. The task implementation function of an asynchronous TDL task always corresponds to a platform task function. In this case, TDL specifies only the data dependencies between an asynchronous task and the other TDL tasks.

An input (output) port of a TDL task T corresponds to a legacy global variable that is read (written) during the execution of the task implementation function and that is written (read) in another part of the legacy application that is independent of the particular TDL task T .

We consider the typical case of memory-mapped I/O devices, where sensors and actuator values are stored in memory locations (global variables mapped to hardware registers). Thus, sensing is performed by first writing in an output variable (which, for example, can be mapped to a command register of an A/D converter) and then reading from an input variable (which, for example, can be mapped to the data register of the A/D converter). Consequently, the TDL model contains no dedicated sensor/actuator variables.

Since we deal here with the migration of monolithic legacy controllers, we define one TDL module per application. A TDL module may contain several modes. We consider here the case where all TDL tasks are present in each mode, such that modes only define different timing behaviors of the tasks.

3.2 Implementation of the TDL Operational Semantics

TDL operations are carried out at runtime by a dedicated component called the TDL-Machine, which deals with activation of synchronous TDL tasks, data transfer, and mode switches. The architecture of the TDL-Machine and its interaction with the legacy application are schematically illustrated in Figure 3. The TDL-Machine has a time-triggered component and an event-triggered component.

The time-triggered component is executed in a periodic platform task with the highest priority and smallest period (the base period). Such a task is common in legacy applications, its main role being to dispatch executions of lower-priority periodic tasks with periods that are multiples of the base period. If the task is not defined in the legacy application, or if the TDL-Machine needs a smaller

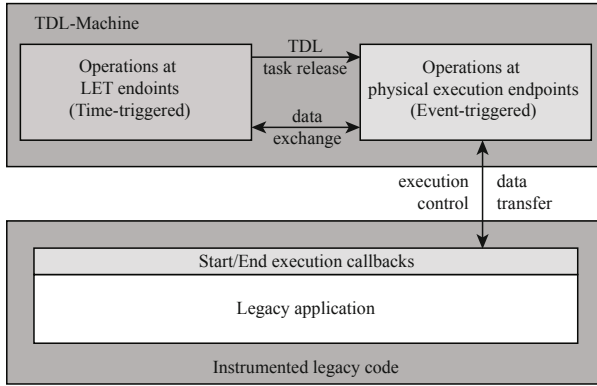


Fig. 3. The TDL-Machine architecture

```

1 void Tx_Implementation() {
2     ... // local variables
3     On_Tx_start_execution(); // execution start callback
4     ... // legacy code
5     On_Tx_end_execution(); // execution end callback
6 }

```

Listing 1.2. LET-based operational semantics

base period (e.g., for a finer granularity of LET endpoints), then an additional platform task needs to be introduced. The time-triggered component performs all the operations that are necessary at LET endpoints. The operations that interfere with the execution of legacy code are synchronous task invocations and data transfers. We describe how to deal with these situations in the sections below. Mode switches are implemented by simply changing the LETs for the task set. These LET intervals for each mode are stored in a table.

The event-triggered component defines one start function and one end function for each TDL task. These functions are called whenever an execution of the task implementation (legacy) function begins, respectively ends. Thus, calls to these functions are inserted at the beginning and end of the corresponding legacy functions. Their role is to perform buffering and synchronization operations, as detailed further in this section.

Listing [1.2](#) sketches the implementation of a LET-based task.

3.3 Activation of Synchronous TDL Tasks

In every execution period, a time-triggered TDL task must be activated at the start of its LET interval. The execution period is a multiple of the period of the platform task that contains the TDL task implementation function. For example, consider a platform task with a period of 5ms, with task function

```

1 void Platform_Task() {
2     taskCounter = taskCounter + 1;
3     legacy_func_5ms();           // executed every 5ms
4     if (taskCounter & 0x01) {
5         legacy_func_10ms();     // executed every 10ms
6     }
7 }

```

Listing 1.3. Platform task

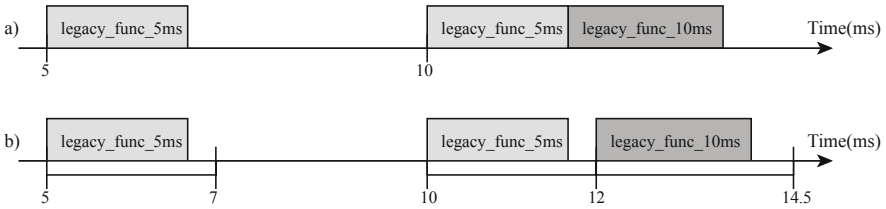


Fig. 4. Execution example: original (a) and with LETs (b)

given in Listing 1.3. One can define two TDL tasks, corresponding to the two periodically executed functions: a TDL task T5 with a period of 5ms and implementation function `legacy_func_5ms`, and a TDL task T10 with a period of 10ms and implementation function `legacy_func_10ms` (note that the period of T10 is twice the period of the platform task containing T10's implementation function). This will be used as a running example for the current subsection.

If the start of the LET interval coincides with the start of the period for a TDL task, then no special activation action is taken by the TDL-Machine for that task, since every platform task is activated by the platform anyway. However, the structure of the legacy code may require that the LET begins at a fixed offset after the start of the TDL task's period. TDL has been extended to deal with such cases, as described next with the help of our example.

The LET of T5 starts at the beginning of every 5ms period; assume it has a value of 2ms. In order to preserve the order of execution of the two legacy functions at the 10ms time points, the LET of T10 must succeed the LET of T5 in each 10ms period. In other words, the LET of T10 must have an offset of at least 2ms in the 10ms period. Figure 4(b) shows an example of task executions where T10 has an offset of 2ms and a LET of 2.5ms.

TDL has been extended to allow the definition of a LET offset - a time interval between the beginning of the period and the beginning of the LET. This feature is offered in TDL as a *slot selection*: the mode period p is divided into f slots and the LET of any individual task invocation is defined more explicitly as an interval that starts and ends at integer multiples of p/f . Thus, a task's LET

```

1  start mode main [period=10ms] {
2  //T5: period=5ms, offset=0, LET=2ms
3      task [freq=10, slots=1-2,5-6] T5();
4
5  //T10: period=10ms, offset=2ms, LET=2.5ms
6      task [freq=20, slots=5-9] T10();
7  }

```

Listing 1.4. TDL mode for Figure 4(b)

corresponds to a *slot group*. The slots are numbered from 1 to f . TDL offers a compact syntax for specifying a task's slot groups within a mode period, as follows. A repeating pattern of slot groups is specified by using the character "*" after the pattern. A slot group can be optional, which means that the corresponding task execution may be skipped at runtime, if this helps in finding a feasible schedule. Some examples are:

slots=1* : all slots are mandatory and $LET=p/f$; this is the default.
 slots= \sim 1|2* : $LET=p/f$, the first slot is optional and the remaining slots are mandatory.
 slots=1-3* : mandatory slot groups with $LET=3*p/f$ each.

The TDL code corresponding to the example in Figure 4(b) is given in Listing 1.4.

In order to enable the TDL-Machine to trigger executions of offset TDL tasks, additional synchronization points are introduced. The implementation of this synchronization is operating system-specific. For example, in the case of an OSEK operating system, a WaitEvent system call is used in the entry instrumentation function. At runtime, a corresponding SetEvent system call is performed by the TDL-Machine at the LET start. This ensures that the legacy function does not start executing before the LET start. No change is made to the platform-triggered activation of platform tasks (which include the asynchronous TDL tasks).

To enforce the specification given in Listing 1.4 for task T10, the task's execution start callback `On_T_10ms_start_execution` is defined to make a blocking call on an operating system resource. The time-triggered component of the TDL-Machine releases the resource at the LET start of T10. This process is described in Listing 1.5, where the resource is an OSEK event.

3.4 Data Transfer Operations

The implementation of LET-based data transfer for a synchronous TDL task must deal with the fact that data communication between the legacy implementation function of the TDL task and the rest of the system is done by shared memory. Since inputs and outputs are provided via global variables, their values need to be buffered in TDL-specific variables as described below. The following behavior must be ensured:

```

1 // called at the beginning of legacy_func_10ms
2 void On_T10ms_start_execution() {
3     WaitEvent(EV_T10_LET_START);
4 }
5 // time-triggered LET scheduler function
6 void LETSchedulerStep(double time) {
7     ...
8     if (time == LET_START_T10ms){
9         SetEvent(Platform_Task, EV_T10_LET_START);
10    }
11    ...
12 }

```

Listing 1.5. Implementation of the TDL program in Listing 1.4 for task T10

- (A) When the LET begins, the value of each original input variable is stored in an additional internal task input variable. This is necessary because the value of an original input variable may change between the starting of the LET and the moment when the physical execution of the TDL task implementation function starts.
- (B) During execution, the task implementation function uses the values of the internal input variables instead of the actual values of the original variables.
- (C) During execution, the values of legacy output variables are stored in additional internal output
- (D) When LET ends, the original global output variables (the legacy variables) are updated with the values of the TDL-internal output variables.

Operations A and D are executed by the time-triggered part of the TDL-Machine. To achieve a minimal instrumentation of the legacy code, we chose to implement B and C in the execution callbacks, as explained below.

Communication Between Synchronous TDL Tasks Only. Consider two periodic time-triggered legacy functions `tt_func_write` and `tt_func_read`, where some execution of `tt_func_write` updates a global variable `gvar`, and

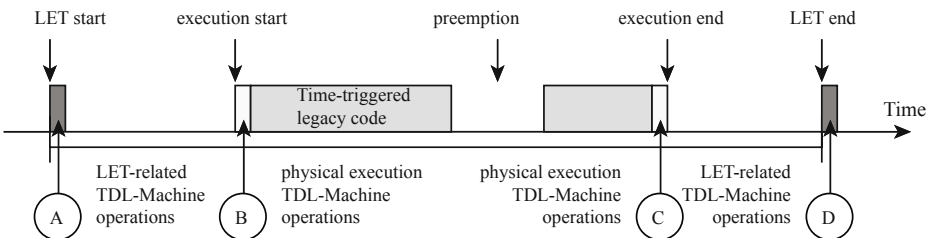


Fig. 5. Data transfer operations for LET-based tasks

```

1  module Example {
2      public task T_WRITE {
3          output int gvar := 0;
4          uses T_WRITE_Implementation(gvar);
5      }
6      public task T_READ {
7          input int gvar;
8          uses T_READ_Implementation(gvar);
9      }
10     start mode main [period=5ms] {
11         task [freq=1] T_WRITE();    //LET = 5ms, offset = 0
12         ms
13         task [freq=5, slots=2-4]
14             T_READ(T_WRITE.o); //LET = 3ms, offset = 1
15         ms
16     }
17 }

```

Listing 1.6. TDL module with 2 tasks

```

1  void TDLMachineStep(time) {
2      if (time == LET_START_T_READ) {                // A
3          T_READ_in_gvar = T_WRITE_o_gvar;
4      }
5      if (time == LET_END_T_WRITE) {                // D
6          T_WRITE_o_gvar = T_WRITE_tp_o_gvar;
7      }
8  }

```

Listing 1.7. TDL-Machine step

```

1  void On_T_WRITE_start_execution() {                // B
2      T_WRITE_tp_gvar = gvar;
3  }
4  void On_T_WRITE_end_execution() {                 // C
5      T_WRITE_tp_o_gvar = gvar;
6      gvar = T_WRITE_tp_gvar;
7  }
8  void On_T_READ_start_execution() {                // B
9      T_READ_tp_gvar = gvar;
10     gvar = T_READ_in_gvar;
11 }
12 void On_T_READ_end_execution() {                 // C
13     gvar = T_READ_tp_gvar;
14 }

```

Listing 1.8. Operations at physical endpoints

```

1  module Example {
2      public task T_RW {
3          input int gvar_r;
4          output int gvar_w := 0;
5          uses T_RW_Implementation(gvar_r, gvar_w);
6      }
7      public task E_WRITE {
8          output int gvar_r :=0;
9          uses E_WRITE_Implementation(gvar_r);
10     }
11     public task E_READ {
12         input int gvar_r;
13         input int gvar_w;
14         uses E_READ_Implementation(gvar_r, gvar_w);
15     }
16     start mode main [period=5ms] {
17         task [freq=1] T_RW(E_WRITE.gvar_r);
18     }
19     asynchronous {
20         E_WRITE();
21         E_READ(E_WRITE.gvar_r, T_RW.gvar_w);
22     }
23 }

```

Listing 1.9. TDL program

some execution of `tt_func_read` reads from the same variable. Assume now that `tt_func_write` is mapped to a TDL task `T_WRITE` and `tt_func_read` is mapped to another TDL task `T_READ`. A sample module with these two TDL tasks is described in Listing [L.6](#).

To ensure LET-based data transfer between the functions `tt_func_write` and `tt_func_read`, the TDL-Machine is generated so that it has an internal output variable for `T_WRITE` called `T_WRITE_tp_o_gvar`, a task output port variable called `T_WRITE_o_gvar`, and an input port variable for `T_READ`, called `T_READ_in_gvar`. The TDL-Machine also uses additional buffer variables `T_WRITE_tp_gvar` and `T_READ_tp_gvar`. The following data transfer callbacks are defined in the TDL-Machine:

- `On_T_WRITE_start_execution` and `On_T_WRITE_end_execution`,
- `On_T_READ_start_execution` and `On_T_READ_end_execution`.

Figure [5](#) shows a sample execution trace which highlights when data transfer operations of the TDL-Machine are executed. The operations at LET endpoints A and D are described in Listing [L.7](#) and operations at physical execution endpoints B and C are described in Listing [L.8](#).

Communication Between Synchronous and Asynchronous TDL Tasks. Consider three legacy functions `tt_read_write`, `ev_write` and `ev_read`, with

```

1  void TDLMachineStep(time) {
2      if (time == LET_START_T_RW) { // A
3          T_RW_in_gvar_r = E_WRITE_o_gvar_r;
4      }
5      if (time == LET_END_T_RW) { // D
6          T_RW_o_gvar_w = T_RW_tp_o_gvar_w;
7      }
8  }

```

Listing 1.10. Operations at LET endpoints

```

1  void On_E_WRITE_start_execution() { // W
2      E_WRITE_tp_gvar_r = gvar_r;
3  }
4  void On_E_WRITE_end_execution() { // X
5      E_WRITE_o_gvar_r = gvar_r;
6      gvar_r = E_WRITE_tp_gvar_r;
7  }
8  void On_E_READ_start_execution() { // Y
9      E_READ_tp_gvar_r = gvar_r;
10     E_READ_tp_gvar_w = gvar_w;
11     gvar_r = E_WRITE_o_gvar_r;
12     gvar_w = T_RW_o_gvar_w;
13 }
14 void On_E_READ_end_execution() { // Z
15     gvar_w = E_READ_tp_gvar_w;
16     gvar_r = E_READ_tp_gvar_r;
17 }
18 void On_T_RW_start_execution() { // B
19     T_RW_tp_gvar_r = gvar_r;
20     T_RW_tp_gvar_w = gvar_w;
21     gvar_r = E_WRITE_o_gvar_r;
22 }
23 void On_T_RW_end_execution() { // C
24     T_RW_tp_o_gvar_w = gvar_w;
25     gvar_r = T_RW_tp_gvar_r;
26     gvar_w = T_RW_tp_gvar_w;
27 }

```

Listing 1.11. Operations at physical execution endpoints

the corresponding synchronous TDL task T_{RW} , and the two asynchronous TDL tasks E_{WRITE} , and E_{READ} , respectively. Assume that tt_read_write reads variable $gvar_r$ and writes into variable $gvar_w$, ev_write writes in variable $gvar_r$, and ev_read reads from both variables. An execution example is shown in Figure 6, and the corresponding TDL-Machine operations are summarized in Listings 1.9, 1.10 and 1.11

For this example, one can check that the LET requirements for data transfer regarding synchronous TDL tasks are satisfied. In particular, the value of $gvar_r$ read in the execution of tt_read_write is the one updated by a previous execution of ev_write , which is not shown in the figure (the one preceding the depicted execution). This is the value of the output port of E_{WRITE} at the beginning of T_{RW} 's LET. However, ev_read uses the latest value of $gvar_r$, updated during the depicted execution of ev_write . Thus, TDL modeling may introduce controlled delays in the communication involving synchronous TDL tasks, but it never delays communication between asynchronous tasks.

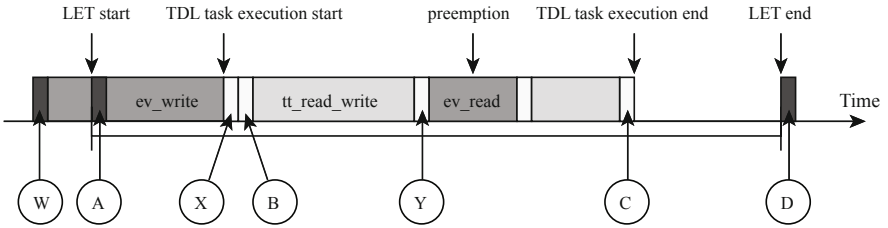


Fig. 6. Mixed time and event-triggered execution example

4 Industrial Application

The approach presented in this paper has been applied to an industrial engine control software, which comprises millions of lines of code and runs on top of an operating system with fixed priority scheduling. From the three periodic platform tasks, we identified 12 TDL tasks with periods between 1 millisecond and 2 seconds. The TDL tasks communicate via approximately 300 global variables. The application also has multiple event-triggered tasks, with the highest priority being given to the crank angle event. The TDL modeling procedure was optimized in order to decrease the additional memory required by TDL buffers, based on the following observations:

- If data transfer through a global variable cannot be affected by preemption, then no buffering is needed for that variable. For example, if all the readers and writers of a variable are part of the same platform task then the variable requires no buffering. Another example in this respect is a variable written only by an event-triggered task with lower priority than all the TDL reader tasks.

- Buffering can be reduced by directly substituting in the code original variables with TDL internal variables. For example, if a variable is written only by one TDL task, the variable can be substituted in the task’s code with the corresponding TDL output buffer variable, eliminating the need to buffer the original variable during the physical execution of the task. This buffering is illustrated in Listing 1.8, where in our example the occurrences of `gvar` in the task’s code are replaced by `T_WRITE_tp_o_gvar`, and `T_WRITE_tp_gvar` is eliminated. A similar reduction can be made for the cases where only one TDL task reads from an input variable.

4.1 Determining the Logical Execution Time

The two main design parameters related to the LET are the offset, i.e., the start of the LET in a task period, and the size of the LET. The LET must be at least as large as the worst case reaction time (WCRT) of the task, in order to ensure that the task is schedulable, i.e., every execution of the task ends before the end of the corresponding LET interval. Note that the maximum execution time of task T in the time interval $[t_1, t_2]$ is the maximum number of executions of T (in that interval) multiplied by the worst case execution time of T , ($WCET(T)$):

$$\left\lceil \frac{t_2 - t_1}{\pi_T} \right\rceil * WCET(T),$$

where π_T denotes the invocation period of T . If T is event-triggered, then it is considered periodic with period equal to the minimum inter-arrival time of the triggering event in the current operating mode.

Let the set \mathbf{T} of all tasks $T_i \in \mathbf{T}$ be ordered according to their priorities, where $i = 1$ means highest priority. The worst case reaction time of task T_i is given by the smallest fixed point of the following recursive equation [16, 17]:

$$R(T_i)^{(k+1)} = WCET(T_i) + \sum_{j=1}^{(i-1)} \left\lceil \frac{R(T_i)^{(k)}}{\pi_{T_j}} \right\rceil * WCET(T_j)$$

with $R(T_i)^0 = WCET(T_i)$. If no fixed point exists, the task is not schedulable.

The WCRT analysis can be employed to find LETs in the common case of legacy applications where platform tasks are scheduled with fixed priority preemptive scheduling. For tasks with the same priority, we distinguish between the following cases:

- Two TDL tasks belong to the same time-triggered platform task. In this case, their LETs are non-overlapping by design (see Section 3.3). Thus, the WCRT analysis can be applied to each of them independently of the other.
- Two TDL tasks belong to different platform tasks, and both platform tasks are time-triggered. In the usual case of the rate-monotonic priority assignment, the two platform tasks must have the same execution period. A well-designed legacy implementation ensures that the tasks are triggered at a fixed offset (i.e., not at the same time). Then the LET design is done as in the previous case.

- A TDL task belongs to a time-triggered platform task that has the same priority as other event-triggered platform tasks. Then the initial condition of the above recursion must include the sum of the WCETs of the event-triggered tasks.

The LET size is a trade-off between opposite requirements. For example, robustness requires larger LETs, while control performance requires smaller LETs, to minimize the additional reaction time incurred due to the LET execution semantics. The trade-off is inherently dynamic, since the relative importance of one requirement or another depends on the operating conditions at a given time.

For the engine control application, we have chosen to specify multiple modes of timing behavior, spanning the entire range of engine speeds, as follows:

- At high engine speeds, when the computational load is severe and fast response times are crucial, the LET of a TDL task is equal to the worst case reaction time of the task.
- At low engine speeds, the LET of a task can be larger by up to 20%. This leaves room for adding and testing new functions without affecting the timing behavior of the application.

Several TDL modes have been defined between the highest and the lowest engine speeds. Each mode has the same tasks, the only difference between modes being in the tasks' LETs. The modes have been established based on WCRT profiles of the TDL task functions, which represent the minimum LET for each engine speed. TDL mode switches are triggered by the engine speed variable, which is an input to the TDL:Machine. Figure 7 shows an example of a WCRT profile which defines five possible timing modes. Note that the LET of task T cannot exceed the task's period $\pi(T)$. Thus, at engine speeds higher than e_2 the task is deemed unschedulable.

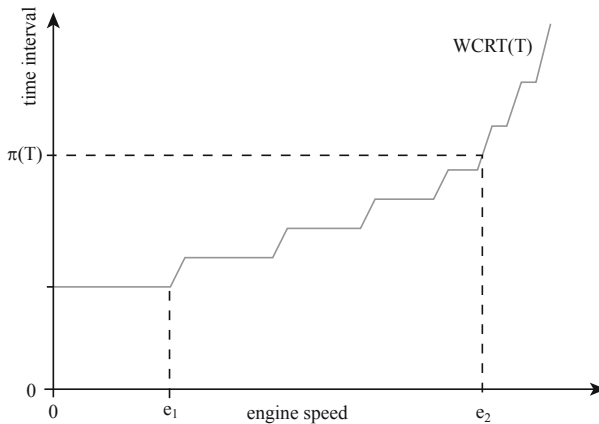


Fig. 7. Worst case reaction time of task T as function of the engine speed

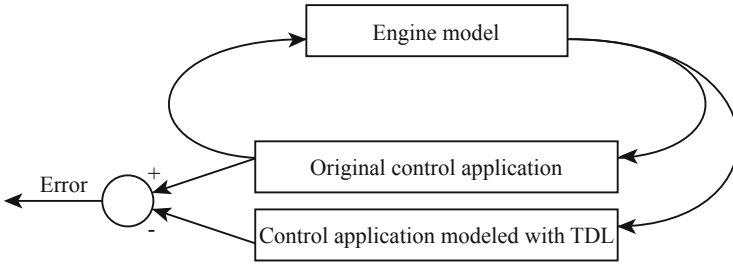


Fig. 8. SIL setup for comparing TDL-based application with original application

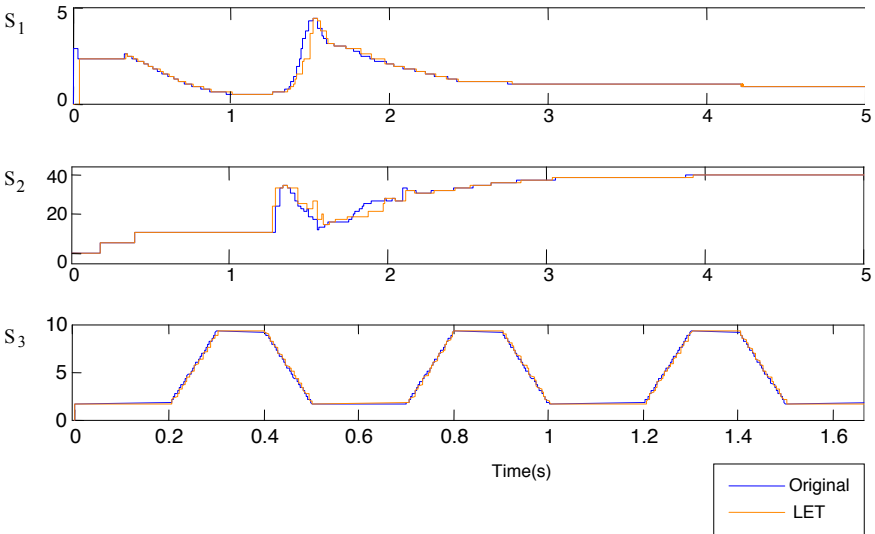


Fig. 9. Signal comparison

The offset of a TDL task T was determined according to the "place" of the TDL task in the owner platform task. In principle, the LET start of a TDL task is the same as the LET end of the preceding TDL task in the same platform task. A TDL task which is always executed first in its platform task has an offset equal to zero. For example, in listing [L.3](#) and Figure [4](#) the task with period 5ms has offset zero (it is always executed first), while the offset of the 10ms task (always executed second) equals the LET of the first task.

The WCETs have been estimated by using the *a3* tool [\[18\]](#). This tool relies on an accurate model of the processor, which was not available at the time of testing. Therefore, a "vanilla" version of the processor was used and consequently the estimates were quite conservative.

4.2 Testing Results

The application modeled with TDL has been tested in a software-in-the-loop (SIL) simulator [19, 20], as well as in a hardware-in-the-loop testbed (HIL). The SIL testing compared signals from the original and TDL-based applications when both were running in parallel, in closed-loop with an engine model, as schematically shown in Figure 8. A sample of the testing results is provided in Figure 9, where one can observe that the behavior of the TDL-modeled system is close to the original one, modulo some small delays introduced by the LET behavior. The delays resulted from using the worst-case scenario for setting the LET of the task and from overestimation of WCET of tasks. They could be reduced by using more accurate execution time estimations.

The HIL testing was performed in a testbed where the original and TDL-based applications were executed on the same Electronic Control Unit interconnected with a real-time computer running a model of the engine. Signals were sampled with a period of 50 microseconds and then compared. In one of the test cases, the robustness of the system was tested by making a non-functional modification in the code and comparing the outputs. In the original software, the change led to a difference in the outputs, while no difference was exhibited in the TDL-modeled version.

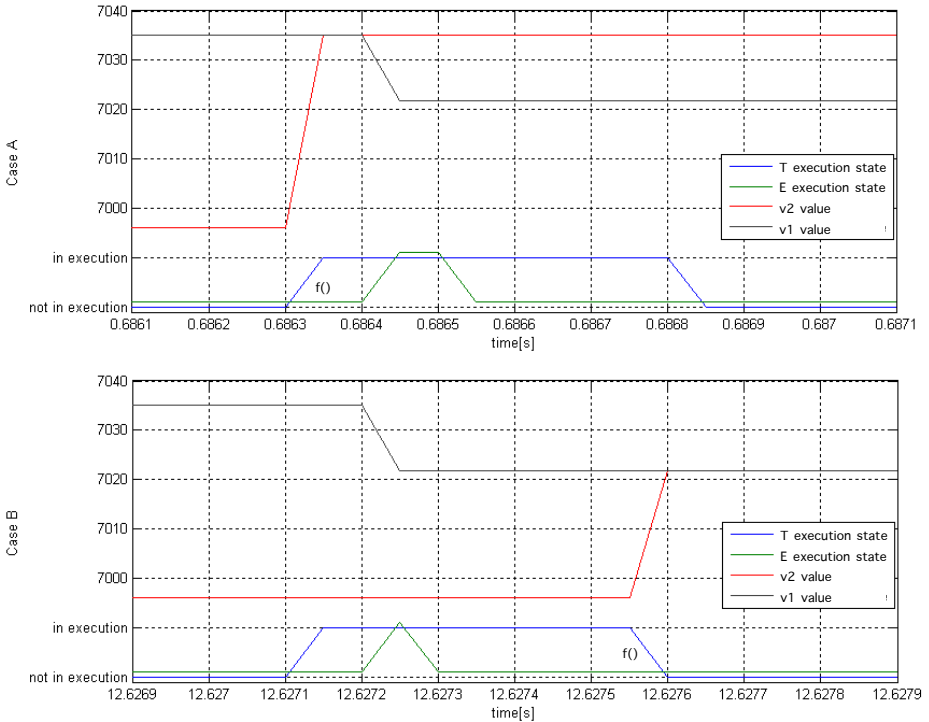


Fig. 10. Changing the place of a function call leads to different values of the output v_2

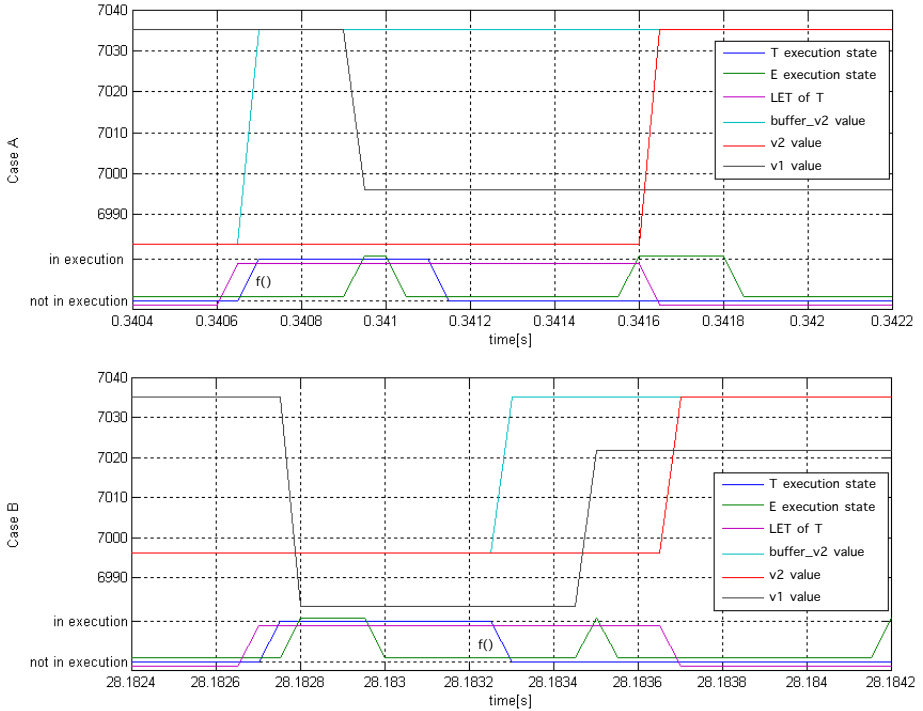


Fig. 11. The output value of v_2 is unaffected in the TDL-based system

Usually, a task function contains a sequence of calls to multiple process functions that need to be executed when the task is triggered. Many such process functions are independent and conceptually the order in which they are called does not matter. We have chosen a periodic task T and moved a process function f from the beginning of the task T to the end. The function f updates a global variable v_2 with the value of another global variable v_1 . The latter is updated by an event-triggered task E , which has higher priority than T . The variables v_1 and v_2 are regarded as an input, respectively an output of T . Case A in Figure 10 illustrates an execution of T in the original application, where f is called at the beginning of T 's execution. Then, the value of v_2 is set to the value of v_1 . Thereafter, E preempts the execution of T and updates v_1 . Case B shows an execution of T in the modified application, where f is executed at the end of T 's execution. Here, the variable v_2 is updated after E 's preemptive execution, and consequently the output of T is different. Notice that no difference occurs when E does not preempt. This is an example where different ways of serializing executions of concurrent components lead to different behaviors in the system, due to preemption from event-triggered tasks. The TDL-modeled system is robust with respect to such changes, as demonstrated by the corresponding cases in Figure 11. In this case, the TDL task T was the only writer of v_2 , hence in

the TDL version the occurrence of v_2 in the code of f was replaced by a TDL buffer output variable $buffer.v2$. Notice that this is updated with the value of v_1 when f is executed and then v_2 is updated with the value of $buffer.v2$ at the end of the LET. One can see that the output of T does not change from case A to case B.

5 Conclusions

This paper presents an approach to enable adaptivity of legacy real-time embedded software based on introducing a semantics-preserving execution environment conforming to the LET discipline of the Timing Definition Language.

Modeling the timing behavior of legacy applications with TDL represents also an instance of bridging the gap between the general benefits advocated by Model-Based-Design approaches (such as predictability, separation of concerns, portability), and the efficiency-oriented design of legacy applications. It is a meet-in-the-middle process, with the top-down direction assumed by TDL and the bottom up direction required by the legacy application.

The method for applying TDL timing specifications to legacy control software focuses on achieving the required timing behavior with minimal intervention in the original application. The paper presents the structure of the runtime system and instrumentation, which are automatically generated from the timing specification and from information about the legacy source code and platform. This approach has been successfully applied to a complex legacy controller system in the automotive domain. Detailed description of other important aspects such as dealing with schedulability and the actual code generation algorithm are omitted due to lack of space.

The refactored legacy system is schedulable if it can be executed such that the TDL timing specifications are satisfied, i.e., every physical execution of a synchronous TDL task takes place in the associated LET interval. Achieving schedulability is especially difficult when asynchronous tasks have higher priorities than synchronous TDL tasks.

It is worth noting that the described TDL modeling can be applied incrementally on a legacy application, starting with a single synchronous TDL task and stepwise adding more synchronous tasks. At each step, the system can be tested for schedulability, as well as for timing and functional properties. This makes the approach feasible in practice and recommends it as the core of a structured process for incremental migration of large legacy software towards more predictable and adaptive systems.

References

- [1] Kirsch, C.M., Lopes, L., Marques, E.R.B.: Semantics-preserving and incremental runtime patching of real-time programs. In: Proc. Workshop on Adaptive and Reconfigurable Embedded Systems, APRES (2008)

- [2] Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Giotto: A time-triggered language for embedded programming. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 166–184. Springer, Heidelberg (2001)
- [3] Caspi, P., Scaife, N., Sofronis, C., Tripakis, S.: Semantics-preserving multitask implementation of synchronous programs. *ACM Trans. Embed. Comput. Syst.* 7, 15:1-15:40 (2008), <http://doi.acm.org/10.1145/1331331.1331339>, doi:10.1145/1331331.1331339
- [4] Object Management Group: Model driven architecture (2010)
- [5] Sangiovanni-Vincentelli, A.: Defining platform-based design. *EEDesign of EE-Times* (2002), <http://www.gigascale.org/pubs/141.html>
- [6] Ghosal, A., Sangiovanni-Vincentelli, A., Kirsch, C.M., Henzinger, T.A., Iercan, D.: A hierarchical coordination language for interacting real-time tasks. In: EMSOFT 2006: Proceedings of the 6th ACM & IEEE International Conference on Embedded Software, Seoul, Korea, pp. 132–141. ACM, New York (2006), doi:10.1145/1176887.1176907
- [7] Pree, W., Templ, J.: Modeling with the timing definition language (TDL). In: Broy, M., Krüger, I.H., Meisinger, M. (eds.) ASWSD 2006. LNCS, vol. 4922, pp. 133–144. Springer, Heidelberg (2008)
- [8] Lee, E.A.: Computing needs time. *Commun. ACM* 52(5), 70–79 (2009), doi:10.1145/1506409.1506426
- [9] Templ, J.: TDL - Timing Definition Language 1.5 Specification. Technical report, University of Salzburg (2008), www.chrona.com
- [10] Chrona: The TDL tool chain (2010), <http://www.chrona.com>
- [11] OSEK: OSEK/VDX operating system specification (2010), <http://www.osek-vdx.org/>
- [12] AUTOSAR Consortium: Specification of multi-core OS architecture v1.0, AUTOSAR release 4.0 (2009)
- [13] Monot, A., Navet, N., Simonot, F., Bavoux, B.: Multicore scheduling in automotive ECUs. In: Embedded Real-Time Software and Systems (ERTS 2010), Toulouse, France (2010)
- [14] Farcas, C.: Towards Portable Real-Time Software Components. PhD thesis, University of Salzburg (2006)
- [15] Ghosal, A., Iercan, D., Kirsch, C., Henzinger, T., Sangiovanni-Vincentelli, A.: Separate compilation of hierarchical real-time programs into linear-bounded embedded machine code. In: Proceedings of the APGES Workshop, Salzburg, Austria (2007)
- [16] Audsley, N., Burns, A., Richardson, M., Tindell, K., Wellings, A.J.: Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal* 8, 284–292 (1993)
- [17] Joseph, M., Pandya, P.: Finding response times in a real-time system. *The Computer Journal* 29(5), 390–395 (1986), <http://dx.doi.org/10.1093/comjnl/29.5.390>
- [18] Absint: aiT worst-case execution time analyzers (2010), <http://www.absint.com/ait/>
- [19] Resmerita, S., Derler, P., Lee, E.A.: Modeling and simulation of legacy embedded systems. Technical Report UCB/Eecs-2010-38, EECS Department, University of California, Berkeley (April 2010)
- [20] Derler, P.: Efficient Execution and Simulation of Time-Annotated Embedded Software. PhD thesis, University of Salzburg (2010)

Towards IT Systems Capable of Managing Their Health

Selvi Kadirvel and José A.B. Fortes

Advanced Computing and Information Systems Lab
NSF Center for Autonomic Computing
University of Florida
Gainesville, Florida
{selvik,fortes}@ufl.edu

Abstract. Self-caring systems are systems capable of monitoring and managing their own health and, indirectly, their useful lifetime. Unlike self-healing systems which are reactive to faults and failures, self-caring systems are aware of their health and hence can potentially circumvent and adapt to impending faults, or recover from them quicker and more effectively. Towards a methodology to model and incorporate health management logic and control mechanisms into an Information Technology (IT) system whose health needs to be managed, we propose the following: 1. the use of Petri nets as a discrete event system (DES) graphical model that can also be used for analysis, simulation and execution control, 2. the use of Remaining-Useful-Life (RUL) management and prognosis as a novel way of looking at health management in IT systems 3. the use of a control theoretic framework for RUL management. As a simple illustration of the concept, a controller was built for useful life management in the application execution stage (containing a potential memory exhaustion fault) of an IT system.

Keywords: autonomic computing, self-caring, health management, Petri net, self-healing, modeling, discrete event systems.

1 Introduction

A growingly common practice in IT design, is that of re-using or combining existing systems to quickly and effectively build larger and more complex systems. The component systems can be software, hardware, data, services, or combinations thereof. They could be passive (e.g. a software module) or active (e.g. a running service), static (e.g. a software library) or dynamic (e.g. an evolving service), and they could be owned by different entities. Of particular interest to this paper are systems built out of active dynamic subsystems controlled by different entities. These systems are becoming common with the emergence of Web services, hosting services, and cloud-provided infrastructure, platforms, and applications that can be combined and composed for different purposes.

The field of autonomic computing has emerged to deal with increased IT system complexity by achieving goals such as self-configuration, self-healing, self-optimization and self-protection. The desirable ability of an IT system to manage

its own health has some overlap with the self-healing property in autonomic computing. However, the self-healing property refers to the ability of a system to recover from events that cause system failure or operational malfunctions [1], whereas health management refers to the system's ability to detect, isolate and identify behaviors leading to faults (as part of diagnosis) and predict impending failures (as part of prognosis) so that corrective action can be taken before the faults occur or progress to a system failure.

The prerequisite to achieve any self-* property is self-knowledge. This is where the field of modeling comes in, by providing a system with a representation of itself. This field, with its vast and rich literature, allows for various types of models such as functional models, reliability models, performance models, cost models and security models, each to capture the behavior that is of interest in the system under study [2]. In the area of modeling of compute infrastructures, extensive research results exist for the modeling of web servers, web applications, multiprocessors, as well as these components put together (termed as multi-tier architectures) [3] [4] [5] [6] [7] [8].

In the context of self-caring systems, the models of interest require the identification of health indicators, the mechanisms available to control these indicators and their dependencies on time, environment and operational parameters. An additional challenge is the representation and modeling of interactions among components that capture how a component's health impacts the health of other components and the whole system. The focus of this work is not on the modeling the internals of individual subsystems. Instead the goal is to capture a subsystem's susceptibility to faults as a black box that performs an activity, and the interactions between subsystems in a typical usage environment. Our approach views any IT system of systems as a discrete-event system whose logic and interactions are captured by Petri nets, a graph-based mathematical formalism that has been successfully applied in, among others, the areas of communication protocols, manufacturing systems [9], and multiprocessor systems.

Health management refers to the taking of appropriate actions based on current system health and estimated future system state with the goal of keeping the system operational. Health management requires six capabilities - monitoring, diagnosis, prognosis, planning, remaining-useful-life management and remediation [37] [10]. The important benefit of health management is that by detecting an incipient fault before it progresses into an error or failure, this unhealthy condition can be treated by a range of recovery actions while possibly avoiding interruption of service. When compared with self-healing and fault-tolerant designs, these actions are typically cheaper to perform, result in shorter system downtimes and thus improve system availability and performance. As illustrated by Figure 1, by detecting the fault (or its precursors) in a component before it affects the subsystem or system that it is a part of, we can replace the component with a non-faulty one or even repair the faulty component. The cost of replacing a component or eliminating the causes of its failure is more affordable than the cost of replacing a subsystem or system and recovering from its failure. Prognostic Health Management (PHM) technologies have been successfully deployed in

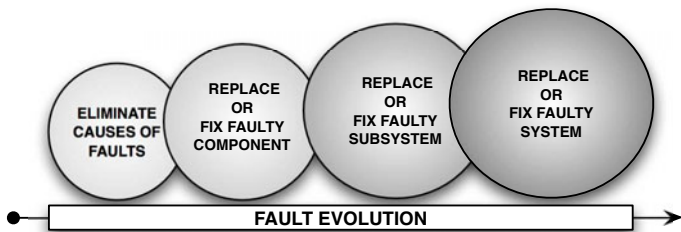


Fig. 1. Recovery actions corresponding to different steps of the progression from fault precursors to component faults to subsystem faults to system failure. Larger circles represent larger recovery costs.

other contexts including mechanical, structural and electronic systems [10] [11] [12] [13] [14] [16] [17]. Where applicable, concepts and techniques used in these implementations can be leveraged for IT infrastructures.

This paper is organized as follows. The concept of health management as it applies to IT systems is presented in Section 2. Section 3 motivates the need for modeling and brings out the suitability and benefits of the chosen model, introducing various aspects of Petri nets that are used in the proposed framework. The health management modeling methodology is presented in Section 4. Section 5 outlines related work in the field, and Section 6 discusses future directions of our work and concludes.

2 Health Management

Health management consists of six main actions: monitoring, diagnosis, prognosis, planning and remaining-useful-life (RUL) management, and remediation. Figure 2 is a high-level view of the health management architecture envisioned by our work. The subsections below provide an overview of system operation and the constituents of this architecture. The focus of our contribution is on RUL management and hence this sub section is described in detail with an example scenario.

2.1 Health Indicators

A health indicator is defined as a system attribute that is indicative of possibly degrading health. Tracking of health indicators can be used to detect fault precursors.

The choice of health indicators depends on the types of faults to be anticipated and the subsystems that make up a system. In this paper, we focus on a large and important class of faults, namely faults caused by exhaustion of resources. In this context, a resource is broadly defined to mean any type of entity or attribute that is consumed by the system and is typically available in finite supply. Resources can correspond to physical hardware, software or middleware

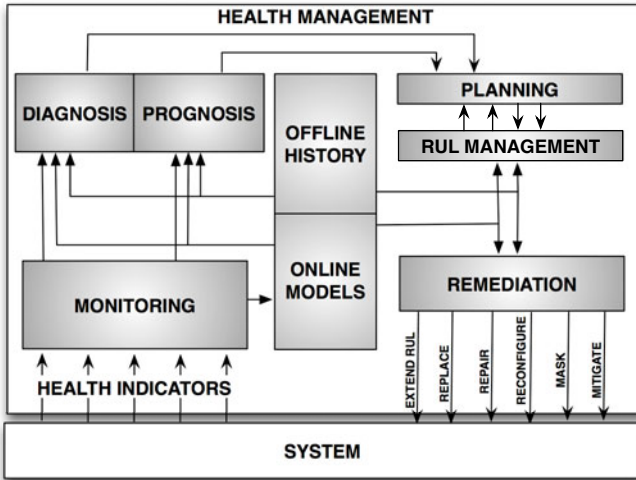


Fig. 2. Health Management Architecture

objects or attributes. The availability of minimum required amounts of resources is essential for the proper operation of IT subsystems. Typical hardware subsystems include computational subsystems (compute nodes), storage subsystem (I/O servers, RAID arrays) and networking subsystem (switches, NIC cards). Software includes the application software and system software that execute on the hardware resources. Middleware includes services such as resource managers, job schedulers, job queues and application servers that enable services to inter-operate, enabling the IT deployment to provide intended services to its users. The health of any these subsystems is reflected by their ability to perform their operations in the present as well as till an intended point of time in the future. Resource exhaustion can have many different causes. These include improperly implemented functionality in software such as not releasing all dynamically allocated memory, not ensuring termination of all created child processes, not releasing numbered resources such as socket descriptors, file descriptors, etc. Resource exhaustion can also result from software aging, unanticipated workloads or malicious code invocations. Furthermore hardware faults can also affect the availability of resources. A database of software vulnerabilities collected and organized by the US Department of Homeland Security [18] shows that an important class of system failures is those that result from resource-exhaustion faults. Numerous production software systems have recorded occurrences of this class of failures such as operating systems, DNS servers, web servers, etc.

For example, a failure can occur in a compute node when an application cannot allocate more memory (memory is exhausted due to memory leaks), create new threads (the process limit for number of threads has been exceeded), open new files (the process limit for number of file descriptors has been exceeded) or add content to an existing file (the process limit for maximum size of the file

has been exceeded). In a denial-of-service type of attack, resource exhaustion occurs typically because too many packets were received; too many sessions were initiated or too much ‘malicious’ workload was received; which in turn prevents legitimate packets, session or workload from being processed.

For the purposes of health management, we classify system operation into two regions: a normal region in which no resource is near exhaustion and a stressed region in which at least one type of resource is being used beyond a predetermined limit but prior to exhaustion. The difference between the safe limit and the hard limit is the remaining useful resource gap; if resources are used at a known rate (or based on a known pattern), one can determine the remaining useful life from the resource gap. The two operating regions can be statically or dynamically determined. A static region can be determined by parameters set as environment variables, virtual machine (VM) parameters, etc. This can be controlled by the designer of the system of systems. For example, the designer can request a VM with some resources (CPU, memory, storage) whose values determine the hard limits. Then the designer can determine safe limits beyond which RUL management is needed. Another example is at the operating system level, where for each process, hard limits can be set through operating system hooks for each of a process’ resources such as number of threads, number of file descriptors or maximum size of the process’ stack or data segment. Other hardware resources such as storage servers or switches have hard limits to their data access rates as well as capacities.

An important benefit of the demarcation between normal and stressed regions of operation is that the frequency and overhead of monitoring can be controlled based on current operating region of the systems of interest. The experience of an administrator or application expert can be leveraged to estimate safe limits. This may not always be possible in which case a learning method must be put in place (offline or online). A dynamic demarcation between the normal and stressed region can be determined by detecting behaviors which are deemed anomalous. This is a larger problem that is beyond the scope of this paper and left for future work. Using this perspective, the static approach corresponds to the case where anomalous behavior corresponds to resource usage exceeding a safe limit.

Figure 3 attempts to demarcate the difference in system states as considered by health management and failure management. In the normal functional state, the system is working within its typical/nominal operating region. The system is in a stressed mode of operation in the stressed functional state. For example, in the case of resource-exhaustion faults, the system moves to this state if the consumption of any resource exceeds the safe limit. This could happen because of an anticipated fault (such as a growing memory leak). The third state is a degraded functional state in which the system is still functioning but at reduced performance. RUL extension techniques can be used to prolong a system’s operation in either the stressed or degraded functional states, thereby delaying a possible move into the fault-present state. The lifetime gained using RUL extension techniques can then be effectively used to evaluate various health

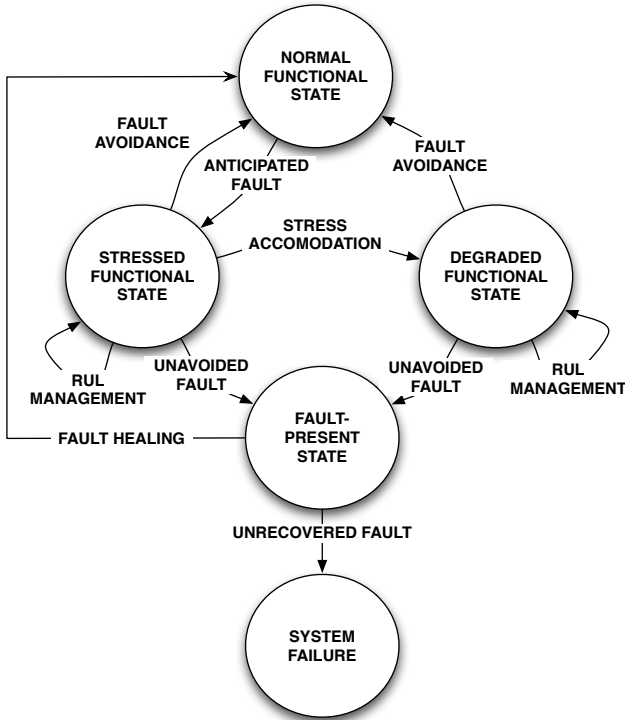


Fig. 3. Diagram depicting system health states, actions and events that determine state transitions

recovery techniques (planning) and initiate an appropriate fault handling technique (remediation). This fault avoidance action will move the system back to its normal functional state. RUL extension techniques could be applied at the subsystem level but interactions between multiple subsystems could cause system performance degradation. This is indicated by the stress accommodation transition between the stressed and degraded functional states.

When in the fault-present state, fault-healing techniques (such as fault-tolerance and self-healing) can be utilized to bring the system back to the normal functional state. If such recovery is not possible, then the system may progress to system failure. The goal of health management is thus to keep the system functioning in the three functional states described above.

2.2 Monitoring

Monitoring refers to the collection of relevant data about operating conditions of different subsystems that constitute the system under study. Health monitoring mechanisms depend on the choice of health indicators and the type of systems to be monitored. In mechanical and electronic components this is performed

using sensors (temperature, pressure, vibration). In software components this is in the form of log files and software probes that could measure application performance, resource consumption, environmental constructs or internal state of the application. In systems of considerable size, the raw data collected is massive and needs to first be reduced by filtering. Feature selection, the process of choosing those data items that are possible indicators of health, needs to be done, either automatically or using the experience of human designers. Feature extraction is then executed in a computationally efficient manner. The extracted features produced by these data preprocessing steps serve as the input for the following diagnostic and prognostic steps. Consumption of resources by a subsystem can often be tracked by readily available sensors, e.g. via operating system level commands.

2.3 Diagnosis

In the above context, health monitoring takes one of two forms - direct detection of resource usage beyond safe limits, and indirect detection via monitoring of performance deterioration attributable to overuse or unavailability of a certain type of resource. In the case of a web server, an example of an indirect measure is the response time. An unusual response time (either too high or too low) may indicate a possible health decline. For a database server, query-processing time is a similar indicator. Diagnosis is the process of using extracted features to detect fault precursors, fault conditions and fault locations, and to characterize the nature and extent of the fault incipency. In some cases (including resource exhaustion) the causal relationship between a health indicator and a potential fault is either known or easy to establish. In other cases diagnostic relationships may have to be learned either offline or online from data that includes values of health indicators and fault occurrences. Fault clustering, classification and decision-making can be performed on this data using statistical and machine learning techniques.

2.4 Prognosis

Prognosis is the process of predicting the time-evolution of a fault or the remaining useful life of a predicted-to-fail component, after an incipient fault has been detected. Prognostic approaches can be data-driven, probability-based or model-based. With the recent development of control-theoretic approaches to modeling and controlling computing systems, model-based techniques can be employed for prognosis in IT systems. Useful life predictions from the prognostics step are then used to initiate preventive maintenance or schedule future maintenance activities.

In the case of impending resource exhaustion failures, a model of resource consumption as a function of time and/or activity (process invocation, user requests, etc) is required. The model can be learned from observed operation in the normal region or in the stressed region or both.

2.5 Planning

Based on the results of the prognostic and diagnostic modules, the HM system needs to initiate appropriate health remediation actions. In the case when an incipient fault is discovered, it may be possible to slow down or reverse fault progression by mitigating or eliminating its causes. If the fault cannot be delayed or avoided, recovery of the component, subsystem or system, might have to be initiated. The choice of the remediation action will depend on the useful life available, cost of remedial actions and duration of downtime that can be tolerated by the IT subsystem being managed. When it is found that the estimated RUL available may not be sufficient for invoking planning algorithms or the remediation operation, then RUL management techniques are employed to attempt to gain valuable useful life.

2.6 Remaining Useful Life Management

Remaining Useful Life (RUL) is defined as an estimate of the time after which a component will fail with high probability. The factors that affect the RUL include workload stresses and operating conditions. The latter includes interactions with other subsystems or components, wear and tear of components due to age, environmental changes, etc. RUL management refers to the process by which the useful life of a system is controlled using some of the factors that affect it, thereby allowing for the extended life to be used effectively. The RUL of a system is determined by the RUL of each of its subsystems. In the case when fault progression in each of these components is independent of progression of faults in other components, then system RUL can be estimated as the shortest RUL of all the components. When fault progression is not independent, explicit models need to be constructed to capture these correlations. Figure 4 shows how system RUL is extended by prolonging useful life of one of its components.

Applying Classical Control Theory - This section will detail an approach to RUL management through the application of the principles of feedback control

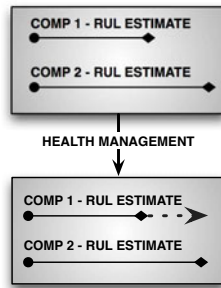


Fig. 4. Prolonging system RUL by extending the RUL of subsystem labelled COMP 1 to match that of subsystem labelled COMP 2

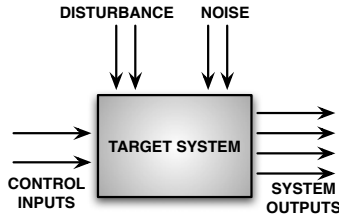


Fig. 5. Overview of target system - In control systems, the target system is viewed to have a finite set of control inputs that can modify system behavior, disturbance and noise inputs that can have affect system behavior and a set of systems outputs that are indicative of system operation

theory. The system of interest, shown in Figure 5, is referred to as the target system. It consists of a number of subsystems or components that are essential for its operation. The measured output of the target system will be the rate at which RUL decreases for its different subsystems. As in previous sections, our scope is restricted to resource-exhaustion failures. The control input consists of those factors of the target system that can be controlled in order to affect the subsystem RUL rates. The disturbances consists of those factors that will affect RUL rates but cannot be controlled and need to be compensated for by the system controller. The noise inputs correspond to measurement noise that gets added to the measured system output. Figure 6 shows the feedback control loop of the target system. The goal of the feedback controller is to control the rate at which component RUL decreases with the ultimate goal of controlling and prolonging system RUL. It is important to note that for any target system it is a challenging task to estimate component RUL and system RUL. As a first step towards solving this challenging problem, we focus our attention on the rate at which RUL decreases in the case of resource-exhaustion faults.

Example Scenario - Let us take the example of a server in a virtualized environment, which is a common subsystem in many IT systems. Since we are interested in resource exhaustion failures, the HM module will check whether resource consumption is within the normal operating region. When the operation enters a stressed region, the HM feedback controller will be invoked to keep the

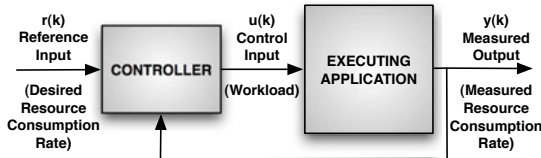


Fig. 6. Feedback control loop of target system. Reference input is the desired rate of RUL change and measured output is the measured rate of RUL change.

resource depletion in check. We created a synthetic test bed to create a proof of concept implementation of a feedback controller.

The target subsystem of interest in our experiment consists of a virtual server running an application. We consider a CPU-intensive workload that is similar to a workload running on a compute node in an HPC environment or a cloud infrastructure. In this experiment we chose to study memory exhaustion failures and so a memory leak was introduced in to the application code. The output of the target system is the rate at which memory is depleted, which indirectly provides a measure of the RUL. The control input is the workload that is to be processed by the application code. This target system is modeled as a first-order linear system.

System Identification - System identification is the process of determining the relationship between the control input and measured output of the target system. The application was executed for different workload values and the memory leak rate was measured for each. Linear regression was used to find values of the model parameters. Figure 7 shows the relationship between actual values and predicted values of the memory leak rate using the model. The deviation of the points from the line captures the difference between the model and observed empirical values.

Controller Design - For the determined model parameters, the closed loop transfer function of the target system is determined. In order to design an integral controller, the range of possible controller gain (K) values are determined such

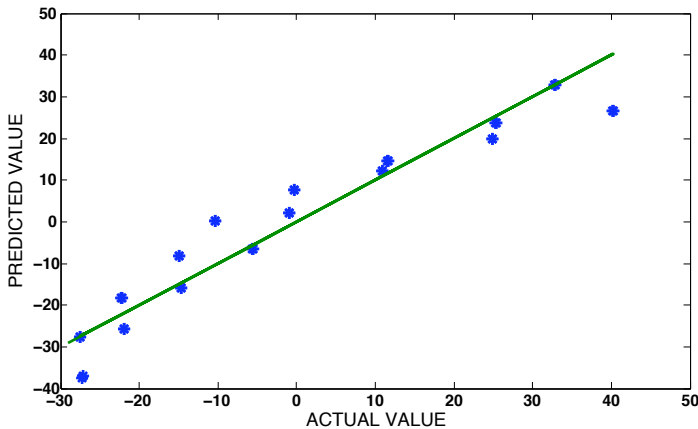


Fig. 7. Graph comparing predicted values $y_{pred}(k)$ of output (using model) with measured output values $y(k)$ (offset from mean) during the experiment. $y_{pred}(k) = a.y(k-1) + b.u(k-1)$, where $y(k) = y_{measured}(k) - y_{mean}(k)$ and $u(k) = u_{measured}(k) - u_{mean}(k)$ and a, b are model parameters. The line is shown as a guide to indicate those points where actual and predicted values match exactly (a perfect model). Asterisks represent data points obtained from experiment.

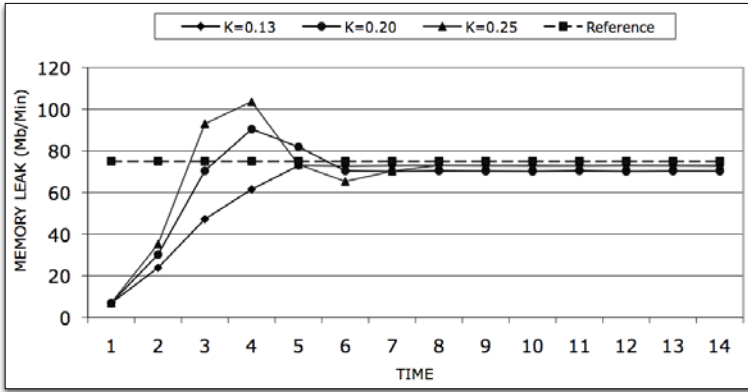


Fig. 8. Graph showing measured output from target system for different controller gains and reference input signal to closed loop system

that the stability condition (pole magnitude is less than one) of the system is always satisfied. Since model parameters only approximate system characteristics, empirical studies need to be performed to determine good controller gain values. Experiments were conducted for different controller gain values. The graph in Figure 8 shows the measured output and reference input for K values of 0.13, 0.20 and 0.25. The overshoot and settling times are determined by the dominant pole in the closed loop system. The best controller gain is chosen based on the needs of the system.

In these experiments, we consider one measured output from the target system, namely memory leak rate and one control input, namely workload. Future work will extend this Single-Input Single-Output (SISO) target system model to a Multiple-Input Multiple-Output (MIMO) model in which the measured outputs will include different subsystem RUL rates.

2.7 Remediation

Remediation refers to the actions that are invoked for fault masking, mitigation, prevention or recovery. In the case of a possible resource exhaustion fault, the useful life of a system operating in the stressed region can be extended either by adding more resources to the system or by decreasing the workload experienced by it. For example when we have a group of web servers processing client requests the number of servers can be expanded seamlessly to reduce overloading of any specific stressed server. The throughput of the system as a whole is maintained (at the cost of additional resources). In the case of virtualized environments, a stressed server's resources can be transparently increased when the server continues to remain online.

Given the above principles of health management, our approach to systematically incorporate health management in to IT systems consists of using a modeling framework to capture the IT system of interest and then use this model as

a system manager to control and operate the system in such a way as to keep its health managed. The following section details the proposed modeling framework and methodology.

3 Modeling Framework

3.1 Requirements of a Suitable Model

In a typical IT system, process invocation and completion events, management events and failure events all occur in an asynchronous and concurrent manner. Thus, in general, an IT system is not amenable to description through discrete or continuous variables modeled by difference or differential equations - it is best viewed as a Discrete Event Dynamic System (DES) [19]. State changes are initiated by the occurrence of the mentioned events in distributed components that are linked through a network. So in addition to asynchrony and concurrency, the model should capture functional dependencies and sequential ordering between interacting components. The model should also provide a way to represent quantitative parameters that define these interactions so that analysis of performance, reliability and QoS measures can be performed. A Petri net is one candidate model that meets these requirements.

3.2 Petri Net Basics

A Petri net is a bipartite graph where the two types of nodes are places and transitions. A Petri net is defined by the quintuple (P, T, I, O, M_0) where $P = \{p_1, p_2, \dots, p_n\}$ is a finite non-empty set of places, $T = \{t_1, t_2, \dots, t_m\}$ is a finite nonempty set of transitions. I is the set of input arcs connecting places to transitions and O is the set of output arcs connecting transitions to places. A marking is defined as a mapping of tokens to places in the net. M_0 is the initial marking. In the graphical notation for Petri nets, places are represented as circles, transitions as rectangles or simply lines and tokens as dots inside places.

Petri net execution consists of the ‘firing’ of transitions. A transition may fire when it is enabled. A transition is considered enabled when the number of tokens in each of its input places is larger than or equal to the number (or cardinality or weight) of arcs connecting that input place to the transition. The firing of the transition results in the production of tokens in all the output places of the transition, the number of which depends on the cardinality of the output arcs.

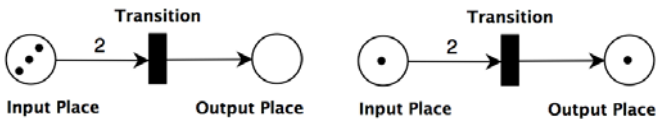


Fig. 9. Simple Petri net before and after firing

Figure 9 shows a simple Petri net with two places and one transition, both before and after firing. The various benefits of Petri Nets as well as the properties of interest to our application are outlined in the Appendix.

3.3 Petri Net Extensions of Interest

Numerous extensions to the basic Petri net model have been developed over the years to apply them for different purposes and types of systems. The following are the extensions that we found useful for representing autonomic properties and health management in IT systems:

Colored Petri Nets (CPN) - Tokens are associated with values belonging to different types (also called colors). This allows for more compact representation of complex nets [20].

Stochastic Petri Nets (SPN) - Random firing delays are associated with transitions whose firing is atomic [21]. Random delays are exponentially distributed.

Stochastic Reward Nets (SRN) - A reward rate is associated with each marking to enable evaluation of the reliability of complex systems [22].

3.4 Hierarchy Modeling

IT health management needs to take into consideration numerous subsystems, their structural connections and functional interdependencies and the processes that use different types of resources. One of the most intuitive ways to handle

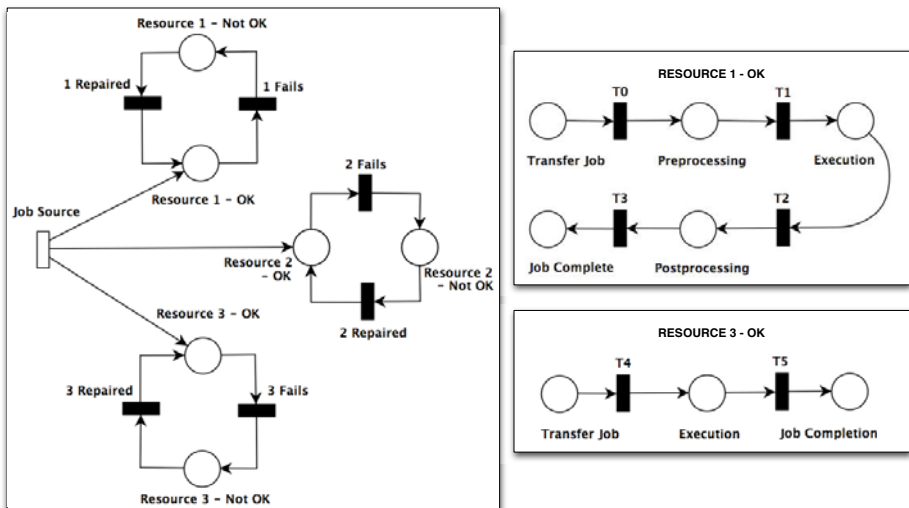


Fig. 10. Hierarchical Petri Net Model At the resource selection level, each place such as Resource 1 - OK or Resource 3 - OK can be expanded in to its own Petri net model (shown in boxes titled RESOURCE 1 - OK and RESOURCE 3 - OK)

this complexity is through hierarchy in modeling and Petri nets provide a well-established paradigm to represent hierarchy.

A Petri net place is made to represent a subsystem. This means a token entering that place would subsequently enter a lower level Petri net that is embedded in that place. Let us consider the case of a business environment. A job represents a task that needs to be performed. At the highest level, a resource needs to be selected that is capable of performing this task. Each resource is represented by two places, one in the available condition (represented as OK in the figure) and the other in the failed condition (represented as Not OK). Each resource place consists of a Petri net that models the functioning of that resource. Figure 10 shows the described scenario.

4 Health Management Modeling Methodology

We introduce a modeling methodology to transform knowledge about how a system operates into a Petri net model and incorporate health management techniques into it. Our initial focus is on systems that consist of the execution of a sequence of activities, each with its own set of resource requirements. In this context we define a resource to be any of a hardware, software or middleware entity that is consumed and is available in finite supply.

4.1 Step I - Modeling System Structure and Dependencies

In this step, the system is modeled as a sequence of activities or stages. Each stage or activity is associated with an n-dimensional hypercube of resources that is required to perform this activity. This step thus captures both ordering of activities as well as the dependencies between resources and activities. It is assumed that resource exhaustion failures do not occur and the system is not yet being controlled or managed.

The system model consists of places that are of two types: activity places and resource places. Existence of a token in an activity place indicates that the activity is being performed on an object within that place. This object could represent a job or client request. The existence of a token in a resource place indicates that a unit of that resource is available for consumption. Since we are considering an n-dimensional resource space each resource type corresponds to a specific colored token. Transitions represent events such as the start and end of an activity. The firing of a transition indicates the occurrence of this event. The pre-resource place of a transition corresponds to the conditions that are required for an event to take place or the transition to fire. The output arcs connecting a transition to a resource place indicates the return of a resource instance back to the pool of available resources.

For modeling purposes we use a generalized stochastic Petri Net (GSPN) model which is an extension to SPN. GSPNs allow two types of transitions, one whose firing delay is zero and the other whose firing delay is exponentially distributed. This provides flexibility in capturing different transition firing times.

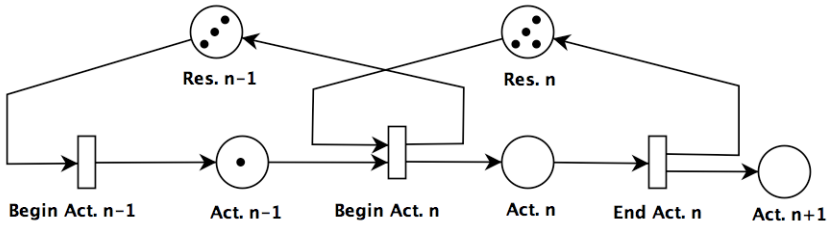


Fig. 11. Partial Petri net model showing activity dependencies on other activities and resource availability. Transition *Begin Act.n* fires after a token is available in *Act.n-1* and *Res.n*. (indicating that a task that has undergone activity *n-1* can move on to activity *n*, only after activity *n-1* is complete and when resource *n* is available).

Figure 11 shows the model of a part of a system that consists of three activity places $\{\text{Act } n-1, \text{Act } n, \text{Act } n+1\}$ and two resource places $\{\text{Res. } n-1, \text{Res. } n\}$. In this figure, each resource place is one-dimensional (in general, by using colored Petri nets, multiple resource dimensions can be captured by a single place). A transition between two activities represents both the end of the previous activity as well as the beginning of the next activity. Activity ‘*n*’ can begin only if a token exists in activity place *Act. n-1* and in the resource place *Res. n*. After activity *Act. n* completes, a token (corresponding to one unit of resource *Res. n*) is returned to the pool. The bound on a resource place will correspond to the maximum number of available resources.

4.2 Step II - Modeling Useful Life Management

In Step II, each stage is augmented with a health management controller. In essence, these controllers detect fault precursors that may indicate resource exhaustion vulnerabilities and use this to make prognostic decisions on the health of the system being controlled. Control parameters appropriate for each system are then modulated in order to prolong useful life.

The model in the Figure 12 shows a specific activity augmented with four places. The normal condition place *P_Normal* is active if all resources needed by this system are above pre-determined safe limits. The stressed condition place *P_Stressed* is active if any of the resources needed by this system are below pre-determined safe limits. The controller place *P_Throttle* indicates the condition when the controller is throttling the arrival of tokens to be processed by the place *Act. n*. The place *P_Enable* is the place that determines whether firing of transition *Begin Execute* is enabled or not. As can be seen from the model, the place *P_Throttle* is active only in the stressed operating region.

This step also aims at capturing the implications of controlling the useful life of one activity as it reflects on other stages in the system. This impact on other stages will be determined by the nature of dependencies between activities.

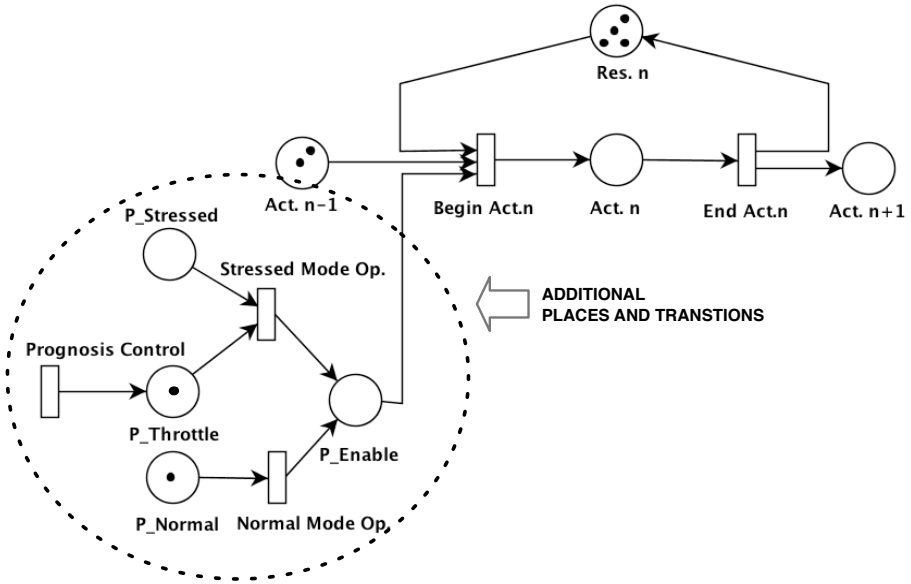


Fig. 12. Partial Petri net model showing places and transitions added in step II augmentation conveying when and how transfer of tasks to activity n are controlled based on system operational condition. Under normal operating condition (token in P_Normal), transition $Begin\ Act.n$ is always enabled under the condition that tokens exist in $Act. n-1$ and $Res. n$. Under stressed operating condition (token in $P_Stressed$) the prognosis controller, throttles rate of tasks moving from $Act. n-1$ to $Act. n$ (indicated by presence or absence of token in $P_Throttle$).

Let us look at an example scenario to describe such an interaction. The model in Figure 13 shows two stages in a queue-based load balancer controlled system. The first stage shows client requests in the queued state $Act. Queue$ and the second stage $Act. Execute$. $Execute$ represents these requests executing on a server. Associated with each stage is the set of n resources that are needed for the activity. $Res. Execute$ would consist of CPU, memory and storage while $Res. Queue$ would consist of parameters such as storage, queue length, etc. The second stage is augmented with a HM controller. The monitoring component of the HM controller will monitor the rate at which resources are being depleted on the server. When the server enters the stressed region, a prognostic decision is taken and the controller will throttle the rate at which client requests are being provided to the server in order to control the rate of resource depletion. In the Petri net model, this is captured by controlling the firing rate of the transition $Begin\ Execute$. This will control the rate of resource consumption of the server and in turn prolong the remaining useful life of this sub system.

An important effect of this controller action is that the number of client requests in the queued stage may build up. In the Petri net model this is reflected

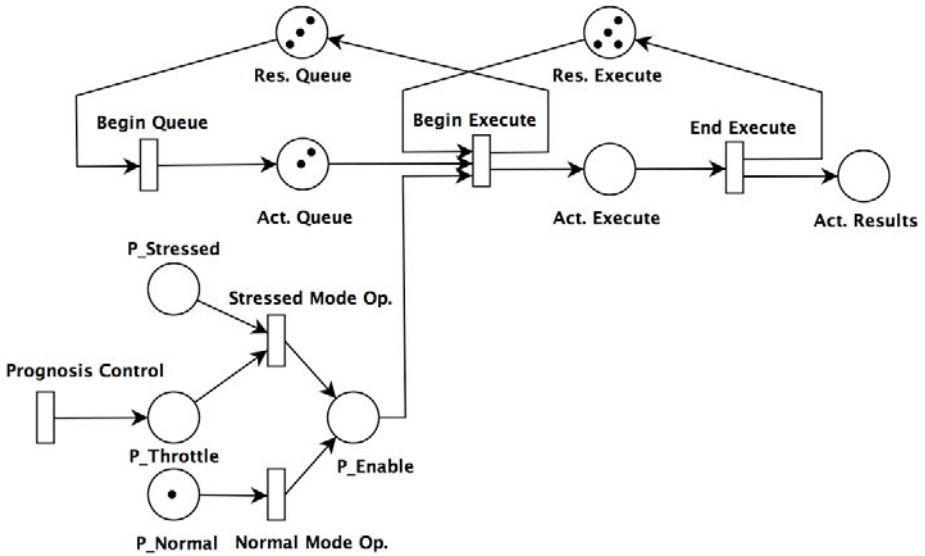


Fig. 13. Partial Petri net model of queued and execution stages of a queue-based system showing interaction between stages. Throttling the firing of transition Begin Execute results in potential accumulation of tokens in place Act. Queue and increased usage of resources in Res. Queue.

by an increase in the number of tokens in the Act. Queue place (A natural consequence of decreased firing rate of transition Begin Execute) as well as an increase in the resource parameter ‘queue length’ in resource Res. Queue. The local HM controller of the activity Act. Queue will take appropriate action when and if this queue length crosses a safe limit. Thus we can see that controlling the rate of consumption of one resource affects the consumption of another resource in a different stage. Coordination between local controllers is needed to ensure smooth operation of the system of systems. We would like to emphasize that the goal of the health management controller is to prolong useful life; and this alone may not prevent the component from failing. So recovery needs to be initiated in the stage under control or in other parts of the system in anticipation of what might happen in the future.

4.3 Step III - Modeling Health Recovery

In step III, the goal is to enable the initiation of health recovery measures. When the system is operating in the stressed region, diagnostic and prognostic results might indicate a need for a health recovery. In this case, possible health recovery actions are evaluated by taking in to the consideration the costs and time overhead involved with the execution of each action as well as the nature of service interruptions that can be tolerated by those activities that utilize the resources involved. The useful life gained by the techniques employed in

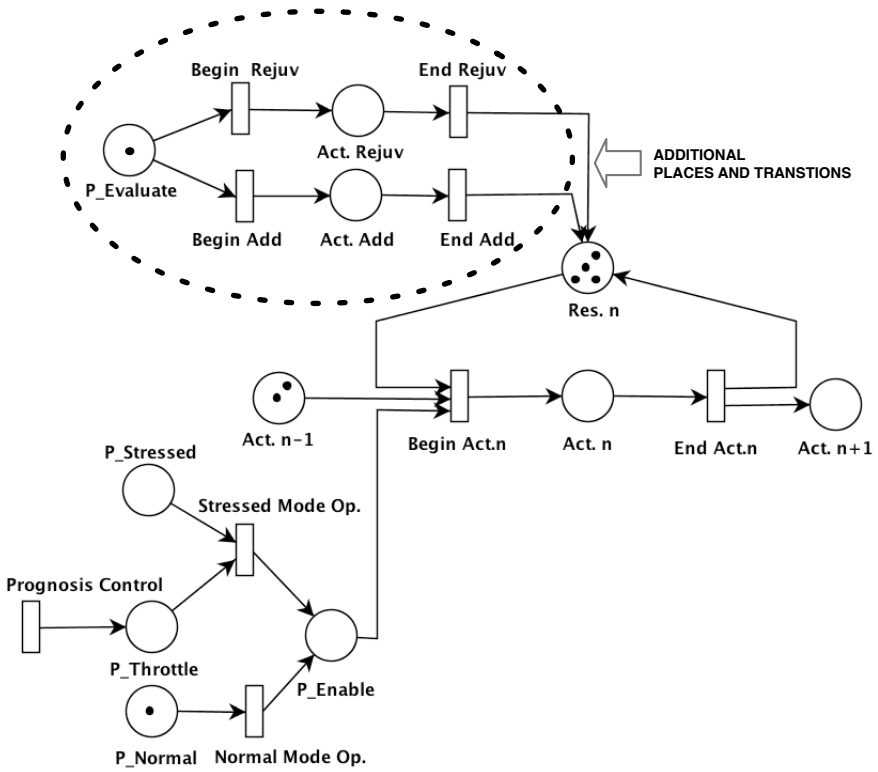


Fig. 14. Partial Petri net model showing places and transitions added in step III augmentation conveying when and how remedial actions are invoked. Increased resources are made available in $Res.n$ in the case of remediation for potential resource exhaustion faults.

the previous step, can be effectively utilized to perform this evaluation, initiate and execute these recovery actions. The goal of these health recovery techniques is to eliminate or mitigate the effects of the observed fault precursors. These techniques can be placed in the broad categories of masking, mitigation, repair, reconfiguration and replacement.

In the previously considered example, a sequence of client requests is being processed by a server. A resource exhaustion failure could occur because of two conditions: 1. too few resources and 2. heavy consumption. The former case could occur because the compute node did not have the appropriate set of resources for its expected load. The latter case could have occurred because of various reasons - load that was unanticipated at design time, malicious usage with the intention of causing a resource exhaustion failure, resource leaks with the system finally running out of resources or a hardware fault affecting resource availability. Health recovery actions to deal with this condition can belong to one of two different classes. Firstly, a failure can be averted or delayed by the addition

of resources to the current server. For example, if the server is running out of disk space, a new storage location could be remotely mounted. In a virtualized environment, the server could be live-migrated to another physical host with more resources of the required type - be it memory, CPU or network bandwidth. Secondly a failure can also be averted by rejuvenation of the server. This is a proven technique that can handle software ageing-related resource-exhaustion failures [23] [24]. The Petri Net model in Figure 14 augments the model from the previous step with 3 new places. The place `P_Evaluate` indicates the cost-benefit analysis performed to compare feasible health-recovery actions. One of the actions `Act. Rejuv` or `Act. Add` is then executed based on the results of the evaluation (other potential actions are not shown for simplicity). Since we are dealing with resource-exhaustion failures, the effect of the execution of either health recovery technique is an increased amount of resources available to the `Act. n` and is captured by the addition of resource tokens to place `Res. n` by places `Act. Rejuv` or `Act. Add`.

4.4 Step IV - Health Management Policies

In this step, policies that determine when and how a remedial action needs to be performed are identified. For example, in the case of resource exhaustion faults, policies determine the resource threshold levels below which RUL management is invoked or when remediation is invoked. These levels can be identified based on the knowledge of a system expert. These policies can also be derived based on simulation and analytical studies of a production system. Since Petri net models allow for both simulation and analysis, these can be used to determine average resource consumption estimates for typical workloads. Significant deviations from these expected resource consumption values can then be used as potential indicators of anomalous conditions.

The above four steps are used to systematically model a given IT system and then augment it with health management capabilities. The final model will then serve as the basis to build the global system manager. The functionality associated with each Petri net place is converted to code that is then input to a generic Petri net execution engine.

Proof-Of-Concept Implementation. The modeling methodology described in this section and RUL management technique described in Section 2 have been applied to a batch-based job submission system in a virtualized environment. A Petri net model was designed and implemented to capture properties of interest and to serve as a global system manager of the job submission system. We show that our RUL extension technique effectively prevents job failures due to memory resource-exhaustion while incurring low overhead. The reader is referred to [15] for details of this implementation.

5 Related Work

Our goals are mostly similar to the work in [16] [25] [26], in which the authors propose the use of pattern recognition, multivariate state estimation and Markov

chain based modeling techniques for prognostics in different components (eg. main memory module, power supply) of a compute server. Though our goals are similar, the contribution of our work is a systematic framework and methodology for health management at the system of systems level. At the subsystem level, we use a control theoretic formulation for useful life management.

Model Driven Approach to Autonomic Systems - In [27] [28] Dobson has shown the need and requirements of models to capture self-* properties in autonomic systems. Our work has been influenced by Sven Graupner's IT management controller [29] that uses High Level Petri Nets to model IT tasks. Our work differs in that the scope of our work includes health management of system level models, rather than error recovery in individual IT components. Petri nets have also been used by Salfner et al [30] to determine service availability improvement achieved through the use of redundant servers. Dai et al. [31] have proposed a model-driven approach to autonomic computing. But this proposal has no specifics and has not been applied to a real world autonomic solution. Bellur's work [32] briefly outlines the use of Hierarchical Queuing Petri nets (HQP) with the ultimate goal of modeling autonomic computing properties. This project's most recent status was the modeling of a Tomcat server using HQPN. The field of workflows has used Petri nets extensively by extending the basic Petri net model into Workflow nets [2] [33]. Domain specific modeling languages have been used in to model autonomic systems. Dubey et al. [35] have used timed automaton models to verify timing behavior of autonomic systems. Architecture models have been used for capturing autonomic properties in IT systems in [36].

Failure, Monitoring, Detection and Prediction Studies - Though the following list of works mostly deal with failures rather than health (which is our focus), techniques from these fields can be leveraged in health management. Salfner et. al [37] have provided a comprehensive survey of online failure prediction in computing systems, principles of which form a part of prognosis. Tiresias [38] is a black box approach to failure prediction in distributed systems using performance metrics. OVIS [39] [40] is a tool for statistical modeling and analysis of monitored data in computational clusters, with the goal of determining advance indicators of failures. In [41], Ren et al. use semi-Markov processes to predict resource failures in fine-grained cycle sharing systems. In [42], Laguna et al. have developed an error detection system for multi-tier applications using a Hidden Markov Model (HMM) based algorithm that is capable of handling the high volume of messages in a distributed system (thereby providing efficient monitoring). Schroeder et al. [43] have studied failures in large-scale systems with conclusions on failure rate, repair time and time between failure distributions. Bayesian estimation and Markov decision theory has been used by Joshi et al. in [44] to choose optimal recovery actions in a distributed system. Large-scale studies of failures in HPC and petascale systems in [45] [46] [47] [48] motivate the importance of health management in these infrastructures.

Resource Exhaustion Studies - These works characterize resource exhaustion faults, while our goal is to use this category of faults as an example for our proposed health management solution. In [7] semi-Markov reward models are used to estimate resource exhaustion rates as a function of workload. In [49] Antunes et al. have proposed a technique to predict resource exhaustion vulnerabilities in both synthetic as well real-world DNS servers using resource usage modeling and a fault injection framework.

Control Theoretic Approaches - The application of feedback control theory to computing systems has been introduced in [50]. The control of memory and CPU utilization in web servers has been studied in [51] [52].

6 Future Work and Conclusions

The goal of our work has been to bring out the potential in the application of health management principles to IT systems of systems. Towards this goal we have proposed the use of a modeling framework and methodology to incorporate health management in an IT system. We also propose the use of control theory to manage useful life extension of subsystems in the case of a possible resource exhaustion fault. As a simple illustration of the concept, a controller was built for useful life management in the application execution stage (containing a potential memory exhaustion fault) of an IT system. The gained useful time can then be effectively utilized to evaluate and initiate health remediation actions. As part of our ongoing work, in order to show the applicability, benefits and ease of use of our methodology in a real-world IT system, we have created a prototype global manager for a batch-based job submission HPC system on a virtualized platform. The simple proportional integral controller discussed in this work is being developed in to auto-tuning and self-tuning versions to allow for online parameter estimation and controller design.

Acknowledgements. This work is supported in part by the National Science Foundation under Grants No. CNS-0821622 and IIP-0758596. The authors also acknowledge the support of the BellSouth Foundation. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or BellSouth Foundation.

References

1. Murch, R.: *Autonomic Computing*. IBM Press (2004)
2. Marinescu, D.C.: *Internet Based Workflow Management: Towards a Semantic Web*. Wiley Interscience, Hoboken (2002)
3. Stewart, C., Shen, K.: Performance modeling and system management for multi-component online services. In: 2nd Conference on Symposium on Networked Systems Design and Implementation (2005)

4. Conallen, J.: Modeling Web application architectures with UML. *Communications of the ACM* (1999)
5. Van der Mei, R.D., Hariharan, R., Reeser, P.: *Web Server Performance Modeling. Telecommunication Systems* (2001)
6. Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., Tantawi, A.: An analytical model for multi-tier internet services and its applications. In: *ACM SIGMETRICS* (2005)
7. Vaidyanathan, K., Trivedi, K.S.: A Measurement-Based Model for Estimation of Resource Exhaustion in Operational Software Systems. In: *10th International Symposium on Software Reliability Engineering* (1999)
8. Kahkipuro, P.: *UML-Based Performance Modeling Framework for Component-Based Distributed Systems. LNCS* (2001)
9. Zhou, M., Venkatesh, K.: *Modeling, Simulation and Control of Flexible Manufacturing Systems A Petri net Approach. World Scientific, Singapore* (1999)
10. Vachtsevanos, G., Lewis, F.L., Roemer, M., Hess, A., Wu, B.: *Intelligent Fault Diagnosis and Prognosis for Engineering Systems. Wiley, John and Sons, Chichester* (2006)
11. Tang, L., Kacprzynski, G.J., Goebel, K., Saxena, A., Saha, B., Vachtsevanos, G.: Prognostics-Enhanced Automated Contingency Management for Advanced Autonomous Systems. In: *Ist International Conference on Prognostics and Health Management (PHM 2008), Denver, CO* (2008)
12. Engel, S.J., Gilmartin, B.J., Bongort, K., Hess, A.: Prognostics, The Real Issues Involved with Predicting Life Remaining. In: *IEEE Aerospace Conference* (2000)
13. Kalgren, P.W., Baybutt, M., Ginart, A., Minnella, C., Roemer, M.J., Dabney, T.: Application of prognostic health management in digital electronic systems. In: *IEEE Aerospace Conference*, pp. 1–9 (March 2007)
14. Michael, J.R., Kacprzynski, G.J., Nwadiogbu, E.O., Bloor, G.: Development of Diagnostic and Prognostic Technologies for Aerospace Health Management Applications. In: *IEEE Aerospace Conference, Big Sky, MT*, pp. 3139–3147 (2001)
15. Kadirvel, S., Fortes, J.A.B.: Self-Caring IT Systems - A Proof-of-Concept Implementation in Virtualized Environments. In: *International Conference on Cloud Computing Technology and Science (CloudCom), Indianapolis, USA* (2010)
16. Urmanov, A.: Electronic Prognostics for Computer Servers. In: *Proceedings of 53rd Annual Reliability and Maintainability Symposium (RAMS), Orlando, Florida*, pp. 65–70 (2007)
17. Pecht, M., Jaai, R.: A prognostics and health management roadmap for information and electronics-rich systems. *Microelectronics Reliability* 50(3), 317–323 (2010)
18. CWE-400: Uncontrolled Resource Consumption. Common Weakness Enumeration. An initiative sponsored by the National Cyber Security Division of the U.S. Department of Homeland Security. <http://cwe.mitre.org/data/definitions/400.html> (accessed: March 16, 2010)
19. Zhou, M., Dicesare, F.: *Petri Net Synthesis for Discrete Event Control of Manufacturing Systems. Kluwer Publishers, Dordrecht* (1993)
20. Jensen, K., Kristensen, L.M., Wells, L.: Coloured Petri Nets and CPN Tools for modeling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer, STTT* (2007)
21. Marsan, A.: Stochastic Petri nets: An elementary Introduction. In: Rozenberg, G. (ed.) *APN 1989. LNCS, vol. 424*, pp. 1–29. Springer, Heidelberg (1990)
22. Muppala, J., Ciardo, G., Trivedi, K.S.: Stochastic Reward Nets for Reliability Prediction. In: *Communications in Reliability, Maintainability and Serviceability* (1994)

23. Kolettis, N., Fulton, N.D.: Software Rejuvenation: Analysis, Module and Applications. In: 25th International Symposium on Fault-Tolerant Computing (1995)
24. Vaidyanathan, K., Trivedi, K.S.: A Comprehensive Model for Software Rejuvenation. *IEEE Transactions Dependable and Secure Computing* (2005)
25. Gross, K.C., McMaster, S., Porter, A., Urmanov, A., Votta, L.G., Langer, Y., Urmanov, A.: System's Availability Maximization Through Preventive Rejuvenation. Sun Microsystems, USA (2006)
26. Hamerly, G., Elkan, C.: Bayesian approaches to failure prediction for disk drives. In: 18th International Conference on Machine Learning, pp. 1–9 (2001)
27. Dobson, S.: Facilitating a well-founded approach to autonomic systems. In: 5th IEEE Workshop on the Engineering of Autonomic and Autonomous Systems, Belfast, UK (2008)
28. Dobson, S.: Achieving an acceptable design model for autonomic systems. In: 4th IEEE International Workshop on Engineering Autonomic and Autonomous Systems Tucson, AZ, pp. 196–202 (2007)
29. Graupner, S., Cook, N., Coleman, D.: Automation Controller for Operational IT Management. *Integrated Network Management*, 363–372 (2007)
30. Salfner, F., Wolter, K.: A Petri net model for service availability in redundant computing systems. In: Winter Simulation Conference (2009)
31. Dai, Y.S., Marshall, T., Guan, X.H.: Autonomic and Dependable Computing: Moving Towards a Model-Driven Approach. *Journal of Computer Science* (2006)
32. Bellur, U.: Automating Applications Management in the Enterprise using DMTF Information Models. Indian Institute of Technology, Bombay, www.dmtf.org/education/academicalliance (accessed: March 16, 2010)
33. Van der Aalst, W.M.P., Van Hee, K.M.: Business Process Redesign A Petri net based approach. *Computers in Industry* (1996)
34. Shetty, S., Nordstrom, S., Ahuja, S., Yao, D., Bapty, T., Neema, S.: Systems Integration of Large Scale Autonomic Systems Using Multiple Domain Specific Modeling Languages. In: 12th IEEE International Conference and Workshops on Engineering of Computer-Based Systems, Washington DC (2005)
35. Dubey, A., Nordstrom, S., Keskinpala, T., Neema, S., Bapty, T.: Verifying Autonomic Fault Mitigation Strategies in Large Scale Real-Time Systems. In: Third IEEE international Workshop on Engineering of Autonomic and Autonomous Systems, Washington DC (2006)
36. Garlan, D., Schmerl, B., Cheng, S.: Software Architecture-Based Self-Adaptation. *Autonomic Computing and Networking Part 1*, 31–55 (2009)
37. Salfner, F., Lenk, M., Malek, M.: A Survey of Online Failure Prediction Methods. *ACM Comput. Surv.* 42(3), Article 10 (2010)
38. Williams, A.W., Pertet, S.M., Narasimhan, P.: Tiresias: Black-Box Failure Prediction in Distributed Systems. In: 21st International Parallel and Distributed Processing Symposium (IPDPS), California, USA (2007)
39. Brandt, J., Gentile, A., Mayo, J., Pbay, P., Roe, D., Thompson, D., Wong, M.: Methodologies for Advance Warning of Compute Cluster Problems via Statistical Analysis: A Case Study. In: Workshop on Resiliency in High Performance Computing (HPDC), Munich, Germany (2009)
40. Brandt, J., Debusschere, B., Gentile, A., Mayo, J., Pbay, P., Thompson, D., Wong, M.: Using Probabilistic Characterization to Reduce Runtime Faults on HPC Systems. In: Workshop on Resiliency in High-Performance Computing (CCGRID), Lyon, France (2008)

41. Ren, X., Lee, S., Eigenmann, R., Bagchi, S.: Resource Failure Prediction in Fine-Grained Cycle Sharing Systems. In: 15th IEEE International Symposium on High Performance Distributed Computing (HPDC-15), France (2006)
42. Laguna, I., Arshad, F.A., Grothe, D.M., Bagchi, S.: How To Keep Your Head Above Water While Detecting Errors. In: ACM/IFIP/USENIX 10th International Middleware Conference, Illinois (2009)
43. Schroeder, B., Gibson, G.A.: A large-scale study of failures in high-performance computing systems. In: International Conference on Dependable Systems and Networks (2006)
44. Joshi, K.R., Sanders, W.H., Hiltunen, M.A., Schlichting, R.D.: Automatic Model-Driven Recovery in Distributed Systems. In: 24th IEEE Symposium on Reliable Distributed Systems (2005)
45. Gibson, G.A., Schroeder, B., Digney, J.: Failure Tolerance in Petascale Computers. *CTWatch Quarterly* 3(4), Volume on Software Enabling Technologies for Petascale Science (2007)
46. Schroeder, B., Gibson, G.A.: Understanding Failures in Petascale Computers. In: *SciDAC 2007. Journal of Physics: Conference Series*, vol. 78 (2007)
47. Schroeder, B., Gibson, G.A.: Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In: 5th USENIX Conference on File and Storage Technologies, San Jose, CA (2007)
48. Schroeder, B., Gibson, G.: A Large Scale Study of Failures in High-performance-computing Systems. In: International Symposium on Dependable Systems and Networks (2006)
49. Antunes, J., Neves, N.F., Veríssimo, P.J.: Detection and Prediction of Resource-Exhaustion Vulnerabilities. In: 19th International Symposium on Software Reliability Engineering, pp. 87–96 (2008)
50. Hellerstein, J.L., Diao, Y., Parekh, S., Tilbury, D.M.: *Feedback Control of Computing Systems*. John Wiley and Sons, Chichester (2004)
51. Gandhi, N., Tilbury, D.M., Diao, Y., Hellerstein, J., Parekh, S.: MIMO control of an Apache Web Server: Modeling and Controller Design. In: American Control Conference, Ann Arbor, Michigan (2002)
52. Diao, Y., Hu, X., Tantawi, A., Wu, H.: An adaptive feedback controller for SIP server memory overload protection. In: 6th International Conference on Autonomic Computing, Barcelona, Spain (2009)
53. Peterson, J.L.: *Petri Net Theory and The Modeling of Systems*. Prentice-Hall, New Jersey (1981)
54. Bonet, P., Llado, C.M., Puijaner, R., Knottenbelt, W.J.: PIPE2.5 - A Petri net tool for performance modeling. In: Proc. 23rd Latin American Conference on Informatics, San Jose, Costa Rica (2007)

Appendix

Benefits of Petri Nets

Petri nets are a mathematical as well as graphical tool for modeling DES. A general introduction to Petri net modeling can be found in [53]. The advantages of using Petri nets include:

- 1) System validation - Various system properties can be determined to validate the correctness of the system.
- 2) System monitoring - When interfaced with a production system, Petri nets allow for real time and visual monitoring of the system status.
- 3) System simulation - A generic Petri-net execution engine can be used for simulating different classes of Petri nets. This is also called a token game because a simulation involves changing the state of the system by moving tokens between places.
- 4) System performance evaluation - The base Petri net model can be augmented with time to enable measurement of the various performance metrics of systems.
- 5) System composition - In the case of a large system, it would be necessary for different experts to model various parts of the system. Petri nets possess the composition property such that under certain predefined conditions, it is guaranteed that system properties are maintained when independent models are combined together to a single one [20] [22].
- 6) System control - Petri nets provide the ability to generate a supervisory controller directly from the system model.

Petri Net Properties of Interest

The following is a listing of certain Petri net properties and how they apply to the modeling of an IT system.

1) Reachability: In a Petri net C , a marking μ' is reachable from marking μ , if there exists a sequence of transition firings that can transform the state from μ to μ' . The reachability set $R(C, \mu)$ is the set of all reachable markings from marking μ . In a system model certain markings can represent the system operating in a faulty mode. In such a situation, it is useful to know if the system can reach (or be steered to reach) a non-faulty marking.

2) Boundedness: A place p_i of a Petri net with an initial marking μ is k -bounded if for all $\mu' \in R(C, \mu)$, $\mu'(p_i) \leq k$. A Petri net is bounded if all its places are bounded. If $k=1$, then the Petri net is said to be safe. In a system model, boundedness can represent the capacity of a resource. If access to a data center is through a job queue, then the bound on the place representing the queue can be used to ensure that the queue neither overflows nor is underutilized.

3) Liveness: A Petri net is live with respect to a marking m_0 , if for any marking in its reachability set $R(m_0)$, it is possible to ultimately fire any transition in the net. Liveness guarantees the absence of deadlocks [11]. In a system model

this would mean that a system could continue to operate without getting stuck in a deadlock situation arising due to resource dependencies.

4) Reversibility: A Petri net is reversible iff for each reachable marking $m \in R(m_0)$, the initial marking m_0 belongs to the reachability set of m . i.e. $m_0 \in R(m)$. A system that possesses this property will be able reinitialize itself if a failure occurs at any stage.

Other properties of Petri nets can be analyzed by using a reachability tree or by matrix equation techniques. All the Petri net models in this work were constructed using the PIPE tool [54].

Self-reconfigurable Modular Robots and Their Symbolic Configuration Space

Souheib Baair¹, Lom-Messan Hillah¹, Fabrice Kordon², and Etienne Renault²

¹ LIP6, CNRS UMR 7606 and Université Paris Ouest Nanterre La Défense
200, avenue de la République, F-92001 Nanterre CEDEX, France
Lom-Messan.Hillah@lip6.fr, Souheib.Baair@lip6.fr

² LIP6, CNRS UMR 7606, Université P. & M. Curie - Paris 6
4, place Jussieu, F-75252 Paris CEDEX 05, France
Fabrice.Kordon@lip6.fr, Etienne.Renault@gmail.com

Abstract. Modular and self-reconfigurable robots are a powerful way to design versatile systems that can adapt themselves to different physical environment conditions. Self-reconfiguration is not an easy task since there are numerous possibilities of module organization. Moreover, some module organizations are equivalent one to another.

In this paper, we apply *symbolic* representation techniques from model checking to provide an optimized representation of all configurations for a modular robot. The proposed approach captures symmetries of the system and avoids storing all the equivalences generated by permuting modules, for a given configuration. From this representation, we can generate a compact *symbolic configuration space* and use it to efficiently compute the moves required for self-reconfiguration (*i.e.* going from one configuration to another). A prototype implementation is used to provide some benchmarks showing promising results.

Keywords: Modular robotics, Self-reconfiguration, Symbolic configuration space, Symmetries, CKBot.

1 Introduction

Context. Modular robotics is an active research field where robots are assembled using numerous identical or different types of small modules. This is a powerful way to design versatile systems that can adapt themselves to different physical environment conditions or according to the purpose of their mission [9]. Moreover, self-reconfiguration allows to adjust the robot to a given task on the fly. Modular robots are thus perceived as a means to reach a balanced compromise between realization cost and multitasking capabilities. They are particularly well-suited to exploration (*e.g.* spatial) and search and rescue missions in hostile environments. Finally, they are robust and cheaper to produce.

Problem. If auto-reconfiguration is a way to change the shape of a robot made of modules, it is also a way to create complex movements by means of successive configuration changes. The robot follows a path of configurations that allows it to move, to grab and drop objects, etc.

This is a key feature whose implementation faces numerous issues [12]:

- memory limitation, computation power and energy consumption,
- limited degrees of freedom, sometimes more constrained because some types of modules are more constrained than the others,
- coordinated communication between modules and inter-module communication schemes,
- structural symmetries in modules, generating equivalent modules configurations.

The last issue raises a problem for the computation of the *configuration space* for a robot composed of N modules. The size of the configuration space increases exponentially with N (where $N_i, i \in [1..T]$ when the robot is composed of T types of modules). Both the generation of the full configuration space and the identification of a given configuration in it are known problems [8].

Contribution. The objective of this paper is to tackle the combinatorial explosion problem in the configuration space of modular robots. Our solution also helps to identify a given configuration among the ones that are equivalent. Self-reconfigurable modular robots are usually classified in two categories.

First, lattice-based self-reconfigurable robots can physically organize themselves in 2D or 3D grid structures. They are rigidly interconnected but are able to connect/disconnect and move relative to one another in a 2D or 3D space. In this kind of configuration, modules can only connect to their adjacent neighbors. Connection is assumed to be performed without alignment, because modules are assumed to be always aligned. This may not be true in practice for large configurations. The ATRON [3] is a typical example of a 3D lattice-based self-reconfigurable robot.

Second, chain-based robots can assemble in serial chains fashion (linear loops connections/disconnections) aligning themselves for connecting. They can form flexible configurations and are efficient for locomotion, since they can bend themselves in arbitrary angles to move. For instance, a snake-shaped robot can move like a snake because its modules bend in coordination to perform this kind of movement. The PolyBot [13] is a typical example of a chain-based robot.

This study focuses on CKBot [10]: a hybrid robot whose modules allow both chain and lattice reconfiguration capabilities [12].

The CKBot has two types of module: the UBar and the L7. Our study focuses on the UBar, whose picture you can find online at [10]. We only consider robots made of one type of module but this work can easily be extended to the case where several types of modules are involved.

Symmetries are a serious issue in the original explicit approaches to generate the CKBot configurations, when the number of modules grows. For instance, the methods presented in [8] take a lot of time for disambiguation because of the symmetries between numerous similar configurations. Our purpose is to propose a new approach for modeling the system, using symbolic representation techniques where symmetries are handled efficiently.

Our contribution lies in the following points. First, we propose an efficient symbolic representation of the modular robot configurations. It represents, by means of a

single matrix, both connections and orientations of modules. It is a significant improvement of the proposal made in [8] where two matrices are involved.

Then, we optimize the symbolic representation, where similar connection schemes of some inter-dependent connectors are only represented once in a symbolic way. From this symbolic representation, explicit configurations may be generated. Similarly to model checking, we produce the configuration space of the system as an oriented graph where nodes represent a set of equivalent configurations and arcs represent an action of any module in the system. This reduced graph replaces the traditional plain graph approach.

Therefore, we can exploit this configuration space to compute the moves that lead from a configuration c to any configuration $c' \in C$, the set of target equivalent configurations. This is a classical path search in an oriented graph.

Content. Section 2 presents the CKBot UBar module and the two-matrix based representation technique of the robot configuration originally presented in [8]. We also describe the core principles of the symmetry techniques applied in this paper. Then, section 3 proposes an alternative to the robot description of [8] and section 4 explains how we turn this new explicit representation into a symbolic one. Section 5 deals with the reconfiguration computation issue, where the transition system of the robot successive configurations is built and used to find paths between configurations. Finally, section 6 presents performance evaluation provided by a prototype implementation before a conclusion in section 7.

2 Problem Statement and Related Works

Controlling the configuration of modular and self-reconfigurable robots is computationally complex. This complexity depends on how the system is organized, both at the hardware and software levels. We consider the CKBot, where both a global bus and neighbor-to-neighbor communication schemes enable the system to determine its configuration.

In the CKBot, connections between modules are represented by graphs, translated into adjacency matrices and ports adjacency matrices. Modules interconnect via connectors and are identified by values from 1 to N , where N is the number of modules¹. Adjacency matrices are $N \times N$ matrices in which 1s denote interconnections between modules and 0s the absence of connection. In port adjacency matrices (which also are $N \times N$ matrices), non-zero numbers identify the IDs of the ports through which modules interconnect. Fig. 3 shows a CKBot assembly and its adjacency matrices.

We first present the CKbot module, then an explicit way to encode its configuration and the symmetry-based methods used in model checking to tackle combinatorial explosion. Section 3 applies this technique to efficiently encode the configuration space.

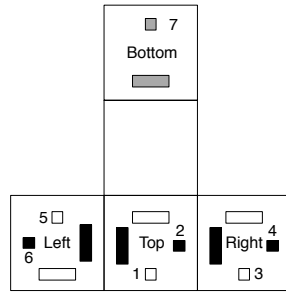
¹ Here, we only consider a system with one type of module. For a system with T types of module, we can consider the identities being $1, \dots, N_1, N_1 + 1, \dots, N_2, \dots, N_{T-1}, \dots, N_T$ where N_i is associated to the i^{th} type of module, $i \in [1, T]$.

2.1 Presentation of the CKBot

Fig. 1(a) presents a picture of a CKBot UBar module² and Fig. 1(b) shows its corresponding 2D schema representation. The CKBot UBar module has 7 ports (pair of infrared transmitters/receivers) and 20-pin headers on each of its 4 faces. Ports, represented by a square, are used for inter-module communication. 20-pin headers, represented by a rectangle, are used for electrical connection and communication on a CAN (Controller Area Network) bus. In the latter part of this paper, we may simply refer to couples ⟨port, 20-pin header⟩ as *connectors* (e.g. the Bottom face holds only one connector).



(a) A CKBot UBar module



(b) Its corresponding 2D structure

Fig. 1. Overview of the CKbot

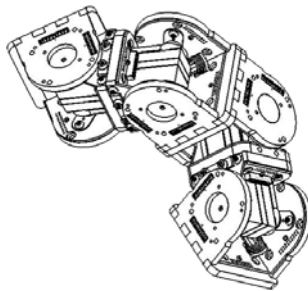


Fig. 2. A 5-module T shape robot and its corresponding adjacency matrices

Top and Right faces share the same disposition for their connectors, whereas the one of Left is reversed (port 5). Thus, a module can attach to another in different ways. According to [12], each module can be *uniquely connected to another module in 10 ways* (3 rotations for each of the 3 top faces and 1 orientation for the bottom). The only impossible connection is when two same faces are in front of one another in reverse positions. Fig. 2 presents an example of modular robot built from four CKBot modules³.

² This picture is extracted from [8].

³ This picture is extracted from [1].

2.2 Explicit Encoding of CKBot Configurations (from [8])

The technique used to describe the robot configuration is based on two $N \times N$ matrices, N being the number of modules⁴. The first matrix is a simple adjacency matrix, where 1-entries denote a connection between two modules and 0-entries no connection. The second matrix is a port adjacency matrix, where non-zero entries denote the type of connection from a module to another (referencing the port number). Fig. 3 shows a 5-module T shape and its corresponding adjacency matrices on the right.

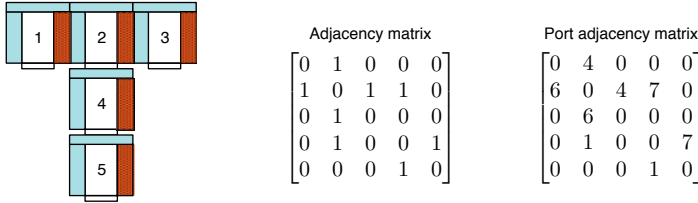


Fig. 3. A 5-module T shape robot and its corresponding adjacency matrices

This representation is suitable for configuration recognition on small configurations. However, it does not scale well. In particular, when reconfiguration is the problem under consideration, searching the configuration space for N modules which can move in parallel, quickly leads to combinatorial explosion of the configuration space for large values of N .

In [8], the author deals with automatic configuration recognition based on three principles:

- graph-based isomorphism identification: this method suffers from the exponential size of the automorphism group in the number of modules (worst case), as the size of the library of predefined configurations grows;
- port adjacency matrix spectral decomposition: it is very fast for small numbers of modules but suffers from numerical issues when numerous modules are involved. Explicit disambiguation due to symmetries in the configurations can be very long;
- heuristic-based linked list (called 3DLL) representation of the physical properties of configurations: it exploits the ports adjacency matrix of the modules. This method appears to be the most scalable, but suffers from the need to run exhaustively through every configuration in the library.

In order to tackle the combinatorial explosion in the configuration space, we propose to get inspiration from formal analysis methods [11] where this problem is common. Different and complementary techniques are used to reduce the size of the state space: decomposition, bounding, partial order, symmetry detection and the use of very efficient data structures (Decision Diagrams).

⁴ In the remainder of this paper, N will always denote the number of modules that compose the complete robot.

It appears that the configurations of the CKBot can be organized into consistent sets of similar configurations where they only vary by module permutations. Hence, a more compact representation for the model can be designed to remove redundant and explicit information which can be inferred otherwise. In this setting, symmetry-based techniques are suitable to build their representation.

2.3 Compact Representation of Large State Spaces

This section presents through an example the principles of the symmetry-based techniques underlying compact representation of large state spaces in model checking. Formal definitions of the underlying theory can be found in [4].

Symmetry-based methods, exploit the presence of similarly behaving components to aggregate states (or, in our case, configurations) and state transitions (or, in our case, configurations changes) into equivalence classes. Hence, they generate a more abstract and compact state space: the *quotient graph*.

To present the quotient graph in the general framework, let us consider the classical example of a client/server system, with two identical clients C_1 and C_2 and a server S . Clients build a message $m \in \{m_1, m_2\}$, send it to S with their identity, and wait for an acknowledgment message. S processes incoming messages and then sends the acknowledgment to the client having issued the request.

We consider two local states for a client: (1) the message construction state and, (2) the receiving state. For the server, we consider: (1) the receiving state and (2) the sending of the acknowledgment. We also consider the network state $R: R(\langle C_1, m_1 \rangle)$ means that message m_1 of client C_1 is passing through the network R . Thus, the global state of our system will be the synthesis of all local states.

The behavior of the system can then be represented by a *reachability graph*, where nodes are global states, and arcs represent changes between states. Figure 4 represents the beginning of the reachability graph of our toy example (the whole graph contains 24 states).

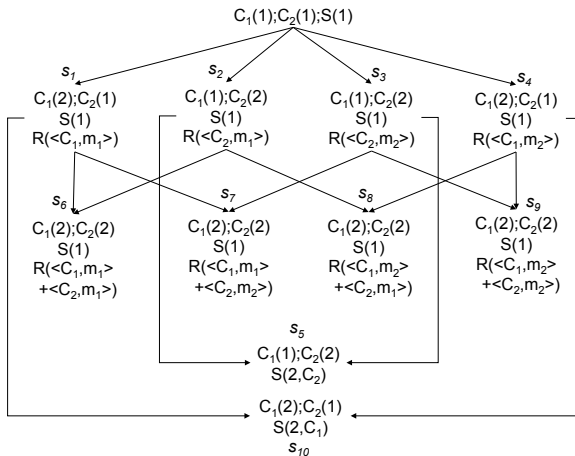


Fig. 4. First 11 states (among 24) of the reachability graph of the client/server example

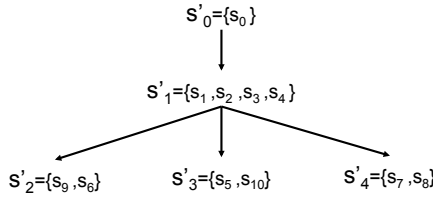


Fig. 5. First 5 states (among 10) of the quotient graph of the client/server example, w.r.t equivalence relation \mathfrak{R}

As shown in Fig. 4 the state space grows quickly with the number of clients and the type of messages. Exploitation of symmetries in the system helps to tackle this combinatorial explosion. We observe that C_1 and C_2 behave identically, hence they are symmetrical. Similarly, messages values are not distinguished by the server (*i.e.* they are processed identically), introducing another symmetry.

Let us formally identify these symmetries by an equivalence relation $\mathfrak{R} = \{C = \{C_1, C_2\}, M = \{m_1, m_2\}\}$, where C and M are equivalent classes. \mathfrak{R} can then be used to build a quotient graph that preserves reachability and some temporal logic properties of the original reachability graph.

According to \mathfrak{R} , states of the reachability graph are partitioned in five equivalence classes: $s'_0 = \{s_0\}$, $s'_1 = \{s_1, s_2, s_3, s_4\}$, $s'_2 = \{s_5, s_7\}$, $s'_3 = \{s_5, s_{10}\}$ and $s'_4 = \{s_7, s_8\}$. The quotient graph corresponding to the 11 states of the reachability graph presented in Fig. 4 is represented in Fig. 5. Let us note that, in this case, its size neither depends on the number of clients nor the number of values for messages.

In the next section, we apply this technique to build all the configurations of a robot made with CKBot modules. To do so, there are three issues to deal with:

- identifying symmetries in the configurations in terms of permutations,
- elaborating an efficient symbolic representation for the equivalent classes generated by these permutations,
- building the symbolic transition relation.

3 Representing CKBot States

The first technique to fight against combinatorial explosion in the configuration space is to design a compact and efficient representation to model the robot. An important requirement for this compact representation is that it must preserve all the important information that cannot be computed from existing ones.

3.1 Matrix Representation

Our new model is also stored in a matrix, where 7 columns encode the ports connectivity of a module⁵ and an 8th column encodes its angle. Figure 6 shows the definition

⁵ Numerotation of ports (*e.g.* *top1*, *top2*, etc.) refers to the 2D flat representation of a CKBot module as presented in Fig. 1(b).

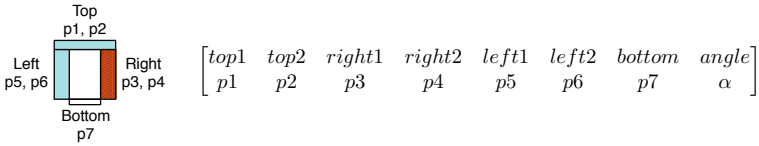


Fig. 6. Ports-connectivity matrix of an UBar module



Fig. 7. Angles representation for a UBar module

of this representation. The configuration of a module is thus encoded in a vector of size *Number of Ports + Degrees of Freedom*. The size of a configuration involving N modules is then $N \times (\text{Number of Ports} + \text{Degrees of Freedom})$. A CKBot module has 1 degree of freedom, hence a single column is sufficient to encode it.

In the ports-connectivity matrix, non-zero entries denote a connection of the corresponding port (column) to a module whose identity is the value of the entry. As for the explicit representation (section 2.2), identities of modules range from 1 to N , so that no ambiguity is raised between an absence of connection and a module identity.

The angle scale ranges from 0 to 180° with a graduation in D° increments. To determine the physical position of a module from its representation as described in Fig. 6, angle is set to zero when the face *Bottom* is positioned such that ports 4, 6 and 7 are aligned. It is illustrated in Fig. 7 with *Bottom* oriented to the right. Angle 180 is determined when the module has made a rotation such that *Bottom* is oriented to the left.

Example. Let us determine the ports-connectivity matrix for the 5-module T shape example shown in Fig. 3. The matrix encoding this configuration is shown in Fig. 8. It has the fixed 8 columns for the UBar and 5 lines for the number of modules. We can expect this type of representation to be much more compact than the adjacency matrices, where the number of modules will be in most cases greater than the sum of the numbers of ports and degrees of freedom.

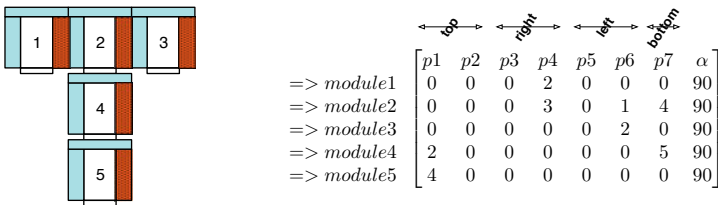


Fig. 8. Ports-connectivity matrix of the 5-module T shape of Fig. 3

3.2 Matrix Encoding

An alphabet can be set up to encode all possible configurations of a module. Since a port can be connected or not, we need two values per port (for 7 ports). Thus, a 7-bit alphabet can encode this.

We set up an alphabet to encode all possible configurations for a module. A port being either connected or not and since we have seven ports, we elaborate a compact alphabet to encode connectivity. Therefore, each letter in the alphabet is a concatenation of two parts:

- a first part contains 7 bits describing the connectivity of the described module, encoded as: $p1 \times 2^7 + p2 \times 2^6 + p3 \times 2^5 + p4 \times 2^4 + p3 \times 2^2 + p2 \times 2^1 + p1 \times 2^0$. Therefore, each configuration is represented in a unique way;
- a second part specifies the angle of the described module. There are x configurations per connectivity, where x is deduced from D ($x = 3$ when $D = 90^\circ$, $x = 4$ when $D = 45^\circ$, etc).

p1	p2	p3	letter	p1	p2	p3	letter
0	0	0	$X_{0,\alpha}$	1	0	0	$X_{4,\alpha}$
0	0	1	$X_{1,\alpha}$	1	0	1	$X_{5,\alpha}$
0	1	0	$X_{2,\alpha}$	1	1	0	$X_{6,\alpha}$
0	1	1	$X_{3,\alpha}$	1	1	1	$X_{7,\alpha}$

Fig. 9. Alphabet encoding the connectivity of a 3-port module. α represents the angle

Example. Let us consider a simple case with a module having only three ports. Figure 9 illustrates this encoding. There are $2^3 \times (\frac{180}{D} + 1)$ values in the alphabet. There is a total order since $\forall i \in [0, 2^3]$, $X_{i,\alpha} \leq X_{i,\beta}$ iff $\alpha \leq \beta$. For the CKBot, we thus use the following formula to compute the number of letters in the alphabet:

$$2^7 \times \left(\frac{180}{D} + 1 \right) \quad (1)$$

4 Symbolic Representation of the CKBot Configuration Space

This section applies the symmetry technique described in section 2.3 to the representation presented in section 3. First, we identify equivalent configurations obtained from module rotation, then we discuss isomorphic configurations obtained by module permutation. We show how the two matrices can be combined and finally present a canonical way to express equivalence classes considering these two types of symmetries. This leads to the notion of *symbolic configuration space*.

4.1 Identification of Structural Symmetries

Module Rotation. In the UBar CKBot module, there exists a symmetry between faces *Right* and *Left*. When this module is rotated with 180° in the orientation *Right-Left*

(or in the reverse way), we obtain a mirror configuration. Ports 4 and 6 are symmetric, as well as ports 3 and 5. Therefore, a connection on p4 (or p3) with angle α is symmetric to a connection on p6 (or p5) with angle $(180 - \alpha) \bmod 180$.

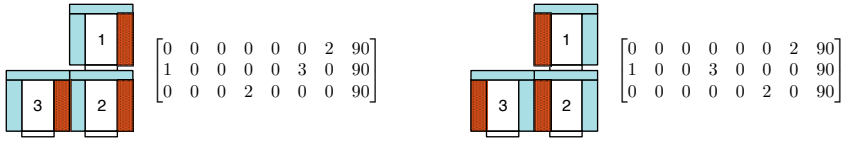


Fig. 10. Symmetry between two L-shape configurations

Figure 10 shows an example of symmetry in a 3-UBar CKBot configuration. On the left, the initial configuration. On the right, the symmetric one. We can observe that all modules have rotated. This corresponds to column permutation in the ports-connectivity matrices that are aside these configurations.

Module Permutation. Two configurations are considered isomorphic when they form the same functional shape but where modules are permuted. Figure 11 shows an example of two T-shape isomorphic configurations with their ports-connectivity matrices. Modules have the same connectivity in the two configurations but modules 1, 2 and 3 are not in the same position. This corresponds to line permutation in the ports-connectivity matrix together with a value change to refer to the new modules id (the corresponding lines are emphasized in the two matrices). On the first configuration, module 1 is on top left and is represented by the first line of the left matrix. On the second configuration, the top left module is 2 and is described by the second line in the right matrix. These two lines are identical in structure (third column is non-zero) but refer to different lines due to different neighbors.

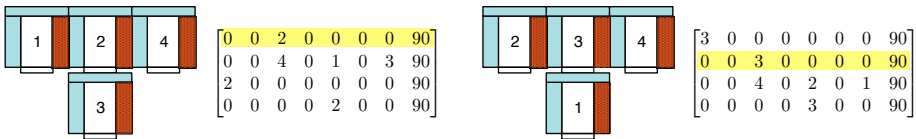


Fig. 11. Two T-shape isomorphic configurations

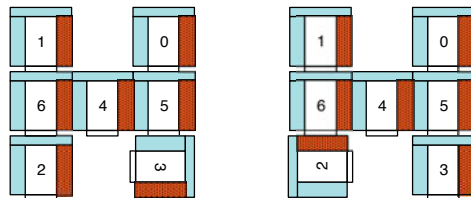


Fig. 12. Two H-shape isomorphic configurations

Module Rotation and Permutation. There is an issue to detect isomorphic configurations when they are also symmetric due to module rotation as shown in Fig. 12.

Therefore, our symbolic encoding of our ports-connectivity matrix must integrate both types of equivalences together and distinguish all equivalence classes unambiguously.

4.2 Symbolic Encoding and Canonization

Encoding. To encode the ports-connectivity matrix, we replace each line by its corresponding letter in the alphabet. Figure 13 shows, for module 1 to 5, (*i.e.* top to bottom) the encoding of a 5-UBar CKBot cross configuration (left) into an explicit ports-connectivity matrix (center) and then its corresponding symbolic representation (right). As mentioned in section 3.2, the number of letters in our alphabet is computed from formula (II).

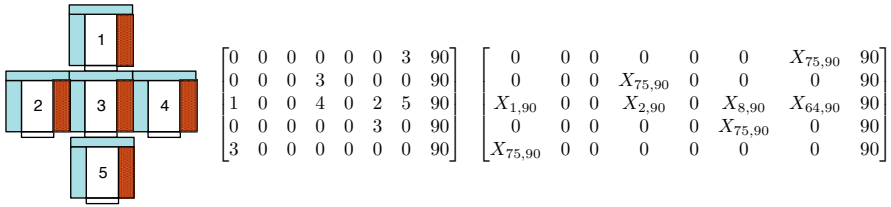


Fig. 13. Encoding of a 5-module cross shape configuration

In this figure, module 1 is connected to module 3 via port 7. So module 3 configuration is the value of the entry at line 1, column 7. Module 3 is connected to module 1 via port 1, so module 1 configuration is the value of the entry at line 3, column 1.

Canonization. To compute the configuration space as a fixed point, we must compare symbolic states to detect if a new state has been already computed or not. However, the symbolic representation is not unique since it depends on the module order. We therefore need to canonize this symbolic representation for comparison purposes. Moreover, all representations of a given class of equivalent configurations must be computed from this canonical representation thanks to columns and lines permutations.

Since the alphabet we defined to encode the port-connectivity matrix is ordered, we can arrange the matrix by sorting lines as if there was a bit-encoding of integer values where 0 means 0 and non-zero values mean 1. We use the quick sort algorithm whose average complexity is in $n * \text{Log}(n)$.

Figure 14 shows the canonization of the symbolic matrix obtained in the example illustrated by Fig. 13. This operation only changes the order of lines. Then, any arrangement of modules can be considered by numbering lines with different module identities.

As an illustration, we can deduce from the canonical matrix, the cross configuration shown in Fig. 13 by labeling lines with module identities in the following order: 1, 4, 2, 5, 3. Similarly, all equivalent configurations due to module permutations can be reconstituted by setting new modules identities affected to lines (*e.g.* configuration where

$$\begin{array}{c}
\text{Computed symbolic matrix} \\
\left[\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & X_{75,90} & 90 \\
0 & 0 & 0 & X_{75,90} & 0 & 0 & 0 & 90 \\
X_{1,90} & 0 & 0 & X_{2,90} & 0 & X_{8,90} & X_{64,90} & 90 \\
0 & 0 & 0 & 0 & 0 & X_{75,90} & 0 & 90 \\
X_{75,90} & 0 & 0 & 0 & 0 & 0 & 0 & 90
\end{array} \right]
\end{array}
\quad
\begin{array}{c}
\text{Canonized symbolic matrix} \\
\left[\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & X_{75,90} & 90 \\
0 & 0 & 0 & 0 & 0 & X_{75,90} & 0 & 90 \\
0 & 0 & 0 & X_{75,90} & 0 & 0 & 0 & 90 \\
X_{75,90} & 0 & 0 & 0 & 0 & 0 & 0 & 90 \\
X_{1,90} & 0 & 0 & X_{2,90} & 0 & X_{8,90} & X_{64,90} & 90
\end{array} \right]
\end{array}$$

Fig. 14. Canonization of the ports-connectivity symbolic matrix shown in Fig. 13

lines in the canonical matrix are 5, 4, 3, 2, 1 correspond to another configuration of the equivalence class).

Symbolic Configuration Space. The symbolic encoding of the configuration space allows us to compute the *symbolic configuration space*. Each node of this reduced graph is a symbolic configuration. The symbolic configuration space is thus much smaller than the configuration space: the node ratio is exponential since each equivalence class grows with the number of modules (and only one symbolic configuration is required to store all the configurations that belong to this class).

In the next section, we focus on the way to compute the symbolic configuration space, as well as on the way to use it for defining the moves a robot made from CKBots must perform to change its configuration.

5 Reconfiguration

Reconfiguration of a modular robot is a key feature since it is used for both movement and adaptation. It faces both scalability and computation time issues, especially when reconfiguration is to be performed on the fly. It is thus necessary to define an efficient transition system and operations. To do so, we take advantage of the symbolic representation elaborated in section 4.

5.1 Transition Relation Between Symbolic Configurations

A transition occurs when the robot changes from one symbolic configuration to another. More specifically, a transition occurs when a module changes its configuration in terms of connectivity and/or rotation. Several modules can perform a transition during a reconfiguration of the robot. We distinguish two types of transitions:

- **Functional Transitions:** they lead to rotations on the different modules degrees of freedom. They enable motion and do not alter the modules connectivity.
- **Structural Transitions:** they involve changes in modules connectivity and consist of a connection/disconnection of ports.

Functional Reconfiguration. Functional reconfiguration does not alter the connectivity. It enables motion and involves the rotation of modules. In this setting, the functional successors of a module configuration are such that only the angle varies.

If D is the increment used on the angle scale such that $0 \leq D \leq 180$, let a module i which has a configuration $X_{i,\alpha}$. The possible successors of the current state for i are

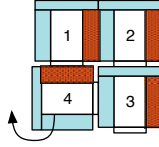


Fig. 15. Impossible rotation from 90 to 180 degrees for the bottom-left module

configurations $X_{i,\gamma}$ where $\gamma = ((\alpha + n \times D) \bmod 180)$, $n \in \mathbb{N}$ (n being the number of D steps of the angle change).

All rotations cannot be performed in a functional reconfiguration. In particular, when all modules are interconnected as in Fig. 15, a rotation from 90 to 180 degrees cannot take place. Since all modules are connected, module 4 must first disconnect from module 1 before performing a rotation. Therefore, a set of structural transitions may be necessary before a functional reconfiguration can actually happen.

Structural Reconfiguration. We consider in this paper that a functional reconfiguration only involves functional transitions, while a structural reconfiguration involves both kinds of transitions.

Since our study focuses on the CKBot, which is a hybrid robot (lattice or chain configurations), three assumptions must be made on the considered transitions:

Assumption 1: The successor of a symbolic configuration is reached through at most an atomic action for each module: connection/disconnection⁶ or rotation.

Assumption 2: When a module is connected through one face only, it cannot disconnect, to avoid breaking the lattice or chain shape of the robot. This restriction is sometimes considered in similar work like [3].

Assumption 3: We consider that only one action is performed at a time for the N modules involved in the symbolic configuration⁷.

Building Successors of a Symbolic Configuration. To illustrate structural reconfiguration, let us consider again the simplified 3-port modules whose alphabet is presented in Fig. 9. Configuration [010] (letter $X_{2,\alpha}$) has the following set of successors: $\{[010 \gamma]$ (letter $X_{2,\gamma}$), $[011 \alpha]$ (letter $X_{3,\alpha}$), $[110 \alpha]$ (letter $X_{6,\alpha}$) $\}$.

5.2 Generating and Exploiting the Symbolic Configuration Space

Computation of the symbolic configuration space is similar to the generation of the state space in model checking. It corresponds to a fixed point on the exploration of all possible moves. As for tools like greatSPN [7] that manage symmetries, each new symbolic state must be discovered by performing a symbolic evolution (*e.g.*, symbolic firing in model checking) of the system as described in [2].

⁶ A module may connect or disconnect at most one face per reconfiguration.

⁷ Similarly to model checking, we set an execution semantics at a low granularity: a single operation in the whole system. Interleaving between these single actions in the system ensures that we are compatible with a semantics where there are several parallel moves as for Petri Nets [6].

Input: *Initial*, the initial configuration of the robot (encoded in a symbolic way)
Output: returns a symbolic configuration Space
 $SymbConfSpace_1 = \emptyset$;
 $SymbConfSpace_2 = Initial$;
while $SymbConfSpace_1 \neq SymbConfSpace_2$ **do**
 $SymbConfSpace_1 = SymbConfSpace_2$;
 foreach configuration $c \in SymbConfSpace_2$ **do**
 foreach configuration $s = Successor(c)$ **do**
 $s' = Canonize(s)$;
 if $s' \notin SymbConfSpace_2$ **then**
 $SymbConfSpace_2 \leftarrow SymbConfSpace_2 \cup s'$;
 end
 Add Link between c and s' ;
 end
 end
end
return $ConfSpace_1$;

Algorithm 1. Generation of the symbolic configuration space

Algorithm 1 describes the computation of the symbolic configuration space. Successors of c are computed by applying possible connections/disconnections and angle rotation of each module of c .

Since the symbolic configuration space is an oriented graph, searching a move that leads from a concrete configuration c to any (the closest) concrete configuration $c' \in C$, the class of the target form the robot must reach is very easy. It corresponds to a shortest path search in an oriented graph like the Dijkstra algorithm [5] (in this case, all arcs in the symbolic configuration space are valued by 1).

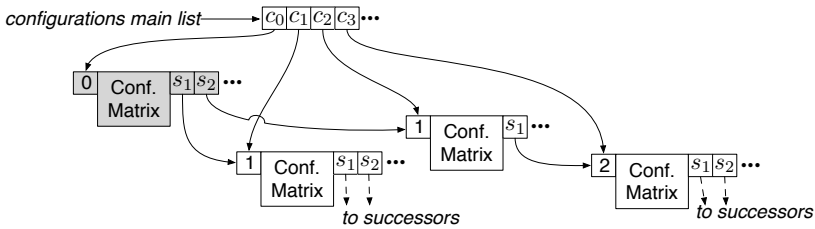


Fig. 16. Data structure to store the symbolic configuration space

So far, the data structure elaborated in the prototype is described in Fig. 16 which shows how 4 configurations $c_0, c_1, c_2, c_3 \dots$ (out of more) are represented. *Configurations main list* represent the head pointer to this list of configurations. In this example, c_0 points to the initial configuration, which is shown in grey. Its distance to the initial configuration is 0 and it has two successors c_1 and c_2 , which S_1 and S_2 of c_0 point to. The distance of c_1 and c_2 to the initial configuration is 1. We also show one successor of c_2 (c_3) whose distance to the initial configuration is 2.

Such data structure is suitable to search paths from one configuration to another one in the symbolic configuration space. Memory required to store the symbolic configuration space can be computed with formula (2) for the CKBot (7 ports and one degree of freedom):

$$Memory(bytes) = \underbrace{[(S_{int} \times N \times 8) + 1] \times NB_{sconf}}_{\substack{\text{all configurations} \\ \text{one symbolic} \\ \text{configuration}}} + \underbrace{S_{pt} \times NB_{arcs}}_{\text{all successors}} + \underbrace{S_{pt} \times NB_{sconf} + 1}_{\text{main list}} \quad (2)$$

where N is the number of modules in the configuration, S_{int} is the number of bytes to store an integer, S_{pt} the number of bytes to store a pointer, NB_{sconf} the number of symbolic configurations in the state space and NB_{arcs} the number of arcs.

6 Performance Evaluation

To assess our modeling approach, we implemented a prototype to evaluate its benefits. The idea is to get an estimation of the gain provided by our symbolic representation. To do so, we consider two experiments:

- the construction of the symbolic configuration space to evaluate if it can be computed off-line and then embedded into a reasonable amount of memory,
- the on-the-fly computation of a path between a concrete current configuration of the system and the "closest"⁸ concrete configuration in a class of configurations the system must reach.

For the experiments, we selected three types of initial configurations. First, the *line* configuration corresponds to a line of N modules. Second, the *square* configuration corresponds to a square of N modules. Finally, the *crawler* configuration is the repetition of a 4-CKBot pattern shown on the left side of Fig. 17.

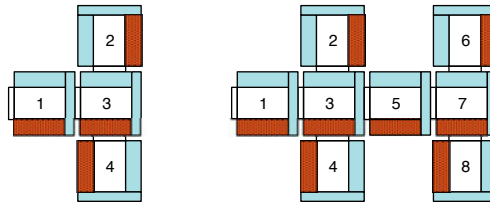


Fig. 17. Crawler configuration for 4 and 8 modules

All experiments to compute the symbolic configuration space were run on a 2.80GHz Intel Hyperthreaded Xeon computer with 14Gbytes of memory. As we show later, this does not mean that such a configuration is required to exploit the produced symbolic configuration space.

⁸ The one that can be reached with the smallest number of transitions.

6.1 Experiment 1: Generating the Symbolic Configuration Space

For the first experiment, we computed the full symbolic configuration space from the three selected initial configurations with several values of N . For rotation, we consider $D = 90^\circ$ (thus, three positions for the angle and an alphabet with $2^7 \times (\frac{180}{90} + 1) = 384$ letters according to formula (1)).

Table 1 summarizes the data collected in the first experiment. Columns, from left to right show: the value of N , the number of concrete configurations, the size of the symbolic configuration space (nodes/arcs), the ratio between the number of symbolic configurations and the number of concrete ones, the memory required by the program to compute the symbolic configuration space, the time required to compute the symbolic configuration space, and the estimated memory required to store the computed configuration space (evaluated with formula (2)).

Table 1. Evaluating performances of computation and storage of the configuration space

N	Number of Concrete Configurations	Size of Symbolic Config. Space	Ratio	Symbolic Representation		
				Memory for Computation (MB)	Time for Computation (s)	Memory to Store Symbolic Conf. Space (MB)
Line configuration						
4	3888	81/202	$2 \times 4!$	3.44	0.42	0.019
6	$\sim 1.049 \times 10^5$	729/1822	$2 \times 6!$	12.48	9.39	0.159
8	$\sim 5.297 \times 10^7$	6561/16442	$2 \times 8!$	116.88	155	1.408
10	$\sim 4.285 \times 10^{11}$	59049/147622	$2 \times 10!$	1272	2496	12.619
Square configuration						
4*	4032	84/925	$2 \times 4!$	4.45	0.56	0.064
8*	$\sim 2.91 \times 10^8$	36093/388209	$2 \times 8!$	789	736.69	25.901
Crawler configuration						
4	3888	81/202	$2 \times 4!$	3.55	0.46	0.019
8*	$\sim 1.01 \times 10^{10}$	124652/311360	$2 \times 8!$	2306	2286	26.616

As expected, the symbolic configuration space offers great gains compared to the concrete one. The largest symbolic configuration space for 8 modules can be stored in a few mega bytes that is now easy to embed in small devices. This could not be the case with a concrete representation of the configuration space that quickly contains billions of elements. Moreover, symbolic ports-connectivity matrices are not stored as sparse ones and we use 64-bits integer to encode the alphabet and 64 bits to encode a pointer ($S_{int} = S_{pt} = 8$ bytes in formula (2)). Consequently, less memory could easily be consumed by using sparse matrices and/or less bits to encode a letter in the alphabet.

For *square* and *crawler* configurations, our fixed-point algorithm also computes configurations that must be discarded because they lead to deadlocks or absurd situations (e.g. modules that are located in the same tridimensional position). When such states are detected, we note the corresponding line with a * in Table 1. Such cases are usually rare compared to the size of the symbolic state space: for example, 360 configurations are discarded for the *square* with 4 modules, 88 560 for the *square* with 8 modules and only 1 for the *Crawler* with 8 modules. This only affects the computation time and not our most important evaluation criterion in this experiment: the amount of memory required to store the symbolic configuration space.

We also note that we could not compute the configuration space for more than 8 modules in most cases. This is an implementation problem that should be considered in

further work (intensive recursions and allocations lead to intensive memory consumption). Execution time is also quite long when N grows but, since the configuration space is computed off-line, there is no time constraint on this step of the process. This should be corrected later with a more refined version of our first prototype.

6.2 Experiment 2: Computing Paths in the Symbolic Configuration Space

For the second experiment, we computed the full configuration space from the three selected initial configurations with several values of N . Then, we performed several searches between a randomly selected departure concrete space and a randomly selected target symbolic state. We also considered $D = 90^\circ$ for rotations.

Table 2 summarizes the data collected in this second experiment. Columns, from left to right show: the value of N , the size of the symbolic configuration state, the minimum, average and maximum time (in ms) required to compute a path, the minimum, average and maximum length of computed paths.

Table 2. Performances of configuration search

N	Size of Symbolic Config. Space	Search Time (ms)			Path length		
		Min	Avg	Max	Min	Avg	Max
Line configuration							
4	81	0.018	0.0915	0.199	1	11.75	31
8	6561	1.195	1.515	1.717	375	1649	2925
Square configuration							
4	84	0.021	1.107	1.697	0	17	88
8	36093	0.506	1.212	404.87	19	781	2604
Crawler configuration							
4	81	0.039	0.118	0.187	3	12	20
8	124652	0.037	1.810	3.561	2	517	1456

Once again, time performances are quite good and show that computation could be performed on the fly by a software that drives the robot since it never takes more than half a second (averages remains around a few milliseconds). The average size of computed paths remain reasonable, compared to the size of the configuration space (e.g. 781 transitions for a 10^{11} states configuration space in the case of the *Square* with 8 modules).

These data are extracted from a prototype that was not optimized and performances can clearly be enhanced.

7 Conclusion

Configuration recognition and dynamic auto-reconfiguration of modular and self-reconfigurable robots face numerous challenges, ranging from hardware to software and control issues. The one we focused on in this paper is related to the configuration space combinatorial explosion when the number of modules grows.

We present in this paper a symbolic encoding technique, inspired from the ones developed for model checking that also suffers from combinatorial explosion. We use this technique to represent configurations of the CKBot, an hybrid modular and

self-reconfigurable robot. The symbolic representation of configurations exploits structural symmetries in the modules that allow to gather in one class several equivalent configurations.

These new techniques are far more efficient compact representation of the robot configuration space than the initial encoding presented in [8]. Modules rotation, permutation or a combination of both can be quickly recognized and disambiguated. Our technique can be easily adapted to robots presenting a hardware topology similar to the CKBot one. We call this new representation the *symbolic configuration space*.

A second contribution is the computation of reconfiguration. Since the symbolic configuration space can be stored in a small amount of memory, reconfiguration corresponds to the computation of a path between two nodes in the configuration space. This can be achieved by classical graph-based algorithms such as the one of Dijkstra.

Experiments made on a first prototype show promising results. The expected exponential gain between the symbolic configuration space of a system and its related configuration space is observed for several types of initial configurations. Thus, all configurations can be stored in a few MegaBytes that could not be the case otherwise (more memory can technically be embedded in small devices like the robot modules we consider). Computation of the symbolic state space can be long but this operation is typically performed off-line. Thus, no performances are really needed.

Experiment also showed that (still with our first prototype) a path between two configurations (*e.g.* reconfiguration of the robot) could be computed in a average time of a fewmilliseconds. This enables the use of our technique on the fly during the robot mission. When the robot is performing a move, the next one can be computed.

Since our experiment was done in a quickly implemented prototype, numerous optimizations can be applied that should increase performances.

Future work will consider some application to a real mission with a CKBot-based robots (possibly with several types of modules) to assess usability with several configurations when the robot is driven by an external computer or with embedded code.

References

1. Arney, D., Fischmeister, S., Lee, I., Takashima, Y., Yim, M.: Model-based Programming of Modular Robots. In: 13th IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC 2010), pp. 87–91. IEEE Computer Society, Carmona (2010)
2. Chiola, G., Dutheillet, C., Franceschinis, G., Haddad, S.: Stochastic Well-Formed Coloured Nets for Symmetric Modelling Applications. IEEE Transactions on Computers 42(11), 1343–1360 (1993)
3. Christensen, D.J.: Evolution of shape-changing and self-repairing control for the ATRON self-reconfigurable robot. In: Proceedings 2006 IEEE International Conference on Robotics and Automation, ICRA 2006, pp. 2539–2545 (2006)
4. Clarke, E.M., Emerson, E.A., Jha, S., Sistla, A.P.: Symmetry Reductions in Model Checking. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 147–158. Springer, Heidelberg (1998)
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press, Cambridge (2003)
6. Girault, C., Valk, R. (eds.): Petri Nets and System Engineering, ch. 2, pp. 9–23. Springer, Heidelberg (2003)

7. GreatSPN. Petri nets suite, <http://www.di.unito.it/~greatspn>
8. Park, M.G.: Configuration recognition, Communication Fault Tolerance and Self-reassembly for the CKBot. PhD thesis, University of Pennsylvania (2009)
9. Shen, W.-M.: Self-reconfigurable robots for adaptive and multifunctional tasks. Technical report, University of Florida, Florida, USA (December 2008)
10. The CKBot home page, <http://modlabupenn.org/ckbot/>
11. Valmari, A.: The State Explosion Problem. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 429–528. Springer, Heidelberg (1998)
12. Yim, M., White, P., Park, M., Sastra, J.: Modular Self-Reconfigurable Robots. In: Encyclopedia of Complexity and System Science. Springer, Heidelberg (2009)
13. Yim, M., Zhang, Y., Roufas, K., Duff, D., Eldershaw, C.: Connecting and disconnecting for chain self-reconfiguration with PolyBot. IEEE/ASME Transactions on mechatronics, special issue on Information Technology in Mechatronics (2003)

Formal Methods @ Runtime

Radu Calinescu¹ and Shinji Kikuchi²

¹ Aston University, Aston Triangle, Birmingham B4 7ET, UK

r.c.calinescu@aston.ac.uk

² Fujitsu Laboratories Limited, 4-1-1 Kamikodanaka, Nakahara-ku, Kawasaki,

Kanagawa 211-8588, Japan

skikuchi@jp.fujitsu.com

*J'aimais, et j'aime encore, les mathématiques pour elles-mêmes comme n'admettant pas l'hypocrisie et le vague, mes deux bêtes d'aversion*¹.

H. B. Stendhal, *La Vie d'Henri Brulard*

Abstract. Heuristics, simulation, artificial intelligence techniques and combinations thereof have all been employed in the attempt to make computer systems adaptive, context-aware, reconfigurable and self-managing. This paper complements such efforts by exploring the possibility to achieve runtime adaptiveness using mathematically-based techniques from the area of formal methods. It is argued that *formal methods @ runtime* represents a feasible approach, and promising preliminary results are summarised to support this viewpoint. The survey of existing approaches to employing formal methods at runtime is accompanied by a discussion of their challenges and of the future research required to overcome them.

1 Introduction

The use of rigorous logic in the design and analysis of computer programs was first proposed in the late 1960s [1,2]. Several decades and a Turing Award [3] later, the set of mathematically-based techniques collectively known as *formal methods* comprises effective tools for the formal specification [4,5], development [6] and verification [7] of computer systems. Already widely adopted in hardware design and verification [8], formal methods have more recently been applied to the development of software systems [9], where they are increasingly used to improve the quality of software alongside traditional approaches such as testing and simulation. Contributors to the latter advance include major software companies, who are not only using formal methods internally, but also integrating formal techniques into their software development platforms, and actively contributing to formal methods research. These developments have established formal methods as an effective aid in producing high-integrity systems—a key challenge for today’s developers of complex computer systems.

¹ “I used to love mathematics for its own sake, and I still do, because it allows for no *hypocrisy* and no *vagueness*, my two *bêtes noires*.”

This paper explores the possibility to use formal methods in addressing another grand challenge of computer systems, namely runtime adaptation. Computer systems are increasingly employed—and expected to cope with limited or no human intervention—in applications characterised by continual change to system state, workload and objectives. In response to this challenge, the research community has come up with the idea of adding self-adaptation capabilities to computer systems. The impressive body of work carried out in newly emerged research fields such as autonomic, context-aware and ubiquitous computing has so far led to the development of adaptive systems based on a combination of heuristics, simulation and artificial intelligence techniques. While experimental results suggest that such solutions are often effective, they alone cannot provide the high levels of predictability and dependability that adaptive, autonomic computer systems are typically expected to attain [10,11]. One of the most promising approaches to achieving these necessary characteristics is to use mathematically based techniques in the runtime adaptation process, i.e., to employ *formal methods @ runtime* (FM@R).

This paper advocates that FM@R have the potential to contribute to the development of adaptive computer systems that offer high levels of predictability and dependability in several ways. Some of these are explored in the following sections, starting with the authors' work in Sections 2–4. Section 2 presents the use of quantitative verification to comply with non-functional requirements in adaptive computer systems. Section 3 describes how lightweight formal methods can be used to synthesise the reconfiguration operations that adaptive computer systems must undertake in order to achieve their objectives in the presence of changes in system state, workload and environment. Section 4 explains the use of model checking to verify adaptation rules in computer systems. The concluding section summarises the potential of FM@R, and suggests possible avenues for overcoming some of their current limitations.

2 Quantitative Verification @ Runtime

2.1 Description

Quantitative verification is a mathematically-based technique for establishing the correctness, performance and reliability of systems that exhibit stochastic behaviour [12]. Given a precise mathematical model of a real-world system, and formal specifications of quantitative properties of this system, an exhaustive analysis of these properties is performed. Example properties include the probability that a fault occurs within a specified time period, and the expected power consumption of a computer system under a given workload.

While quantitative verification is traditionally used for the off-line analysis of system models such as Markov chains and Markov decision processes, recent research has successfully employed an on-line version of the technique to support self-optimisation in adaptive computer systems [13,14,15]. This approach involves using a quantitative system model parameterised by the different scenarios that the system can operate in and by the different configurations that

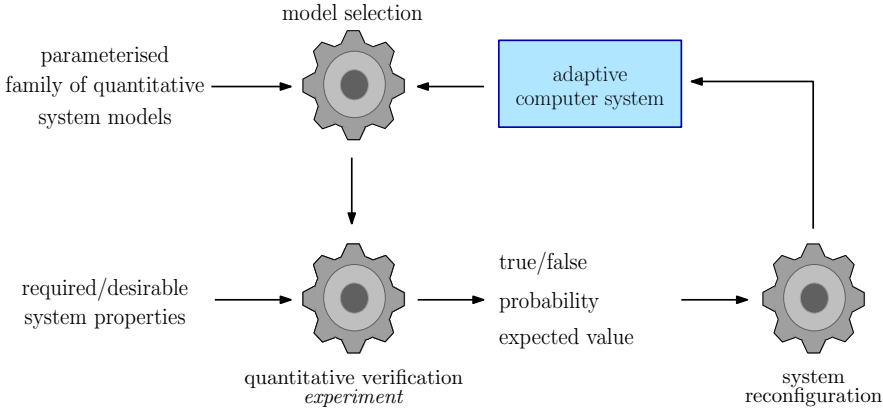


Fig. 1. Using quantitative verification in adaptive computer systems

can be selected for it (Figure 1). The system is monitored continuously to identify the scenario it operates in, and this monitoring information is used to fix the scenario-based model parameters. A quantitative verification *experiment* is then carried out at runtime, to assess which feasible configuration satisfies the system objectives best for these parameter values. Finally, this configuration is adopted automatically, thus ensuring that the system continues to achieve its objectives as it transitions from one operating scenario to another.

The approach is applicable to systems that fulfil two requirements:

1. The system behaviour that is of relevance for the planned adaptation can be modelled as a Markovian chain. Several examples of applications that satisfy this requirement are described later in this section.
2. The system objectives can be expressed as combinations of formal quantitative properties, i.e., properties specified in probabilistic computational tree logic (PCTL) [16] and continuous stochastic logic (CSL) [17] for discrete- and continuous-time Markovian models, respectively [12]. As illustrated in the remainder of the section, this ensures that non-functional properties describing reliability, performance and cost/reward system characteristics are supported by the approach.

2.2 Practical Realisation

The probabilistic model checker PRISM [18] and the general-purpose autonomic computing framework GPAC [13] were used to implement the FM@R approach from Figure 1 in a range of application domains:

- Dynamic power management [14]. The approach was used to ensure that a Fujitsu disk drive subjected to variable workload achieved predictable response time in an energy-efficient way. The disk drive was modelled as a continuous-time Markov chain, and the system objectives were specified as a combination of *reward-extended* [12] CSL properties.

- Adaptive allocation of data-centre resources [19]. Runtime quantitative verification was used to support the adaptive allocation of servers to variable-workload clusters in the presence of data-centre component failures and repairs. A continuous-time Markov chain was used to describe the system behaviour, and the required cluster reliability properties were expressed as CSL formulas. Provably optimal server allocations guaranteed to comply with the required reliability thresholds were achieved irrespective of the cluster workload.
- Dynamic QoS management in service-based systems [15]. The web services used to carry out operations within service-based systems were chosen optimally from sets of functionally equivalent services characterised by different failure rates, response times and costs. A combination of discrete- and continuous-time Markov models and of PCTL and CSL properties were used to analyse the reliability- and performance-related system properties, respectively.

In all these applications, the computation overheads associated with carrying out the quantitative analysis experiments at runtime were acceptable for realistic system sizes. For the dynamic power management of a Fujitsu disk drive, the analysis took up to a few hundreds of milliseconds. This represents, for instance, a fraction of the 1.6 seconds required for the disk to transition physically from the idle state into the sleep state. The CPU overhead was in the range 1.5%-2.5%, which was deemed acceptable for this application [14].

In the case study involving the adaptive allocation of data-centre servers, the runtime quantitative verification took between 10–30 seconds for systems comprising up to tens of servers [19]. This response time was acceptable because it represented a small delay compared to the time required to provision a server allocated to a new cluster.

Finally, using the approach for dynamic web service selection in service-based systems was feasible for workflows comprising up to eight web service invocations. Note that many of the workflows in use by the scientific community today do not exceed this size. For instance, the study carried out in [15] found that over 70% of the bioinformatics workflows from the widely used myExperiment workflow repository² comprise between one and eight web service invocations.

2.3 Challenges

Reducing the overheads associated with runtime quantitative verification and scaling the approach beyond small to medium system sizes represents a major challenge. The options explored in the effort to address this challenge include a combination of software engineering techniques and novel quantitative verification algorithms [14,15]. The former category of solutions includes the pre-execution and/or caching of quantitative analysis experiments, and the execution of the PRISM experiments for different requirements in parallel, e.g., on a multicore-processor server or on multiple machines. The latter category

² <http://www.myexperiment.org>

includes efforts for the design of iterative algorithms that derive the results of an experiment from those of the previous ones.

Another challenge is related to the expert knowledge that is required to build the models employed in the runtime quantitative analysis. Building a quantitative model at the right level of abstraction represents a non-trivial task that requires a good understanding of both formal modelling techniques and the behaviour of the target system. The applications described in the previous section were implemented by research teams with significant experience in formal methods, who invested significant time in understanding the behaviour of the computer systems involved in these applications. It is expected that an increase in the teaching of formal methods by undergraduate and graduate Computer Science programmes will enable future engineers of adaptive computer systems to apply this FM@R approach with less need for expert support.

One last challenge that is worth mentioning here is the potential need for a continual update of the model used by the approach. Systems that require the ability to adapt are often affected by internal changes that modify their behaviour in ways that may not or cannot be captured by a static model. While preliminary work to learn the parameter values for parameterised system models (cf. Figure 1) has been successful [20,15,21], there are multiple applications in which systems undergo unpredictable changes that require structural model changes (e.g., to reflect components leaving or joining the system dynamically). Devising monitoring techniques and learning algorithms capable of dealing with this scenario represents a major challenge.

3 Lightweight Formal Methods @ Runtime

3.1 Description

Lightweight formal methods represent techniques for the (often partial) specification of computer system requirements using mathematical notation drawn from set theory and first-order logic [22]. Their benefits include expressing system requirements concisely and unambiguously, and the ability to automatically derive artifacts guaranteed to satisfy these requirements. These artifacts can be fully-fledged software components or models of the system that the specification describes. The derivation process is termed *refinement* in the former case [23] and *model finding* in the latter case [24].

An FM@R approach that employs model finding was proposed in [25] and further developed in [26]. In this approach (Figure 2), a formal specification is obtained through combining formal descriptions of the characteristics, operations, constraints, goals and state of a computer system. Model-finding techniques are then employed to synthesise a model of the system that satisfies this specification. The synthesised model corresponds to a system configuration that fulfils the system goals given its current state, and is used to derive a configuration procedure capable of reaching this target configuration without violating any system constraints.

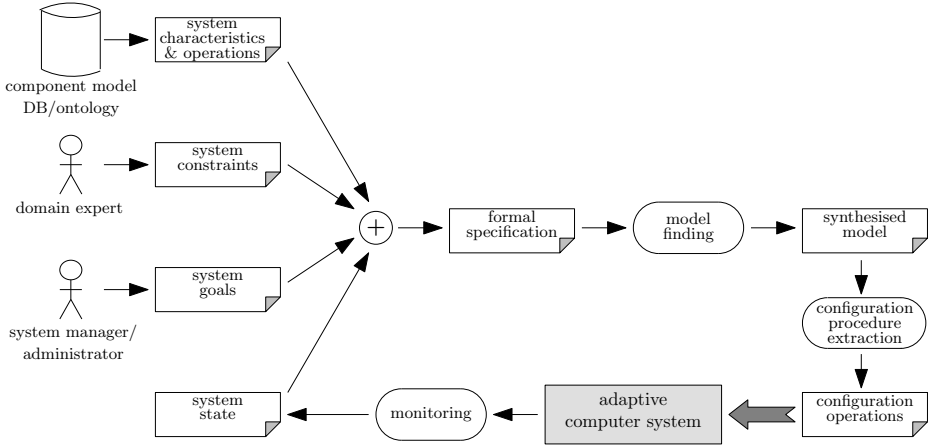


Fig. 2. Lightweight formal methods @ runtime

The approach is applicable in system adaptation scenarios that:

1. require the synthesis of configuration change procedures (or *workflows*) capable of achieving given goal conditions;
2. involve the integration of information about the system components and constraints provided by multiple domain experts.

In such scenarios, extensive discussions among the domain experts are usually required to define a configuration change procedure that can achieve the system goal without violating any of its critical conditions. As these discussions are known to be time consuming and prone to errors [25], using the FM@R approach described in this section has the advantage of deriving a provably safe configuration procedure automatically. Furthermore, organising the information provided by different domain experts into a repository of formally and declaratively specified system constraints and objectives can encourage their reuse across applications from the same domain.

3.2 Practical Realisation

The variant of the approach presented in [25,26] uses the formal specification framework Alloy [24]. The Alloy declarative specification language is used to define the formal specification in Figure 2, and the model synthesis is carried out using the model finder Alloy Analyzer.³ The Alloy formal specification comprises two parts:

1. A static part is used for the constant elements of the system. This part consists of Alloy `sig`(nature) and `fact` constructs defining the system characteristics, operations and constraints.

³ <http://alloy.mit.edu/alloy4/>

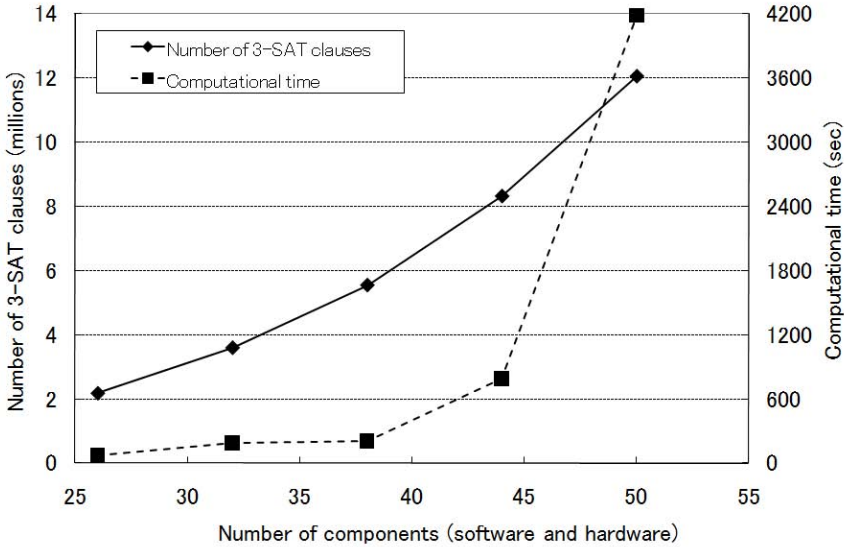


Fig. 3. The variation of the number of 3-SAT clauses and the computational time with the system size in lightweight formal methods @ runtime

2. A second part of the specification is derived at runtime. This variable part consists of Alloy `sig/pred(icate)` and `fact` constructs that encode the system goals and state, respectively.

The model that the Alloy Analyzer tool synthesises when supplied with this specification is a sequence of value assignments to variables from the operation definitions in the specification. As a result, the model maps directly on to a sequence of configuration parameter changes satisfying all given `fact` declarations. This represents a sequence of feasible state transitions that starts with the current state of the system, terminates with a state that satisfies the system goals, and does not enter any state that violates system constraints.

The approach and its application in a case study involving virtual-machine consolidation within a cluster of physical servers are presented in detail in [26]. The experimental results in Figure 3 illustrate the computational time required to synthesise a configuration change procedure in this case study. This graph shows the relation between the size of the system (i.e., the number of system components) and both (a) the number of SAT clauses to which the system specification was reduced by Alloy Analyzer; and (b) the time spent to derive the result by the SAT solver employed by the tool. These experimental results show acceptable overheads for systems comprising up to 30-40 software and/or hardware components.

3.3 Challenges

Unsurprisingly, the main challenge of using lightweight formal methods at runtime is the limited scalability of the approach. Although systems with up to 40 components are not uncommon in real-world applications, extending the applicability of the approach to larger systems is key to its adoption.

Opportunities for taking advantage of this approach in larger computer systems do exist. One such opportunity is to improve the performance of the SAT solvers at the core of the model synthesis process. A second opportunity is to devise algorithms that translate formal specifications into sets of SAT clauses comprising significantly fewer elements than the sets produced by the algorithms currently in use within Alloy.

Some of the challenges described in Section 2.3 are also valid for the FM@R approach described here. They include the expertise required to derive an appropriate system specification, and the need to keep this specification in step with potential changes in the system characteristics and operations. The potential solutions for these challenges are those already presented in Section 2.3.

4 Model Checking @ Runtime

4.1 Description

Model checking represents a formal technique for verifying whether a system satisfies its requirements [7]. The technique involves building a mathematically-based model of the system behaviour (e.g., a *Kripke structure* [7] or a *process algebra model* [27]), and checking that system properties specified formally hold within this model. For each refuted property, the technique yields a counterexample consisting of an execution path for which the property does not hold. The result is based on an exhaustive analysis of the state space of the considered model—a characteristic that sets model checking apart from complementary techniques such as testing and simulation.

Model checking @ runtime has the potential to play two key roles in adaptive computer systems. First, model checking techniques could be used to guide the adaptation process by means of an approach similar to the one described in context of quantitative verification in the previous section. Second, model checking can be used to address a major concern of adaptive computer systems, namely the correctness of the objectives specified by the administrators and users of these systems. These two applications of model checking @ runtime are detailed below.

Goal policies are often used to specify constraints, invariants or final-state conditions that an autonomic computer system is required to achieve by appropriately reconfiguring itself [13]. These policies comprise Boolean expressions similar to the formally specified requirements that model checking can verify given an appropriate model of the system behaviour. As self-adaptation requires choosing among a set of possible configuration, model checking @ runtime has the potential to confirm which of these configurations achieves the system goals.

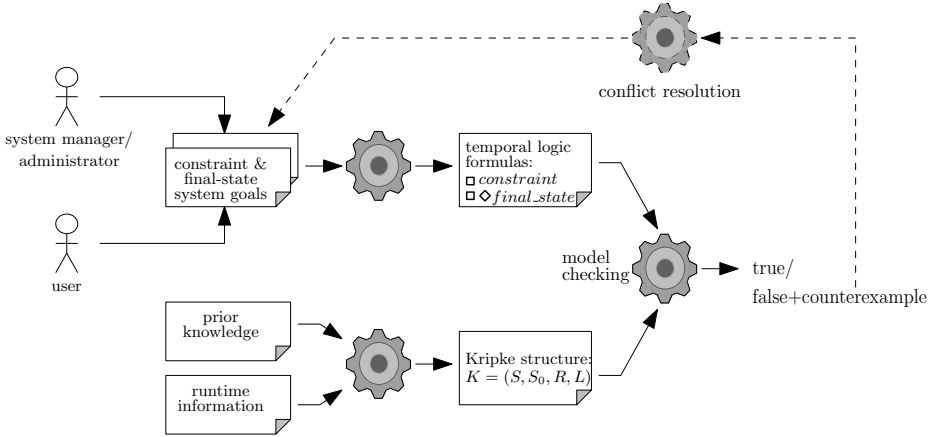


Fig. 4. Model checking for conflict detection and resolution in adaptive-system goals

Similar to the FM@R solution described in Section 2.3, this would require the verification of the formally specified system goals against a parameterised family of system models. The feasibility of the approach—in terms of response time, overhead, scalability and expressiveness—is still to be determined.

The detection and resolution of conflicts in the objectives that developers, administrators and users specify for adaptive computer systems represents another area in which model checking @ runtime could provide effective solutions. Unlike the configuration procedure synthesis presented in Section 3, this approach is suitable for identifying the conflicts between atomic “if-then”-type rules and objectives to be achieved by an adaptive computer system. This application of model checking @ runtime is described in more detail next.

4.2 Practical Realisation

The model checking @ runtime approach in [28,26] uses the model checker SPIN [29] to detect conflicts in the constraint and final-state goals of an adaptive computer system. Note that the model checking is performed at run time each time when the Kripke structure that describes the system behaviour changes to reflect the variable context that the system operates in. The counterexamples generated by the model checker for each identified conflict can be used to identify reachable system states that do not comply with its goals, and thus represent a starting point for resolving these conflicts as indicated in Figure 4.

Using the off-the-shelf model checker SPIN (i.e., a tool not intended for run-time use) and without effort to minimise the size of the model state space through techniques such as model abstraction [30], the approach was shown to be applicable to real-world adaptive systems comprising under ten components [28]. The experimental results from a case study involving the detection of policy conflicts in a utility resource management system [26] are shown in Figure 5.

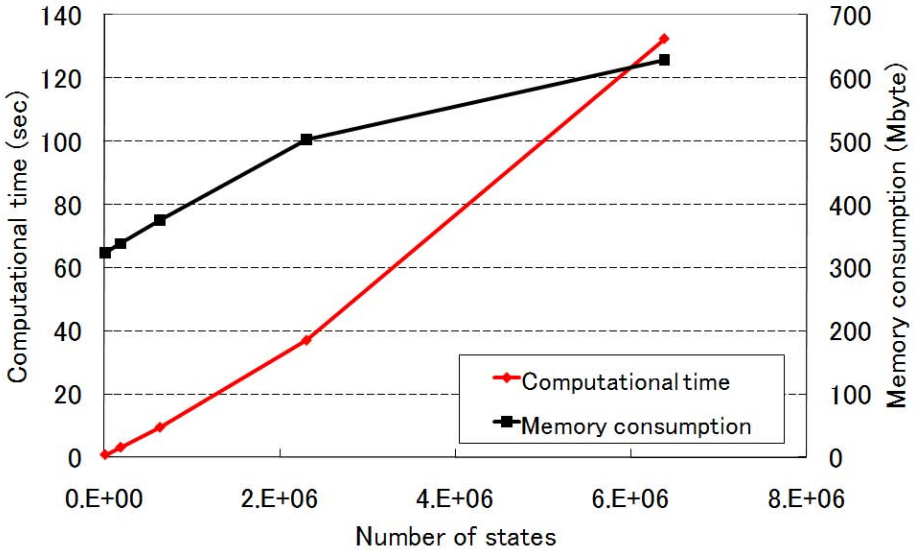


Fig. 5. Runtime detection of policy conflicts using model checking—CPU and memory overheads

4.3 Challenges

As for the other FM@R approaches described in the paper, the main challenge with model checking @ runtime is scalability. Work to address this challenge is still in its very early stages. The only promising result that the authors are aware of is the technique presented in [31], which aims at reducing the number of possible system configurations based on the system context or circumstances. The integration of this technique within the general FM@R approach from Figure 4 has the potential to reduce the size of the analysed Kripke structure significantly, and thus to render the approach applicable to larger adaptive systems.

5 Other FM@R-Related Approaches

Over the past decade, two research communities have carried out work that is relevant to the runtime use of formal methods.

The *Runtime Verification* research community⁴ has been running annual workshops to explore techniques for the monitoring and formal analysis of program executions since 2001. The techniques developed by this community are concerned with verifying the correctness of formally-specified properties against individual execution traces or programs or *runs* [32]. This represents a powerful approach to detecting violations of correctness properties after they happen and

⁴ <http://runtime-verification.org>

their associated runs are recorded. As the goal is to analyse a single run at a time, these techniques scale well. However, unlike the model checking @ runtime approach presented in Section 4, runtime verification cannot guarantee the lack of constraint violations or help prevent their occurrence.

More recently, the *Models@Run.Time* community was formed to bring together researchers working on projects that involve the use of various types of models at runtime. A wide range of results including new techniques and applications have been presented at the “Models@run.time” workshops organised annually since 2005.⁵ These results include the use of evolutionary computation techniques to synthesise models for evolving the configuration of systems that need to meet changing requirements [33].

A summary of the key techniques and approaches developed by the Models@Run.Time research community is available from [34]. In addition to presenting a general overview of using models in runtime environments [35], [34] describes an approach that uses an architectural model to automate configuration change in dynamically adaptive systems [31].

6 Conclusion

The computer systems of the future will increasingly face two conflicting challenges. On the one hand, they will need to adapt continually to change. On the other hand, they will be required to provide high integrity, availability and predictability. So far, the two challenges have been addressed largely in isolation, by different research communities. Adaptive computer systems have typically been produced through the application of heuristics, simulation and artificial intelligence techniques. In contrast, high integrity and predictability has been achieved through the application of formal methods—often to systems that operate in fixed scenarios, such as hardware components or communication protocols.

This paper advocates the integration of the two research areas. It is envisaged that the use of *formal methods @ runtime* (FM@R) has the potential to contribute to the solution of the dual challenge faced by future computer systems. The early FM@R research summarised in the paper suggests several ways of exploiting formal methods in the context of adaptive computer systems. Using off-the-shelf tools, these FM@R techniques were shown to be applicable to small or, in some cases, to medium-sized systems. In this respect, FM@R are in a position similar to that of model checking in the early to mid 1980s: the benefits of the approach are clear, but it is hard to confidently predict whether it can scale to large systems. Some of the avenues to explore in the search for a positive answer to this unknown have been suggested in the paper—more effective translators of formal specifications into SAT clauses (Section 3), iterative verification techniques (Section 2) and algorithms for reducing the state space of formal models (Section 4). Other potential FM@R research directions include taking advantage of assume-guarantee [36] and hierarchical [37] verification techniques in the runtime analysis of adaptive computer system models.

⁵ <http://www.comp.lancs.ac.uk/~bencomo/WorkshopMRT.html>

Pursuing these directions will require significant research effort, and achieving their objectives will take more than isolated breakthroughs. What justifies this undertaking is the promise of dependable and predictable adaptation—a major stepping stone in the human quest for non-biological systems capable of learning, adaptation, reasoning and planning.

Acknowledgements. This work was partly supported by the UK Engineering and Physical Sciences Research Council grant EP/H042644/1.

References

1. Hoare, C.A.R.: An axiomatic basis for computer programming. *Communications of the ACM* 12(10), 576–583 (1969)
2. Floyd, R.W.: Assigning meanings to programs. *Proceedings of the American Mathematical Society Symposia on Applied Mathematics* 19, 9–31 (1967)
3. US National Science Foundation: Model checking pioneers receive Turing Award, most prestigious in computing, Press Release 08-022 (February 2008)
4. Abrial, J.-R.: *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, Cambridge (1996)
5. Woodcock, J., Davies, J.: *Using Z. Specification, Refinement and Proof*. Prentice-Hall, Englewood Cliffs (1996)
6. Lano, K.: *The B Language and Method: A Guide to Practical Formal Development*. Springer, Heidelberg (1996)
7. Clarke, E.M., Grumberg, O., Peled, D.A.: *Model Checking*. MIT Press, Cambridge (2000)
8. Kropf, T. (ed.): *Formal Hardware Verification: Methods and Systems in Comparison*. LNCS, vol. 1287. Springer, Heidelberg (1997)
9. Clarke, E.M., Lerda, F.: Model checking: Software and beyond. *Journal of Universal Computer Science* 13(5), 639–649 (2007)
10. Dai, Y.-S.: Autonomic computing and reliability improvement. In: *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2005)*, pp. 204–206 (2005)
11. Sterritt, R., Bustard, D.: Autonomic computing — a means of achieving dependability? In: *Proceedings of the 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, ECBS 2003* (2003)
12. Kwiatkowska, M.: Quantitative verification: Models, techniques and tools. In: *Proc. 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 449–458. ACM Press, New York (2007)
13. Calinescu, R.: General-purpose autonomic computing. In: Denko, M., et al. (eds.) *Autonomic Computing and Networking*, pp. 3–30. Springer, Heidelberg (2009)
14. Calinescu, R., Kwiatkowska, M.: Using quantitative analysis to implement autonomic IT systems. In: *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pp. 100–110 (2009)
15. Calinescu, R., Grunske, L., Kwiatkowska, M., Mirandola, R., Tamburrelli, G.: Dynamic QoS management and optimisation in service-based systems. *IEEE Transactions on Software Engineering* (2010), <http://doi.ieeecomputersociety.org/10.1109/TSE.2010.92>

16. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. *Formal Aspects of Computing* 6(5), 512–535 (1994)
17. Aziz, A., et al.: Model checking continuous time Markov chains. *ACM Transactions on Computational Logic* 1(1), 162–170 (2000)
18. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: Hermanns, H. (ed.) *TACAS 2006*. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
19. Calinescu, R., Kwiatkowska, M.: CADs*: Computer-aided development of self-* systems. In: Chechik, M., Wirsing, M. (eds.) *FASE 2009*. LNCS, vol. 5503, pp. 421–424. Springer, Heidelberg (2009), <http://qav.comlab.ox.ac.uk/papers/fase09.pdf>
20. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by runtime parameter adaptation. In: *Proc. 31st International Conference on Software Engineering (ICSE 2009)*, pp. 111–121. IEEE Computer Society, Los Alamitos (2009)
21. Calinescu, R., Johnson, K., Rafiq, Y.: Using observation ageing to improve Markovian model learning in QoS engineering. In: *Proceedings 2nd ACM/SPEC International Conference on Performance Engineering* (2011)
22. Agerholm, S., Larsen, P.G.: A lightweight approach to formal methods. In: Hutter, D., Traverso, P. (eds.) *FM-Trends 1998*. LNCS, vol. 1641, pp. 168–183. Springer, Heidelberg (1999)
23. Schneider, S.: *The B-Method*. Palgrave Macmillan, Basingstoke (2001)
24. Jackson, D.: *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Cambridge (2006)
25. Kikuchi, S., Tsuchiya, S.: Configuration procedure synthesis for complex systems using model finder. In: *Proceedings of the 15th IEEE International Conference on Complex Computer Systems*, Oxford, UK (March 2010) (to appear)
26. Calinescu, R., Kikuchi, S., Kwiatkowska, M.: Formal methods for the development and verification of autonomic IT systems. In: Cong-Vinh, P. (ed.) *Formal and Practical Aspects of Autonomic Computing and Networking: Specification, Development and Verification*. IGI Global (to appear, 2011)
27. Roscoe, A.W.: *The theory and practice of concurrency*. Prentice-Hall, Englewood Cliffs (1998), <http://web.comlab.ox.ac.uk/oucl/work/bill.roscoe/publications/68b.pdf>
28. Kikuchi, S., Tsuchiya, S., Adachi, M., Katsuyama, T.: Policy verification and validation framework based on model checking approach. In: *Proceedings of the 4th IEEE International Conference on Autonomic Computing*, Jacksonville, Florida (June 2007)
29. Holzmann, G.J.: *The SPIN Model Checker*. Addison-Wesley, Reading (2003)
30. Wang, C., Hachtel, G.D., Somenzi, F.: *Abstraction Refinement for Large Scale Model Checking (Series on Integrated Circuits and Systems)*. Springer-Verlag New York, Inc., Secaucus (2006)
31. Morin, B., Barais, O., Jezequel, J.-M., Fleurey, F., Solberg, A.: Models@ run.time to support dynamic adaptation. *Computer* 42(10), 44–51 (2009)
32. Leucker, M., Schallhart, C.: A brief account of runtime verification. *Journal of Logic and Algebraic Programming* 78(5), 293–303 (2009)
33. Ramirez, A.J., Cheng, B.H.C.: Evolving models at run time to address functional and non-functional adaptation requirements. In: *Proceedings of the Fourth Workshop on Models at Run Time*, Denver, Colorado, USA, pp. 31–40. ACM, New York (2009)

34. IEEE Computer: Special Issue on Models@Run.Time 42(10) (October 2009)
35. Blair, G., Bencomo, N., France, R.B.: Models@ run.time. *Computer* 42(10), 22–27 (2009)
36. Pasareanu, C.S., Dwyer, M.B., Huth, M.: Assume-guarantee model checking of software: A comparative case study. In: *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, London, UK, pp. 168–183. Springer, Heidelberg (1999)
37. Alur, R., Yannakakis, M.: Model checking of hierarchical state machines. *ACM Trans. Program. Lang. Syst.* 23(3), 273–303 (2001), doi:10.1145/503502.503503

Modular State Spaces for Prioritised Petri Nets

Charles Lakos¹ and Laure Petrucci²

¹ University of Adelaide
Adelaide, SA 5005
Australia

`Charles.Lakos@adelaide.edu.au`

² LIPN, CNRS UMR 7030, Université Paris XIII
99, avenue Jean-Baptiste Clément
F-93430 Villetaneuse, France

`Laure.Petrucci@lipn.univ-paris13.fr`

Abstract. Verification of complex systems specification often encounters the so-called state space explosion problem, which prevents exhaustive model-checking in many practical cases. Many techniques have been developed to counter this problem by reducing the state space, either by retaining a smaller number of relevant states, or by using a smart representation. Among the latter, modular state spaces [CP00, LP04] have turned out to be an efficient analysis technique in many cases [Pet05]. When the system uses a priority mechanism (e.g. timed systems [LP07]), there is increased coupling between the modules — preemption between modules can occur, thus disabling local events. This paper shows that the approach is still applicable even when considering dynamic priorities, i.e. priorities depending both on the transition and the current marking.

Keywords: Modular state spaces, prioritised Petri Nets.

1 Introduction

State space exploration is a convenient technique for the analysis of concurrent and distributed systems. Its chief disadvantage is the so-called state space explosion problem where the size of the state space can grow exponentially in the size of the system.

One way to alleviate the state space explosion problem is to use modular analysis, which takes advantage of the modular structure of a system specification. The internal activity of the modules is explored independently rather than in an interleaved fashion. Modular state space exploration has yielded significant efficiency gains in the analysis of systems where the modules exhibit strong cohesion and weak coupling [LP04, Pet05]. The benefits arise because the internal activity of individual modules can be explored independently without considering the many possible interleavings of this internal activity. Interaction between modules is only considered at synchronisation (or fused) transitions.

If the system has some form of priority, e.g. time, then the internal activity of the modules is no longer independent. An earlier internal event in one module

will precede a later internal event in another. In this way, a high priority module may preempt *all* activity of a low priority module, even without interaction. If the priority scheme is dense, e.g. real number priorities, then the priorities may eliminate many possible interleavings of activity and modular state space exploration will yield few benefits. However, if the priority scheme is coarse grained, then modular state space exploration may still be of value. This is the situation that we explore in this paper.

We consider Modular Petri Nets which incorporate a dynamic priority scheme similar to that of Bause's work [Bau97]. The scheme is termed *dynamic* because the priority of a transition depends on the current marking, not just on the firing mode. Bause considered the constraints on the priority scheme so that the prioritised net would preserve liveness and home properties of the non-prioritised net, despite having a reduced state space. By contrast, we are interested in the possible benefits of modular state spaces for prioritised nets.

We choose a priority scheme where the greater priority value implies a higher priority. Equally well, we could choose a priority scheme where lower priority values indicate a higher priority. For example, if the priority value was given by an enabling time, earlier timed events would preempt later ones.

The paper is organised as follows. After introducing the basic definitions and notations in section 2, we adapt, in section 3, the modular state space exploration technique from [CP00, LP04] to modular nets with dynamic priorities. Associated algorithms are given in section 4, together with the formal results on which they depend. Section 5 presents experimental results, showing the benefits of the approach. Finally, section 6 summarises the contributions and gives perspectives for future work.

2 Basic Definitions

This section introduces the basic concepts and notations used in the paper. A parallel is drawn between the definitions of Petri nets and prioritised Petri nets, and then between their modular extensions.

2.1 Petri Nets

We first recall the basic definitions and notations for Petri nets:

Definition 1 (Petri nets)

A Petri net is a tuple $PN = (P, T, W, M_0)$, where:

- P is a finite set of places.
- T is a finite set of transitions such that $T \cap P = \emptyset$.
- W is the arc weight function mapping from $(P \times T) \cup (T \times P)$ into \mathbb{N} .
- M_0 is the initial marking, namely a function mapping from P into \mathbb{N} .

The elements defining the Petri net behaviour can now be expressed:

Definition 2 (Markings, enabling rule)

- A marking is a function M mapping from P into \mathbb{N} . The set of all markings is denoted by \mathbb{M} .
- A transition $t \in T$ is enabled in a marking M , denoted by $M[t]$, iff $\forall p \in P : W(p, t) \leq M(p)$.
- When a transition $t \in T$ is enabled in a marking M_1 , it may occur, changing the marking M_1 to another marking M_2 , denoted by $M_1[t]M_2$ and defined by: $\forall p \in P : M_2(p) = (M_1(p) - W(p, t)) + W(p, t)$. The set of markings reachable from a marking M is: $[M] = \{M' \mid \exists \sigma \in T^* : M[\sigma]M'\}$ where T^* is the transitive closure of T .

2.2 Prioritised Petri Nets

We extend the above definitions to prioritised Petri nets, where the priority of transitions is *dynamic*, i.e. it depends on the current marking [Bau97].

Definition 3 (Prioritised Petri net)

A Prioritised Petri net is a tuple $PPN = (P, T, W, M_0, \rho)$, where:

- (P, T, W, M_0) is a Petri net.
- ρ is the priority function mapping a marking and a transition into \mathbb{R}^+ .

The behaviour of a prioritised Petri net is now detailed, markings being those of the associated Petri net. Note that the firing rule is the same as for non-prioritised Petri nets, the priority scheme influencing only the enabling condition.

Definition 4 (Prioritised enabling rule)

- A transition $t \in T$ is priority enabled in marking M , denoted by $M[t]^\rho$, iff:
 - it is enabled, i.e. $M[t]$, and
 - no transition of higher priority is enabled, i.e. $\forall t' : M[t'] \Rightarrow \rho(M, t) \geq \rho(M, t')$.
- The definition of the priority function ρ is extended to sets and sequences of transitions (and even markings M):
 - $\forall X \subseteq T : \rho(M, X) = \max\{\rho(M, t) \mid t \in X \wedge M[t]\}$
 - $\forall \sigma \in T^* : \rho(M, \sigma) = \min\{\rho(M', t') \mid M'[\sigma]M' \text{ occurs in } M[\sigma]^\rho\}$.

In the definition of $\rho(M, X)$, the set X will often be the set T of all transitions, in which case the T could be omitted and we could view this as a priority of the marking, i.e. $\rho(M)$. The definition of $\rho(M, X)$ means that we can write the condition under which transition t is priority enabled in marking M as $M[t]^\rho$, or in the expanded form $M[t] \wedge \rho(M, t) = \rho(M, T)$. We prefer the latter form if the range of transitions is ambiguous.

If the priority function is constantly zero over all markings and all transitions, then the behaviour of a Prioritised Petri Net is isomorphic to that of the underlying Petri Net. With this in mind, the subsequent presentation only includes definitions of prioritised constructs — the non-prioritised versions can be deduced by setting the priority function to a constant zero value.

Note that we choose to define priority as a positive real-valued function over markings and transitions — the higher the value, the greater the priority. We could equally define priority in terms of a *rank function* which maps markings and transitions to positive real values, but where the smaller value has the higher priority. This would be appropriate, for example, if the rank were an indication of earliest firing time. Note that the dependence of the priority function on the markings (as well as the transitions) means that *the priority is dynamic*.

2.3 State Spaces of (Prioritised) Petri Nets

The *state space* (also named *occurrence graph*) of a Petri net is represented as a graph which contains a node for each reachable marking and an arc for each possible transition occurrence. Since state spaces are defined similarly for Petri nets without and with priorities, only the latter definition is given. The sole difference is whether there are priorities or not for the firing rule.

Definition 5 (State space of a prioritised Petri net)

Let $PPN = (P, T, W, M_0, \rho)$, be a prioritised Petri net. The Prioritised State Space of PPN is the directed graph $PSS = (V, A)$, where:

1. $V = [M_0]^\rho$ is the set of vertices.
2. $A = \{(M_1, t, M_2) \in V \times T \times V \mid M_1[t]^\rho M_2\}$ is the set of arcs.

Example: The Petri net in figure 1(a) is equivalent to the modular Petri net of figure 4. Its (full) state space is shown in figure 1(b). Note that the initial state is shown as A1B1C1, thus indicating that place A1 is marked with a token in module A, place B1 is marked with a token in module B, and place C1 is marked with a token in module C. In this initial state, only transition F1 is enabled, its occurrence leading to state A2B2C1.

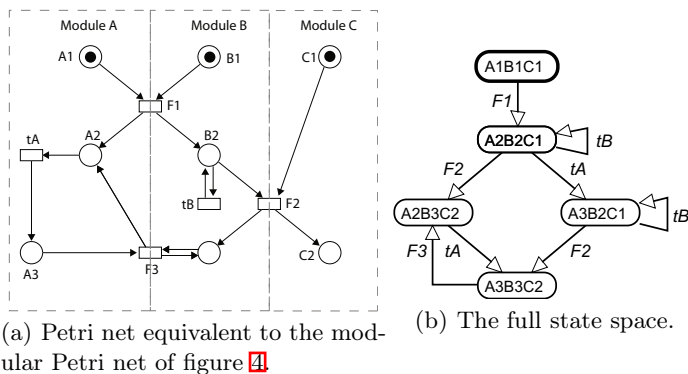


Fig. 1. The Petri net and state space of the system in figure 4

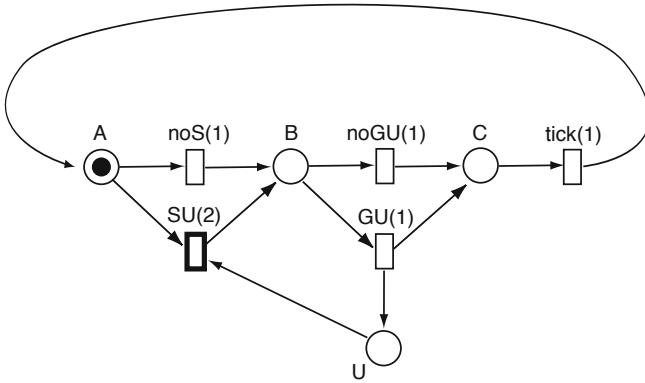


Fig. 2. Simplified Petri net for device message generation

Example: The Petri net in figure 2 is a simplified version of the one considered in more detail in figure 5. It captures part of the message-handling of a device, such as those used on a factory floor in the Fieldbus protocol [MSF+99]. The device cycles through states *A*, *B* and *C*. Place *U* holds one token for each urgent message that is waiting to be sent. At each cycle, the device can send an urgent message (if one is available) by firing transition *SU*, or it can choose not to send a message by firing transition *noS*. Similarly, in each cycle, it can generate an urgent message (by firing transition *GU*), or choose not to generate such a message by firing transition *noGU*.

The state space for this system is shown in figure 3. Here, the states are annotated with the places which hold a token, and place *U* is flagged with the number of tokens in the place. If the net is *not* prioritised, then the number of urgent messages can grow without limit, as indicated by the incomplete state space. If the transitions are prioritised (with the priorities shown in parentheses), then transition *SU* has higher priority than *noSU*, and the greyed-out part of the state space will be omitted.

This partial example only illustrates the value of a static priority scheme. The value of a dynamic priority scheme is shown in the extended example of figure 5.

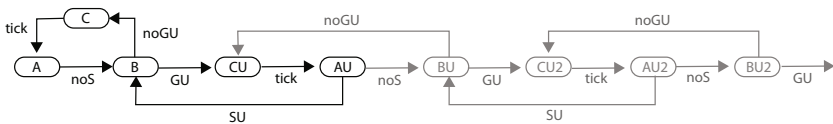


Fig. 3. State space for simplified Petri net for device message generation

¹ The state space for the modular prioritised Petri net of figures 5 and 6 is too large to be represented here.

2.4 Modular Petri Nets

Modular Petri nets are defined in a similar manner to Petri nets. Unlike the definitions of [CP00] we only consider communication through transitions.

Definition 6 (Modular Petri Net)

A modular Petri net is a pair $MN = (S, TF)$, satisfying:

1. S is a finite set of modules such that:
 - Each module, $s \in S$, is a Petri net:

$$s = (P_s, T_s, W_s, M_{0_s}).$$
 - The sets of nodes corresponding to different modules are pair-wise disjoint: $\forall s_1, s_2 \in S : [s_1 \neq s_2 \Rightarrow (P_{s_1} \cup T_{s_1}) \cap (P_{s_2} \cup T_{s_2}) = \emptyset]$.
 - $P = \bigcup_{s \in S} P_s$ and $T = \bigcup_{s \in S} T_s$ are the sets of all places and all transitions of all modules, and $W = \bigcup_{s \in S} W_s$ is the composite weight function defined on all arcs.
2. $TF \subseteq 2^T \setminus \{\emptyset\}$ is a finite set of non-empty transition fusion sets.

In the following, TF also denotes $\cup_{tf \in TF} tf$.

We now introduce transition groups for the actions of the Modular Petri Net, both simple and composite.

Definition 7 (Transition group). A transition group $tg \subseteq T$ consists of either a single non-fused transition $t \in T \setminus TF$ or all members of a transition fusion set $tf \in TF$. The set of transition groups is denoted by TG . The transition groups which consist only of transitions in a set $T' \subseteq T$ is denoted $TG|_{T'}$.

A transition can be a member of several transition groups as it can be synchronised with different transitions (a sub-action of several more complex actions). Hence, a transition group corresponds to a synchronised action. Note that all transition groups have at least one element.

Next, we extend the arc weight function W to transition groups:

$$\forall p \in P, \forall tg \in TG : W(p, tg) = \sum_{t \in tg} W(p, t), \quad W(tg, p) = \sum_{t \in tg} W(t, p).$$

Markings of modular Petri nets are defined as markings of Petri nets, over the set P of all places. The restriction of a marking M to a module s is denoted by M_s .

The enabling and occurrence rules of a modular Petri net can now be expressed.

Definition 8 (Modular Petri net firing rule)

- A transition group tg is enabled in a marking M , denoted by $M[tg]$, iff $\forall p \in P : W(p, tg) \leq M(p)$.
- When a transition group tg is enabled in a marking M_1 , it may occur, changing the marking M_1 to another marking M_2 , defined by $\forall p \in P : M_2(p) = (M_1(p) - W(p, tg)) + W(tg, p)$.

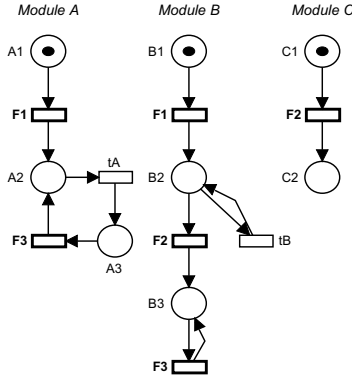


Fig. 4. Modular PT-net with modules A, B and C

Example: Figure 4 depicts a modular Petri net consisting of three modules A, B and C. Modules A and B both contain transitions labelled F1 and F3, while modules B and C both contain transition F2. These matched transitions are assumed to form three transition fusion sets.

2.5 Prioritised Modular Petri Nets

Similar to modular Petri nets, prioritised modular Petri nets can now be defined:

Definition 9 (Prioritised Modular Petri Net)

A Prioritised Modular Petri net is a tuple $PMN = (S, TF, \rho)$, where:

- (S, TF) is a Modular Petri net.
- ρ is the priority function mapping a marking and a transition into \mathbb{R}^+ .

Transition groups are defined as for non-prioritised Petri nets. They also have an associated priority function:

Definition 10 (Priority of transition groups). The priority function ρ is extended to transition groups by defining it to be the minimum priority of its elements, i.e. $\rho(M, tg) = \min_{t \in tg} \rho(M, t)$.

Note that the definition of the priority of a transition group is somewhat arbitrary. A simpler approach would have been to insist that all elements of a transition group have the same priority. This would mean that the priority allocations in one module would need to take account of the priority allocations in all other modules with which this one might synchronise. This seems excessively onerous in practical applications, especially since the priorities must agree for all reachable markings. The decision to define the priority of a transition group as the minimum over the elements has been guided by timed systems — if one transition in a group is enabled earlier than the others, then it must wait till the others are also enabled.

The arc weight function W is extended to transition groups, and markings are defined as for modular Petri nets. The firing rule in prioritised modular Petri nets takes into account the priority of transition groups.

Definition 11 (Priority enabling of transition groups). A transition group $tg \in TG$ is priority enabled in a marking M , denoted by $M[tg]^p$ iff:

- it is enabled, i.e. $M[tg]$, and
- no transition group of higher priority is enabled, i.e. $\forall tg' \in TG : M[tg'] \Rightarrow \rho(M, tg) \geq \rho(M, tg')$.

Example: Figure 5 depicts a module of a prioritised modular Petri net. It captures the message-handling of a device, such as those used on a factory floor in the Fieldbus protocol [MSF⁺99]. Messages are generated by the device — urgent messages (indicated by U) need to be processed in a timely manner, while normal messages (indicated by N) can wait, but not too long. The device cycles through states A, B, C and D . Place U holds one token for each urgent message that is waiting to be sent, with a maximum of 2 (to limit the size of the state space). The capacity can be imposed by a capacity constraint, or by the use of a complementary place. If place N is non-empty, then there is a normal message to be sent, and the number of tokens indicates how long it has been delayed — it is incremented each time transition $tick$ is fired. (This is shown as a double-ended, dashed arc between transition $tick$ and place N . The precise notation is not shown to avoid clutter and because it will depend on the specific kind of Petri net.) Again we impose an arbitrary capacity of 2 on this place.

Transition noS is local and indicates that no message is to be sent, while transitions SU and SN are fused to others in the environment (as indicated by the bold outlines) and denote that an urgent or a normal message is sent to some controller module. Transition $noGU$ indicates that no urgent message is

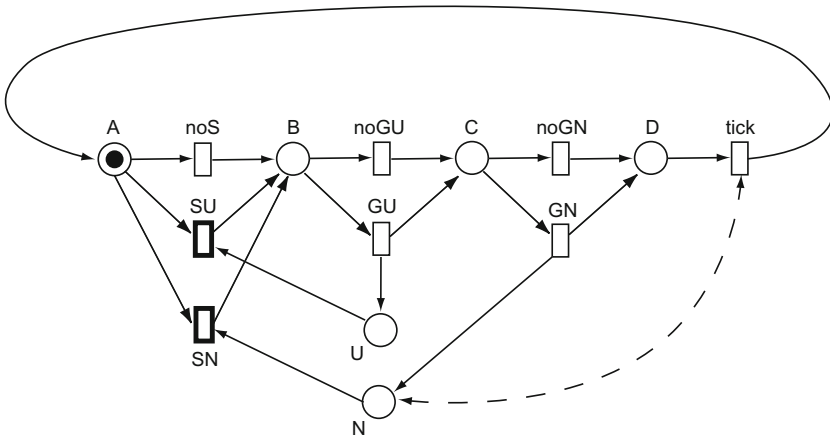


Fig. 5. Module of a prioritised PT-net for the message-handling of a device

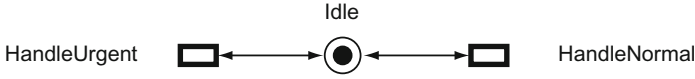


Fig. 6. Module of a controller to receive messages from the devices

generated in this cycle, while transition GU indicates that an urgent message is generated. Similarly, transitions $noGN$ and GN relate to the generation of normal messages. (Note that the diagram is again simplified to avoid clutter.)

We attach a priority of 1 to all local transitions. The priority of transition SU is set to the number of pending urgent messages plus 1, while the priority of transition SN is set to the number of ticks that the normal message has been waiting. Thus, if an urgent message is waiting and no normal messages, then transition SU has priority over noS . However, if the controller is not ready to receive the urgent message, then transition noS will fire. Similarly, if there is no urgent message but there is a pending normal message, then that message will be sent (by firing transition SN) if it has been waiting for more than 1 cycle, or it *may* be sent if it has been waiting only 1 cycle. Again, a normal message which has been waiting at least 2 cycles, may compete for processing with an urgent message which has only just been generated.

Figure 6 depicts a trivial controller for accepting the messages from a device. It has one transition to handle each of the urgent and normal messages, and we may assume that these transitions have the same priority.

3 Modular State Spaces

In the definition of modular state spaces, we denote the set of states reachable from M by occurrences of local (non-fused) transitions only, in all the individual modules, by $[[M]]$.

The notation with a subscript s means the restriction to module s , e.g. $[M]_s$ is the set of all nodes reachable from global marking M by occurrences of transitions in module s only (excluding fused transitions). We will also use lower case m to refer to the local marking of a module.

We use $M_1[[\sigma]]M_2$ to denote that M_2 is reachable from M_1 by a sequence $\sigma \in (T \setminus TF)^* TF$ of internal transitions, followed by a fused transition, i.e. $\sigma = \sigma' tf$ and $M_1[[\sigma']]M_1'[tf]$.

The definition of a modular state space consists of two parts: the state spaces of the individual modules and the synchronisation graph. We can now present the definition of modular state spaces for prioritised modular Petri nets. The definition of a modular state space for modular Petri nets given in [CP00] uses strongly connected components for optimisation and efficiency purposes. However, the computational benefits of using local strongly connected components are negated by the need for local activity to abide by the global priority function. Hence, strongly connected components are not used in prioritised modular state spaces. Therefore, we will avoid cluttering the paper with the definition of

modular state spaces (see [CP00, LP04]), and will directly formulate the prioritised version.

In order to be able to focus on the local context of an individual module, we need to have a localised priority function which is consistent with the global priority function.

Definition 12 (Consistency and locality of priority functions)

Let $PMN = (S, TF, \rho)$ be a prioritised modular Petri net.

- The priority function ρ is consistent iff $\forall s \in S : \forall t \in T_s : \forall M, M' : M_s = M'_s \Rightarrow \rho(M, t) = \rho(M', t)$.
- Given a consistent priority function ρ , we define local priority functions ρ_s as the projection onto the local marking, i.e. $\forall t \in T_s, M : \rho_s(M_s, t) = \rho(M, t)$.

Thus, with a consistent priority function, the priority of a local transition is determined solely by the local marking. If this were not the case, the modularity of the system would be seriously flawed, and the local state space could not be explored without reference to the global state of the system. If it were desired for a local transition to have a priority depending on some global state, then that transition ought to be synchronised with another transition having access to that state.

We can now define the modular state space for prioritised modular Petri nets.

Definition 13 (Prioritised modular state space). Let $PMN = (S, TF, \rho)$ be a Prioritised Modular Petri net with the initial marking M_0 . The prioritised modular state space of PMN is a pair $PMSS = ((PSS_s)_{s \in S}, PSG)$, where:

1. $PSS_s = (V_s, A_s)$ is the prioritised local state space of module s :
 - (a) $V_s = \bigcup_{v \in (V_{PSG})_s} [v]_s^{\rho_s}$.
 - (b) $A_s = \{(M_1, t, M_2) \in V_s \times (T \setminus TF)_s \times V_s \mid M_1[t]^{\rho_s} M_2\}$.
2. $PSG = (V_{PSG}, A_{PSG})$ is the prioritised synchronisation graph of PMN :
 - (a) $V_{PSG} = [[M_0]]^\rho \cup \{M_0\}$.
 - (b) $A_{PSG} = \{(M_1, (M'_1, tf), M_2) \in V_{SG} \times ([M_0]^\rho \times TF) \times V_{SG} \mid M'_1 \in [[M_1]]^\rho \wedge M'_1[tf]^\rho M_2\}$.

Explanation:

(1) The definition of the state space graphs of the modules is a generalisation of the usual definition of state spaces.

(1a) The set of nodes of the state space graph of a module contains all states locally reachable from any node of the synchronisation graph.

(1b) Likewise, the arcs of the state space graph of a module correspond to all priority enabled internal transitions of the module.

(2) Each node of the synchronisation graph is a representative for all the nodes reachable from M by occurrences of local transitions only, i.e. $[[M]]^\rho$. The synchronisation graph contains the information on the nodes reachable by occurrences of fused transitions.

(2a) The nodes of the synchronisation graph represent all markings reachable from another marking by a sequence of internal transitions followed by a fused transition. The initial node is also represented.

(2b) The arcs of the synchronisation graph represent all occurrences of fused transitions.

The state space graphs of the modules only contain local information, i.e. the markings of the module and the arcs corresponding to local transitions but not the arcs corresponding to fused transitions. All the information concerning these is stored in the synchronisation graph.

It is important to note that the above definition, in contrast to [CP00], introduces a disconnect between the local state spaces and the synchronisation graph. It is not immediately apparent how the computation of the local state spaces in Def. 13 part 1, and specifically of $[v]_s^{\rho_s}$, is used to compute the synchronisation graph in Def. 13 part 2 and specifically $[[M_1]^\rho$. This is a significant algorithmic issue which is addressed Section 4.

Example: The modular state space for the modular PT-net of figure 4 is shown in figure 7. Note that there is a local state space for each module, as well as a synchronisation graph which captures the occurrence of fused transitions.

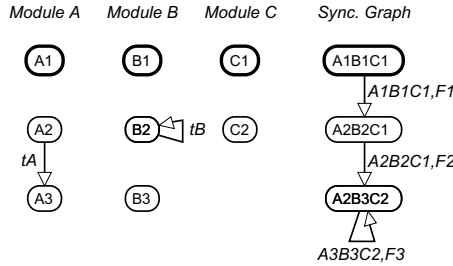


Fig. 7. The modular state space of the system in figure 4

In [Pet05], several experiments were conducted, showing that the size of the modular state space is significantly reduced (compared to the size of the flat state space) when the modules exhibit strong cohesion and weak coupling.

The efficiency gains achievable from modular state spaces arise from the ability to explore local state spaces (of modules) independently, and then combine them via the synchronisation graph to form a composite state space. If desired, a

² The example being a modular Petri net without priorities, the definition of modular state spaces (with strongly connected components) from [CP00] should be used. However, since in that particular case, strongly connected components always contain a single node, the modular state space is also obtained using Def. 13 without considering priorities.

full unfolded state space can be generated from the modular state space (as in Def. [14](#)), though it is computationally more efficient to analyse the system without enumerating all possible interleavings.

Definition 14 (Unfolded state space). *Given a prioritised modular Petri net $PMN = (S, TF, \rho)$ and its modular state space $PMSS = ((PSS_s)_{s \in S}, PSG)$, then the unfolded state space of $PMSS$ is $SS = (V, A)$ where:*

1. $V = \bigcup_{v \in V_{PSG}} [[v]^\rho$.
2. $A = \bigcup_{(v, (m, tf), v') \in A_{PSG}} \{(m, tf, v')\} \cup \bigcup_{\substack{m \in V, s \in S, (m_s, t, m'_s) \in A_s, \rho(m, t) = \rho(m, TG) \\ \text{where } m_s^*(p) = m_s(p) \text{ for } p \in P_s \text{ and } m_s^*(p) = 0 \text{ for } p \in P \setminus P_s.}} \{(m, t, (m + m'_s) - m_s^*)\},$

The above definition is similar to that of [CP00](#), except for the addition of priorities. Specifically, part [1](#) considers states reachable from v by transitions respecting the global priority function, and part [2](#) considers individual transitions satisfying the same constraint, captured as $\rho(m, t) = \rho(m, TG)$.

The theorem which states the equivalence of the above unfolded state space and the state space of the equivalent non-modular Petri net carries over with only minor changes since both state spaces reflect the priority scheme.

4 Algorithms

In general, modular state spaces can alleviate the state space explosion provided it is possible to construct the modular state space and determine properties based on this state space without needing to explore the possible interleavings of activity between multiple modules, i.e. without having to generate the unfolded state space of Def. [14](#).

With prioritised modular nets, this is less straightforward because the priority function imposes a global constraint on the behaviour of individual modules. The use of prioritised modular state spaces to determine system properties is the subject of further work. Here, we consider the construction of these prioritised modular state spaces.

The definitions of section [3](#) are consistent with the definitions of [CP00](#) but they hide a key computational issue — it is assumed that the computation of $[[M_1]^\rho$ in Def. [13](#) part [2](#) is supported by the computation of the local state spaces in part [1](#). (A similar comment applies to the computation of $[[v]^\rho$ in Def. [14](#) part [1](#).) In other words, the computation of local state spaces is assumed to help determine the global markings reachable from a synchronisation node by the firing of non-fused transitions alone. In the case of non-prioritised modular nets, this is straightforward — localised transition sequences from a synchronisation node can be interleaved in any order. With prioritised modular nets, the interleaving is constrained by the priority function. Accordingly, we

need to know whether the local state space, computed with the localised priority function ρ_s , contains *all* the information necessary to compute the interleaved sequences, and then we need to know how to compute such priority-respecting interleaved sequences in an efficient manner. These two questions are addressed in Lemma 1 and Proposition 1, respectively.

Lemma 1. *Given a prioritised modular Petri net $PMN = (S, TF, \rho)$ with a consistent priority function ρ , $M[[\sigma]^\rho M'$ implies $M_s[\sigma_s]_s^\rho M'_s$ for all $s \in S$, where σ_s is the restriction of σ to the internal transitions of module s .*

Proof. If one ignores the priorities, then it clearly follows that $M[[\sigma] M'$ implies $M_s[\sigma_s]_s M'_s$ for all $s \in S$. In other words, we can split the composite sequence into subsequences for each module. Now, if the priorities are taken into account, then the only way that the result would not hold is that one of the local sequences includes a transition which is not (locally) of maximum priority. But, if it is of maximum priority in the global sequence, then it must be of maximum priority in the local sequence (without fused transitions), in view of the fact that ρ is consistent. \square

A consequence of the lemma is that any priority-respecting transition sequence formed from non-fused transitions has its counterpart in local priority-respecting transition sequences of the individual modules. In other words, the local state spaces of Def. 13 part 1 contain all the information necessary to compute $[[M_1]^\rho$ in Def. 13 part 2, and sometimes even more information.

It is important to note that the above lemma means that transitions in local state spaces are provisional, in the sense that they will not necessarily appear in the unfolded state space. This is because their enabling in the unfolded state space depends on the priorities of transitions in other modules. On the other hand, transitions in the synchronisation graph do carry over into the unfolded state space, because these already consider global conditions.

Example: The (abbreviated) state space of Fig. 8 captures both the local state space for the *Device* module of Fig. 5 and the unfolded state space for the system consisting of the *Device* and *Controller* modules of Figs. 5 and 6. The states are encoded as a letter (to indicate which of the places A to D is marked), followed by the number of tokens in place U (which is the number of pending urgent messages), followed by the number of tokens in place N (which is the number of ticks that a normal message, if any, has been waiting). For example, state $C12$ means that place C is marked, there is one urgent message pending, and there is a normal message that has been waiting for at least 2 ticks. Note that dashed arcs indicate that the state is dealt with elsewhere (to avoid many crossing arcs).

For the local state space of the *Device* module, the synchronised transitions SU and SN of Fig. 8 would not be included — they would only appear in the synchronisation graph. Further, based solely on local information, we cannot be sure whether the transitions shown with dotted arcs will be preempted or not. Thus, in state $A01$, the sending of the normal message competes at the same priority as the transition *not* to send a message. On the other hand, in state $A02$,

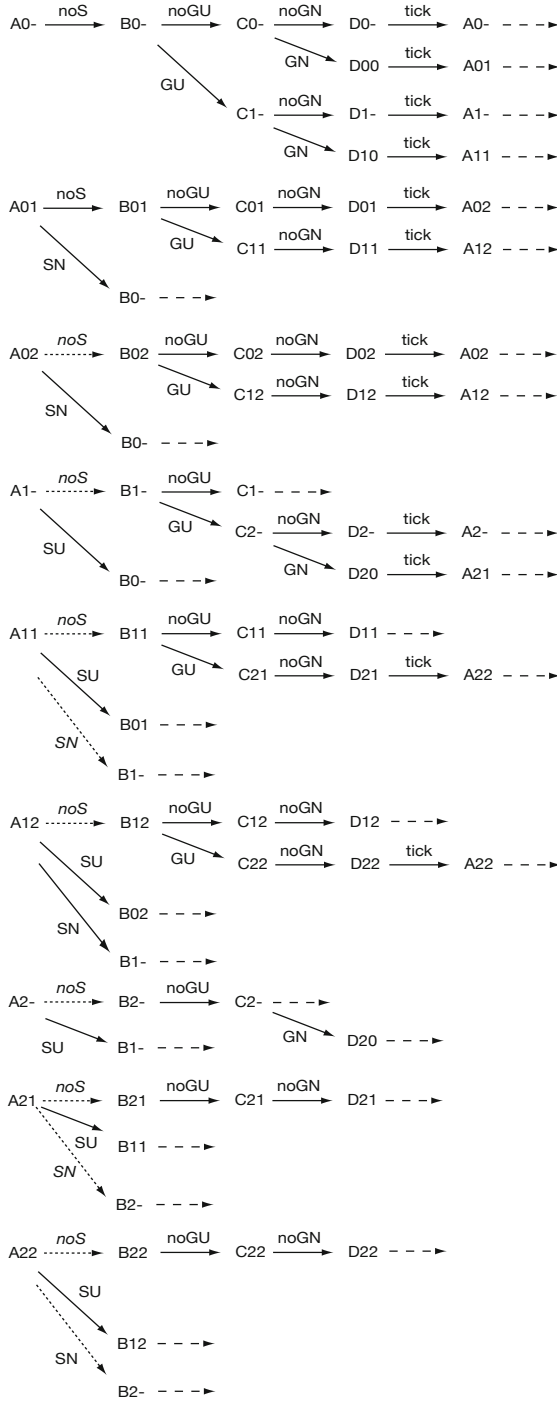


Fig. 8. Local state space of a prioritised PT-net

transition SN is of higher priority, but its enabling depends on the enabling of the fusion partners, and hence the alternative noS needs to be included as well.

The fusion of the *Controller* net with the *Device* net leads to the same net structure except for the addition of the place *Idle*. Hence the unfolded state space of the composite system is as shown in Fig. 8 except that the dotted arcs (with italic annotations) are omitted because global knowledge of the priorities allows us to deduce that these transitions are preempted by others of higher priority.

Lemma 1 showed that the local state spaces of Def. 13 part 1 capture all the behaviour required to compute $[[M]^\rho$ of part 2. We now identify a property that allows it to be computed in an efficient manner.

We first introduce some auxiliary terminology:

Definition 15. *Given a prioritised modular Petri net $PMN = (S, TF, \rho)$, and given a local execution sequence $m[\sigma]_s^{\rho_s} m'$ in module s , m' is a synchronisation point if m' priority enables the local component of a fused transition, i.e. $\exists f \in T_s \cap TF : m'[f]_s^{\rho_s}$. Intermediate synchronisation points are (potential) synchronisation points in σ prior to m' . A realised synchronisation point is one that is matched by appropriate synchronisation partners.*

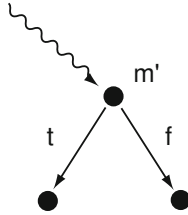


Fig. 9. A synchronisation point

The above definition may be clarified by the example of Fig. 9. Having arrived at local marking m' (which is part of a global marking M'), we may find both a local transition t and a transition f , part of a fused transition tf , enabled. If $\rho_s(m', f) < \rho_s(m', t)$, then f cannot be priority enabled whatever the situation with the synchronisation partners. (Recall Def. 10 where the priority of a transition group is the minimum of the priorities of the constituent transitions.) Alternatively, if $\rho_s(m', f) \geq \rho_s(m', t)$, then f is priority enabled locally but tf may not be priority enabled globally. This may be because the synchronisation partners do not enable their component of the fused transition at the same time, or because the priority of the transition group is decreased so that $\rho(M', tf) < \rho_s(m', t)$.

Proposition 1. *Given a prioritised modular Petri net $PMN = (S, TF, \rho)$, and given local execution sequences $\sigma_1, \sigma_2, \dots$ such that $M_1[\sigma_1]_1^{\rho_1} M'_1, M_2[\sigma_2]_2^{\rho_2} M'_2, \dots$ then there is a composite execution sequence σ with $M[\sigma]^\rho M'$ if $\exists s \in S : \rho_s(M_s, \sigma_s) = \min\{\rho_1(M_1, \sigma_1), \rho_2(M_2, \sigma_2), \dots\} \geq \rho(M', TG|_{T \setminus T_s})$, provided no preempting intermediate synchronisation points of $\sigma_1, \sigma_2, \dots$ are realised in σ .*

Proof. The local sequences can be combined using a simple merge algorithm — at each step, the transition at the head of the local sequence with highest priority is removed and added to the composite sequence. The choice is only non-deterministic when multiple local sequences have, as head, transitions with the same maximum priority. In this case, choose any one *not* in module s . Note that the non-deterministic choice only affects the order of interleaving and not the final reached marking.

Recall from Def. 3 that the priority of a sequence is the minimum priority of its constituent transitions. So, if the marking M' has priority less than or equal to the priority of all the local sequences, then the transitions of *every* local sequence will be added to the composite sequence *before* any transition enabled in marking M' . If this is not the case, we can still allow a module to have a sequence with minimum priority less than $\rho(M', TG)$, provided that that sequence enables the relevant transitions in marking M' , i.e. those with priority $\rho(M', TG)$. \square

Corollary 1. *In determining whether a fused transition is enabled, we only need to know the priority of the sequences leading to the synchronisation point and not the sequences themselves.*

We are normally interested in merging local sequences only when it comes time to consider whether a fused transition is enabled. The above proposition and corollary mean that we need only consider the priority of the local sequences leading from one synchronisation point to the next, together with the priorities of the associated sequences.

Further, if we have a priority scheme where the priorities are non-increasing, then the priority of a sequence is given by the priority of the last element of the sequence.

The above formal results provide the foundation for the following algorithms to compute the modular state space. These algorithms are refined versions of those presented in [LP07] for Modular Timed Petri Nets.

Algorithm 1 computes the synchronisation graph, while algorithm 2 computes the local state space for module i . Both follow the common pattern of maintaining a set of as-yet unexplored markings, called **Waiting**. Each iteration of the main loop explores the transitions enabled in these markings, adds new arcs to the relevant state space with the function `ARC.ADD(...)`, and adds new nodes to the state space and to **Waiting** with the function `NODE.ADD(...)`.

In algorithm 2 the local markings are stored together with their preceding synchronisation point — the notation $M'_i \{ \}^p M''_i$ represents local marking M''_i which is reached from a preceding synchronisation point M'_i with a local transition sequence of priority p . For consistency, a zero length sequence has priority ∞ . Lines 8–11 consider the local components of fused transitions — if they are (locally) enabled, then they are added to the eventual result $trysynch_i$ and the marking is identified as a synchronisation point. Lines 12–20 then consider internal transitions. The first alternative deals with the situation where the transition enabling is *not* dependent on the realisation of a synchronisation point, while in the second alternative, the enabling is dependent on such a realisation, and

Algorithm 1. Prioritised synchronisation graph.

```

1 set Waiting  $\leftarrow \emptyset$ ;
2 NODE.ADD( $M_0, \infty$ );
3 repeat
4   forall  $(M, p) \in \text{Waiting}$  do
5     Waiting  $\leftarrow \text{Waiting} \setminus \{(M, p)\}$ ;
6      $\forall i : \text{trysynch}_i \leftarrow \text{EXPLORE}(S_i, M_i)$ ;
7     forall  $tf \in TF$  do
8       forall  $M'$  s.t.
9          $\exists i : (tf \cap T_i \neq \emptyset \wedge$ 
10            $\forall j : (\sigma_j = M_j \rangle^{q_{j1}} M_{j1} \rangle^{q_{j2}} \dots \rangle^{q_{jn}} M_{jn} \subseteq \text{trysynch}_j \wedge$ 
11              $((tf \cap T_j \neq \emptyset \wedge M_{jn} = M'_j \wedge M'_j \langle tf) \wedge$ 
12                $\rho_j(\sigma_j) \geq \rho_i(\sigma_i) \wedge \rho(M', tf) \geq \rho_j(M'_j, T_j \setminus TF)) \vee$ 
13                $(tf \cap T_j = \emptyset \wedge M'_j \in [M_{j,n_j}]_j \wedge \rho_j(M'_j) \leq \rho_i(\sigma_i) \leq \rho_j(\sigma_j))) \wedge$ 
14                $\rho_i(\sigma_i) \geq \rho(M', TG|_{T \setminus T_i}) \wedge$ 
15               no preempting intermediate synch points are realised)
16         do
17           if  $M' \langle tf \rangle M'' \wedge \rho(M', tf) = \rho(M', TF)$  then
18             NODE.ADD( $M'', \min(p, \rho(M', tf))$ );
19             ARC.ADD( $M \langle (M', tf) \rangle^{\rho(M', tf)} M''$ );
20           endif
21         endforall
22       endforall
23     endforall
24 until stable ;

```

hence the new marking is paired with this synchronisation point. The result of a call to $\text{EXPLORE}(S_i, M_i)$ is a set of candidate synchronisations which record the preceding synchronisation point and the priority of the transition sequence leading from one to the other.

In algorithm [1](#), the central loop (in lines 8–22) tries to match up the candidate synchronisations so that they satisfy the condition of proposition [1](#). It considers a subset of the elements returned by each call to $\text{EXPLORE}(S_i, M_i)$, treating them as a composite sequence from local marking M_i to local marking M'_i with all intermediate synchronisation points identified. In detail:

- Line 9 identifies one of the synchronisation participants — as we shall see below, it is the one with minimum priority sequence.
- Line 10 considers the sequences of local transitions for all modules. In an abuse of terminology, a set of abstract edges from the module is concatenated together to form a sequence, which we then refer to as σ_j which technically should be the sequence of transitions (without the intermediate markings).
- Lines 11–12 consider modules participating in the synchronisation — the end of the returned sequence enables tf locally, the priority of the sequence is greater than that from module i , and at the end of the sequence, no local transition will (necessarily) preempt the firing of tf .

- Line 13 considers modules *not* participating in the synchronisation — they reach an end point which can lead locally to a marking compatible with the synchronisation, i.e. where the module will wait for the synchronisation.
- Line 14 requires that the minimum priority of module i is greater than the other priorities at the synchronisation point, i.e. so that module i can catch up.
- Line 15 requires that no intermediate synchronisation points are realised, the intermediate synchronisation points being the markings identified in the sequences at line 10. We can determine whether any of these intermediate synchronisation points are realised by applying the same logic as above.

Algorithm 2. Prioritised local state space — EXPLORE(S_i, M_i).

```

1 set Waitingi ← ∅;
2 set trysynchi ← ∅;
3 NODE.ADD( $M_i \uparrow^\infty M_i$ );
4 repeat
5   forall ( $M_i \uparrow^p M_i''$ ) ∈ Waitingi do
6     Waitingi ← Waitingi \ {( $M_i \uparrow^p M_i''$ )};
7     synchpt ← false;
8     forall  $tf \in TF \cap T_i, M_i''[tf], \rho_i(M_i'', tf) \geq \rho_i(M_i'', T_i \setminus TF)$  do
9       trysynchi ← trysynchi ∪ {( $M_i \uparrow^p M_i'', tf$ )};
10      synchpt ← true;
11    endforall
12    forall  $t_i \in T_i \setminus TF, M_i''[t_i]M_i'''$ ,  $\rho_i(M_i'', t_i) = \rho_i(M_i'', T_i \setminus TF)$  do
13      if  $\neg \text{synchpt} \vee \rho_i(M_i'', t_i) = \rho_i(M_i'', T_i)$  then
14        NODE.ADD( $M_i \uparrow^{\min(p, \rho_i(t_i))} M_i'''$ );
15        ARC.ADD( $M_i''[t_i]^{\rho_i(t_i)} M_i'''$ );
16      else
17        NODE.ADD( $M_i'' \uparrow^{\rho_i(t_i)} M_i'''$ );
18        ARC.ADD( $M_i''[t_i]^{\rho_i(t_i)} M_i'''$ );
19      endif
20    endforall
21  endforall
22 until stable ;
23 return trysynchi

```

5 Results

The *Maria* tool [Mäk02] was extended with dynamic priorities along the lines of the algorithms in Section 4. Here, we consider some of the results produced with this prototype implementation. The results were produced on a Mac Pro with two 2.66 GHz dual-core Intel Xeon processors and 2 GB memory. Note that this is not a complete implementation but it is sufficient to provide a proof of concept.

Table 1. State space results for device message handling

Devices	Modular state space				Unfolded state space			
	Nodes	Arcs	Sec	KB	Nodes	Arcs	Sec	KB
1	7	42	0.009	2.8	39	58	0.005	3.7
2	39	48	0.061	11.4	1441	3482	0.074	100.2
3	343	6,174	0.565	101.9	51,304	156,609	3.643	4,205.5
4	2,401	57,624	5.412	912.6	178,2011	6,264,028	196.140	164,439.5
5	16,807	504,210	51.856	7,902.5	—	—	> 3320	> 2,570,457.13

A simple example of message-handling for devices, such as those used on the factory floor in the Fieldbus protocol [MSF⁺99], was introduced in Section 2. The state space sizes and machine resource requirements are shown in Table 1.

On the left are the results for the modular state space for between 1 and 5 devices. The number of nodes and arcs are the figures for the synchronisation graph, while the time and space requirements (in seconds and kilobytes) are for the construction of the entire modular state space. The size of the state space for each module is 39 nodes and 48 arcs. This is similar to the flat state space for 1 device. Note, however, that the local state space does *not* include the fused transitions, while the flat state space will record their occurrence.

On the right of the table are the results for a flattened system, i.e. the unfolded state space. Note that memory was exhausted for 5 devices.

It might be argued that if the devices were identical, then symmetry reduction would probably give similar, if not better, results. However, if the devices were *not* identical, then the modular state space exploration would still be effective.

6 Conclusions

The modular state space technique [CP00] proves to give good results in practical cases [LP04, Pet05] to alleviate the state space explosion problem. This technique is designed to handle nets where modules communicate through transition fusion, i.e. synchronise. The technique is particularly efficient when the system modules exhibit strong cohesion and weak coupling.

In practice many systems use some kind of priority mechanism, either implicit or explicit. This is the case when there are timing constraints or when some events should be executed before others, once they are enabled (e.g. in scheduling problems). The priority used may either be static, i.e. it is fixed for a given transition and will never change during the life of the system, or dynamic, meaning that it does not depend solely on the transition involved, but also on the current marking.

In this paper, we have first introduced modular Petri nets with dynamic priorities and then adapted the modular state space technique to these nets. This involved defining the priority for synchronisation transitions in a way which is consistent with both practical and theoretical concerns. The resulting modular

state space with priorities contains more states than necessary since part of the preemption due to priorities cannot be known *a priori*. Some preliminary results have been generated from a partial implementation. These results provide a proof of concept for the proposals. A fully-fledged implementation is required to produce more extensive comparative results.

One could consider other alternatives for the priority of synchronisation transitions. Motivated by a consideration of timed systems, we set the priority of synchronisation transitions to be the *minimum* of the priorities of the constituent transitions. Instead, it could be set to the *maximum*. In this case, the preemption rules would change — if the local component of a synchronisation transition had priority greater than that of a local transition, then preemption would always occur. On the other hand, if the local component of a synchronisation transition had lesser priority than that of a local transition, then preemption may still occur. This change would not affect the propositions, but would require changes to the logic of the presented algorithms.

Further work is required on both theoretical and practical issues. We should study the use of prioritised modular state spaces to prove system properties. In order to verify some properties, it will be necessary to locally unfold part of the state space, so as to get rid of the spurious states, while other properties will be directly verified on the modular structure. Then, we intend to apply the technique to a large case study. An appropriate example is the avionics mission system from [PKBQ03], which includes both timing constraints and explicit priorities of tasks to be scheduled on CPUs within a certain time frame. Another example with both timing constraints and priorities is the Fieldbus protocol [MSF⁺99], which provided the motivating example of message handling in Section 2.

References

- [Bau97] Bause, F.: Analysis of Petri nets with a dynamic priority method. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 215–234. Springer, Heidelberg (1997)
- [CP00] Christensen, S., Petrucci, L.: Modular analysis of Petri nets. *The Computer Journal* 43(3), 224–242 (2000)
- [LP04] Lakos, C., Petrucci, L.: Modular analysis of systems composed of semiautonomous subsystems. In: Proc. 4th Int. Conf. on Application of Concurrency to System Design (ACSD 2004), Hamilton, Canada, pp. 185–194. IEEE Comp. Soc. Press, Los Alamitos (2004)
- [LP07] Lakos, C., Petrucci, L.: Modular state space exploration for timed Petri nets. *Journal of Software Tools for Technology Transfer* 9(3-4), 393–411 (2007)
- [Mäk02] Mäkelä, M.: Model Checking Safety Properties in Modular High-Level Nets. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 201–220. Springer, Heidelberg (2003)

- [MSF⁺99] Mnaouer, A.B., Sekiguchi, T., Fujii, Y., Ito, T., Tanaka, H.: Colored Petri nets based modeling and simulation of the static and dynamic allocation policies of the asynchronous bandwidth in the fieldbus protocol. In: Billington, J., Díaz, M., Rozenberg, G. (eds.) APN 1999. LNCS, vol. 1605, pp. 93–130. Springer, Heidelberg (1999)
- [Pet05] Petrucci, L.: Cover picture story: Experiments with modular state spaces. *Petri Net Newsletter* 68, Cover page and 5–10 (2005)
- [PKBQ03] Petrucci, L., Kristensen, L.M., Billington, J., Qureshi, Z.H.: Developing a formal specification for the mission system of a maritime surveillance aircraft. In: Proc. 3rd Int. Conf. on Application of Concurrency to System Design (ACSD 2003), Guimarães, Portugal, pp. 92–101. IEEE Comp. Soc. Press, Los Alamitos (2003)

A Problem Frame-Based Approach to Evolvability: The Case of the Multi-translation

Gianna Reggio, Egidio Astesiano, Filippo Ricca, and Maurizio Leotta

DISI, Università di Genova, Italy

{gianna.reggio,astes,filippo.ricca,maurizio.leotta}@disi.unige.it

Abstract. The problem frame approach allows to precisely pin the software development problems before starting to work on them, thus avoiding to solve the wrong problems. Furthermore, the problem frames allow to develop tailored methods and schematic solutions to handle the tasks required to solve the corresponding problems. In this paper we adopt this approach to study the problem of developing a large class of software systems able to translate in different ways some inputs in outputs (e.g., hybrid mail or big brothers filtering digital communications for suspicious words). Our interest in this kind of systems has been prompted by a cooperation with a big company producing systems of this kind and by their search of techniques and approaches to handle predictable and unpredictable changes. We want to investigate how and if the problem frame based approach will help to master the aspects relative to predictable and unpredictable changes in the context, in the domain and in the requirements. We thus present the Multi-Translation Frame.

1 Introduction

Evolvability, i.e., the ability to evolve software over time to meet the changing needs of its stakeholders, is one of the principal challenges currently facing software engineering¹. Software architecture decay over the years, aged programming languages, software written by other developers, all contribute to create what is typically an evolution nightmare. In the literature, this challenge has been faced in several ways. Among the several proposals, the more accredited is building from scratch the system using specific and rigorous techniques/methodologies for evolvability [4,6,17]. While we share with these authors the same forward engineering vision, we propose a *frame-driven development approach* [15] to cope with evolvability problems.

Our interest in evolvability has been prompted by a cooperation with a local big company producing various kinds of complex systems. One of them, is the hybrid mail system XYZ². XYZ is used by the postal organizations that offer

¹ Third International IEEE Workshop on Software Evolvability at IEEE International Conference on Software Maintenance (ICSM) Paris, France 1 October 2007
<http://homepages.feis.herts.ac.uk/~comqcln/EN/software-evolvability07.html>

² For privacy reason, we cannot report here the name of the system.

their customers — mainly big companies such as banks — specific services to produce big amounts of electronic/physical mail (e.g., invoices and bank statements) starting from electronic data files. Therefore, the main activities of a hybrid mail system are: receiving customer data in several formats (e.g., XML and PDF), processing them to produce the required mails and printing them (close to their final destinations). Afterwards, the produced physical mails are supplied to a logistics service for the delivery.

The main problem of this company is handling in reasonable time predictable and unpredictable changes in XYZ. Predictable changes can be forecasted by looking at the current domains and at the requirements of the system, instead unpredictable changes cannot be imagined (e.g., a new law changing the business rules of XYZ is promulgated).

In this paper we propose the motto “developing for change” to characterize a software development method able to cope with the changes that may be required in the future by the stakeholders. In some sense we try to enlarge the scope of the old motto “design for change” [20,19] trying to cover also the other phases and activities of the software development. Moreover, we investigate how and if a *frame-driven approach*, a cornerstone of our principle, will help to master the aspects relative to predictable and unpredictable changes. Following the indications of Jackson, who claims that *a clear understanding of requirements (the problem) is crucial to building useful systems, and that to evolve successfully, their design must reflect problem structure*, we have created a specific problem frame called Multi-Translation Frame (shortly MTF). The MTF offers the first help for structuring, abstracting and documenting and thus follows, during the development of a system, the “developing for changes” principle. Moreover, to better explain our proposal, we have used, as running example, a simple toy case called Toy-HMS. Toy-HMS is a toy example of a hybrid mail system and, at the same time, an instantiation of the MTF.

The paper is organized as follows. Sect. 2 describes both the MTF and its instantiation Toy-HMS, also showing the use of the frame to qualify the types of the possible changes. Sect. 3 illustrates our approach “developing for change”, based on the MTF, by sketching the Domain Modeling, the Requirement Specification and the Design development phase for the Toy-HMS and then showing how to cope with possible changes. Finally, Sect. 4 presents some related works and concludes the paper.

2 Proposal of a Specific Problem Frame

M. Jackson “Problem Frames” [15] are a good tool to tackle with a first structuring of software development problems. For each problem frame, a diagram is settled, showing the involved domains, the requirements, the design, and their interfaces. Five basic problem frames, plus some variants, have been originally provided by M. Jackson in [15]. Other problem frames have been recently proposed together with some extensions to the original notation for the frame [10,11]. Problem frames are also presented with the idea that, once the appropriate

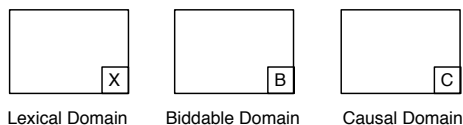
problem frame is identified, then the associated development method should be given “for free”, as shown in [10][11] where development methods based on the UML are provided for the various frames.

In problem frames presented by M. Jackson, there is a distinction between existing domains and the system to be built as a new part in that world. This implies that the various entities considered in the existing domains are not modified (or removed) when the new system is introduced.

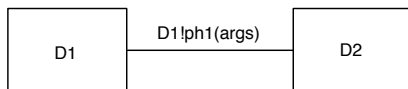
Following Jackson’s notation, different types of domains are used to represent the frames. As shown below, a machine domain (the software to be built) is denoted by a box with a double stripe, a designed domain (data structures or subsystems that may be freely designed and specified) by a box with a single stripe, and a given domain (a problem domain whose properties are given) by a box with no stripe. To help to grasp this classification consider these examples: the commands sent to the system controlling a dam can be changed up to some extent, e.g., by using different ways to name them or add shortcuts for special cases or combinations of commands and they were designed by someone, this is a case of designed domain. The dam is a given domain since in the dam control frame it is assumed that the dam is not going to be modified.



Letters in the lower right corner reflect a coarse classification of domains, orthogonal with the previous one: - *lexical* (data), *biddable* (people or systems, with no predictable internal causality), or *causal* (predictable causality, controlling and controlled by some phenomena).



A solid line connecting two domains is an interface of shared phenomena. Below, phenomena *ph1* is controlled by domain *D1* and is shared between domains *D1* and *D2*.

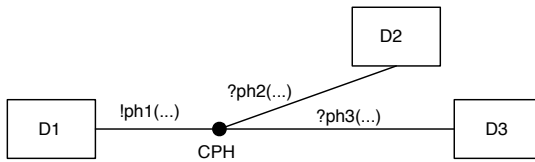


Requirements are denoted by a dashed oval. Dashed lines connect a domain and a requirement (constraining if with an arrow).



A frame diagram is a graph the nodes of which are domains, and its arcs are interfaces of shared phenomena. A *problem frame* is a particular frame including at least a machine and a requirement.

It is possible to connect several domains with an hyper-arc to denote a complex interaction built out of various basic phenomena, that we name composite phenomena [11]; in the picture below CPH is a composite phenomenon corresponding to a complex interaction among three domains, and involving the shared phenomena ph1, ph2 and ph3 (where D1 is responsible to initiate the interaction by means of the shared phenomena ph1).



2.1 MTF: A Problem Frame for Multi-translation

For the multi-translation we propose a specific problem frame called MTF. The MTF offers the first help for structuring, abstracting and documenting and thus follows the principle of “developing for changes”. Precisely, it provides: the separation between the translation rules and the dispatch rules, their explicit description and the possibility to find the commonalities among the producers/consumers/input/output data.

Fig. 1 presents the MTF using the M. Jackson notation [15] together with the extensions of one of the authors for the composite phenomena [11].

The input and the output data are given lexical domains, whereas the producers and the consumers are given biddable domains, but here for simplicity we omit to mark them with the letters in the lower right corners.

In the MTF there are various kinds of input data (domains ID_1, \dots, ID_n) generated by producers of various kinds (domains $Prod_1, \dots, Prod_s$) and of output data (domains OD_1, \dots, OD_k) that will be sent to consumers of different kinds (domains $Cons_1, \dots, Cons_r$).³ For simplicity we represent only a generic representative of these series of domains in Fig. 1.

The Multi-Translator is the machine (i.e., the system to be built, denoted by a box with a double stripe on the left); it receives some input data from some producers (complex interaction INPUT, whereas $send(idx)$ is a shared phenomenon

³ In the most general form of MTF, some input and some output data are given but the remaining ones are designed; for simplicity here we assume that all of them are given, but this distinction may be relevant for the coping with the changes, in such cases a richer version of the MTF should be developed.

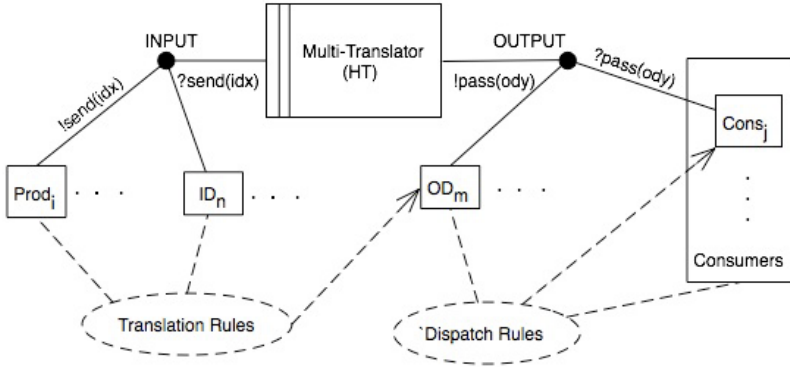


Fig. 1. Multi-Translation Frame (MTF)

having an argument typed by ID_n and $Prod_i$ has the responsibility to initiate the sharing) and gives out some output data to some consumers (complex interaction OUTPUT). In this frame the requirements are split in two parts: the **TranslationRules** that define the relationships between input data with their producers and the resulting sets of output data, whereas the **DispatchRules** determine to which consumer send the output data. In the picture a dashed line means that a domain is referred in the requirement (thus the **TranslationRules** refers also to the producers and to the input data, and since the current situation of existing producers may affect the choice of the consumers to whom send an output data, the **DispatchRules** refer to the **Consumers**), whereas a dashed line with arrow head means that the requirement may affect a domain (thus an output data may be affected by the **TranslationRules** and a consumers by the **DispatchRules**).

We assume that the number and the kinds of the producers and of the consumers may vary dynamically while the system is running (for example some new one may appear and some may disappear, but their types are already known); thus the context of the system may change and so the machine must be a context aware system, able to cope with the changes in the context.

A Hybrid Mail System (HMS) can be viewed as an instantiation of the MTF. As we have already said, HMSs are postal systems used to compose and print mails close to their final destinations⁴. These systems are called Hybrid Mail systems because transform mails/data given in “electronic” form to “physical/papery” mails. As particular case of MTF an HMS: (i) requires data (ID) to create mails, (ii) produces the mails ready to print (OD), (iii) executes some procedures to compose/create mails (**Translation Rules**), (iv) executes some procedures to dispatch mails to the correct printing center (**Dispatch Rules**). The input data are provided by the clients of the HMS (Producers) and mails are sent to different printing centers (Consumers).

⁴ Here we do not consider the hardware of these systems (e.g., machines used to print mails).

2.2 Toy-HMS: A Toy-Case Instantiation of MTF

Toy-HMS is a toy case of hybrid mail system and an instantiation of the MTF. Toy-HMS is a HMS used by only one company, i.e., there is a unique producer, that sends periodically mails to its customers. Toy-HMS has three *printing centers*: North-PC, Center-PC and South-PC (located in Milan, Rome and Naples), for the Northern, Central and Southern Italy respectively. Each printing center is used to print the mails sent to addresses of its geographic competence. All printing centers can print B/W, whereas North-PC and South-PC can print also in colour.

When a *clerk* of the company has to send mails to customers, (s)he submits the data in electronic mode to the Toy-HMS. The set of all required data is called *batch*. From a batch, Toy-HMS produces a set of mails ready to print and organizes the mails in groups (called *print batch*) to be sent in electronic mode to the three printing centers following some routing rules. All the mails of a batch follow a schema called *template* that defines the common structure of the mails.

Toy-HMS is suitable for handling advertising letters, notices etc, i.e., mails which differ from each other only for the address and the name of the customer.

In Fig. 2 we present the instantiation of the MTF for the case of the Toy-HMS.

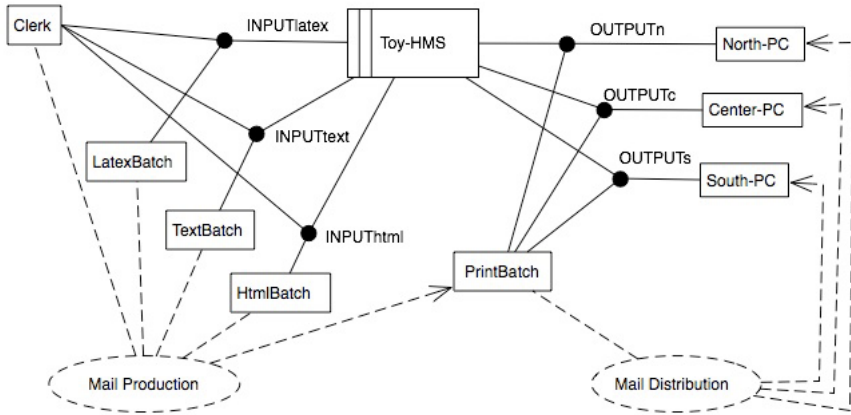


Fig. 2. Multi-Translation Frame (MTF) instantiated for the Toy-HMS case

In this case we have a unique producer (the Clerk) and three consumers (North-PC, Center-PC and South-PC), three types of input data (LatexBatch⁵, TextBatch and HtmlBatch) and a unique type of output data (PrintBatch).

Any batch is composed of the print type required (B/W or colour), the list of the addresses of the receivers, and the template. A template represents the model for each mail of the batch, and consists of a file in the proper format (either Latex or HTML or Text), where there are some place-holders corresponding to the parts of a correct address.

⁵ <http://www.latex-project.org/>

A print batch consists of a PDF file containing the generated mails, the indication of whether it requires colour printing, and a ZIP Code (all the mails in a print batch are sent to addresses with that code).

The requirements of the Toy-HMS are given by the MailProduction and the MailDistribution.

The MailProduction requires to control and correct up to some extent the address list, to divide the address list in sub address lists one for each ZIP Code, then to compose the mails using the template and the addresses in the various sub-lists (the templates are assumed to be always well formed), and finally to pack them in various print batches consisting of mails sent to the same ZIP Code.

The MailDistribution requires to send the B/W print batches to the nearest printing center since all the three of them can print B/W, and to send the colour print batches to the nearest printing center chosen between North-PC and South-PC (those that can print in colour).

To determine which is the closest printing center a function distance between the print batch ZIP Code and the printing center ZIP Code is used. This function is defined taking into account a large set of factors, including the availability of routes, railways, motorways and these factors have been already considered when the ZIP Codes were defined in Italy.

To validate Toy-HMS, we have implemented a SOA-based Java prototype. Moreover, we have compared the architecture of the prototype with the actual architecture of the hybrid mail system XYZ [18].

2.3 Relating the Variety of Changes to MTF

The MTF also offers the possibility to describe and classify the possible changes of the Multi-Translator system that its developer may have to cope with. We classify the changes in three broad categories: changes in context, proactive standard changes and unpredictable changes.

Changes in context: The number and the features of the producers and of the consumers connected to the system may change while the system is running (new producers/consumers may be added, however their type belongs to an existing list; existing producers/consumers may be eliminated; some producer/consumer may change its features, but the modification is relative to an existing list of features, whose possible forms are already known). These are the possible changes that affect the context of the system Multi-Translator (context intended as the entities interacting with the system itself).

Example of changes of this kind in the case of Toy-HMS are: a printing center breaks down and it is not available till it will be repaired, a new B/W printing center with the same characteristics of the existing ones is built in another Italian city (for example Palermo or Venice), and the printing center in Rome becomes able to print also in colour.

Proactive standard changes: These are changes that can be forecasted by looking at the domains and at the requirements of the MTF. We name these changes “proactive” since the developers should be proactive and propose them to the client as a way to increase their business, and “standard” since they are specific of the MTF. Here we illustrate two typical sample categories.

- New kinds of “derived” functionalities are required for the **Multi-Translator**. A derived functionality is a new one that just requires to collect or to elaborate information already computed for the existing functionalities, or just to log the performing of some activities. These changes modify the requirements enriching them in a conservative way. Some examples of changes of this kind in the case of **Toy-HMS** are the request to: – produce a report associated with each batch listing information on the sent paper mails, – collect data on the number and types of the processed batch for the accounting department, – record the number of wrong address files.
- The **Multi-Translator** system should perform simplified variants of its functionalities; for example a client provides pre-elaborated input data that do not require to perform the initial elaborations, or a producer requires to receive some intermediate data skipping some of the final elaborations. This kind of changes should lead to a nice simplification of the requirements and the update of the system should be very easy, but in practice they may result in hard work in case of wrong architectural choices.

Example of changes of this kind in the case of **Toy-HMS** are the request to: – skip the control and the correction of the address files since the addresses are surely correct, – send to the printing center a print batch in the form of a Latex or HTML source file or as a text file, instead of a PDF file (the PDF file generation will be performed by the printing center), – receive as input data some print batches so that only the dispatching activities will be performed.

Unpredictable changes: The unpredictable changes are those that cannot be imagined by examining the current domains and requirements of the frame. For example, completely new kind of producers or consumers or completely new kinds of input and output data may appear, or new laws or rules that disrupt the current requirements may become in effect.

Example of changes of this kind in the case of **Toy-HMS** are: – the appearing on the market of new printers (e.g., printers using paper of different weighs); – the emergence in the market of the private email services that completely change the rules for determining the most convenient printing center (e.g., someone following a policy where everything goes first to Rome); – the appearing of a new document format (e.g., docx).

Obviously, in this case a socio-economic analysis may help to get a rough idea on which changes of this kind are more likely in the future (e.g., in the case of Echelon the char set used for the English language is not going to change very soon, whereas a change of the kinds of communication devices to monitor is highly probable).

3 Developing for Change

In this section we outline the approach “developing for change” that we have mentioned in the introduction; an approach characterizing a software development method able to cope with the changes that may be required in the future by the stakeholders. With respect to the old motto “design for change” [20,19], we try to cover also the other phases and activities of the software development. Obviously this is the exact contrary of other current approaches (e.g., extreme programming [5]), where not even the design for changes is recommended, and obviously a cultural/technological/business analysis has to be performed initially to see whether the system to be developed will be a long lived one and whether it will ever have to cope with changes (e.g., chess games have not changed too much in the last centuries, whereas taxing mechanisms change continuously).

In this section we show how we can use the MTF to develop its instantiations in a way that makes easier and more efficient tackling the need for changes that may arise during and after the development. The general guidelines that we propose are cast within a development method based on the rigorous well-founded usage of the UML as notation for expressing all the required artifacts, see, e.g., [1,2,3].

3.1 Key Principles in Developing for Change

To make our treatment more understandable and convincing, we recall in this subsection some key principles for good design, that are preliminary to cope with changes. We will refer frequently to those principles in the following sections and thus we label them for an easy reading without repetitions. Most of them are classical, but in our approach they will be used not only at design level, but everywhere.

KeyP1. Structure and document everything, not just the code. It is not sufficient to document the code or the detailed architecture of some modules, but all the aspects of the system should be documented, e.g., also the producers and the consumers, and the rationale behind the dispatch rules should be properly documented; and obviously everything should be structured to allow the human reader to grasp it.

KeyP2. Provide high-level presentation/documentation. It not sufficient to attach a lot of comments to the code or to some architectural very detailed schema or to complex XML definitions of data structures; rather, all the key concepts of the system should be presented in a brief and compact way avoiding too many technical details. For example, a large part of the documentation should be understandable by the business expert (e.g., the core of the dispatch rules should not be hidden in a small stored procedure in a database written in a proprietary language, even if well commented; instead it should be presented by a set of conditional rules written using well structured natural language referring to business entities).

KeyP3. Explicit (abstract) typing, i.e., give a precise abstract type to each entity used in the system. Consider the following example, a fundamental data

structure, as an input data, should be explicitly defined by giving the operations for acting on it, and then by saying how it will be realized using the chosen technical means. For example it cannot happen that a fundamental data structure is implicitly realized by some XML files and some records in two different databases without no way to see which modules are using it and how; it becomes abstract typing when put together with the next principle.

KeyP4. Encapsulation, i.e., give a precise type/interface to each part of the system and of the software, as data structures, modules, components, functionalities and external subsystems (e.g., the support offered by an external subsystem should be presented by means of a function/procedure with a precise type and well defined meaning; any external device should have a high-level interface to be accessed, it is not sufficient to say that to pass an input to some external systems one has to put a file starting with some lines written using some cryptic codes in some specific point of the file system).

KeyP5. Separate commonalities from particular aspects in any part of the system not only in the data types, and this requires also to organize the data types and the other parts in explicit specialization/refinement structures (e.g., the various kinds of output devices may be very conveniently presented as a specialization hierarchy having an abstract device at the top; a group of requirements may be nicely organized in a hierarchy making explicit the common parts and the variants).

KeyP6. Use generative techniques, namely try to use high-level presentation of data structures and function that can be automatically transformed into the corresponding code (e.g., provide conditional rules that may be automatically transformed into code, instead of writing directly such code; provide a BPMN diagram [9] to be transformed into BPEL codes instead of writing directly some Java code to orchestrate some Web services).

3.2 Guidelines for a MTF-Based Development

We assume to have instantiated the MTF having determined who are the producers, the consumers, the input and the output data and which are the translation and the dispatch rules.

We note from the beginning that the use of the frame leads to apply *KeyP1*: indeed it obliges to provide a first documentation clarifying the four parts of the frame and a first initial structuring (e.g., the organization in terms of the domains parts of the frames and the separation of the translation requirements from those concerning the dispatching).

Domain Modelling. The domains in the MTF are the input/output data and the producers/consumers.

It will be highly probable that the various input and output data, the producers and the consumers share some commonalities; thus, following *KeyP5*, we look for specialization relationships among domains, possibly by introducing “abstract domains”, where domain has to be intended in the sense of Jackson [15], and to try to encapsulate them (*KeyP4*).

In our approach (see, e.g., [12,3]) the domains are modelled by means of UML classes. Thus the domain model in the case of the MTF is a UML class diagram with a class for each kind of producer, consumer, input and output data present in the frame. The classes corresponding to the input/output data, which are lexical domains using Jackson’s terminology, are static whereas those corresponding to the producers/consumers (biddable domains) are active classes with an associated behaviour, which may be modelled by a state machine. The producers/consumers must have an operation corresponding to the send/pass a data (shared phenomena with the Multi-Translator). It is important to fully model the data classes whereas it may happen that the model of the producers/consumers may be quite underspecified.

The fact that the various domains should be modelled by UML classes is in agreement with *KeyP2*, *KeyP3* and *KeyP4*. In Fig. 3 we show the Domain Model for the Toy-HMS case. In this diagram the class *Clerk* represents the producer type and the class *PrintCenter* the consumer type. The input data is represented by the class *Batch* specialized in *LatexBatch*, *TextBatch* and *HtmlBatch*. The output data is represented by the class *PrintBatch*. The other classes in the diagram (e.g., *TextFile*) are used for typing the attributes.

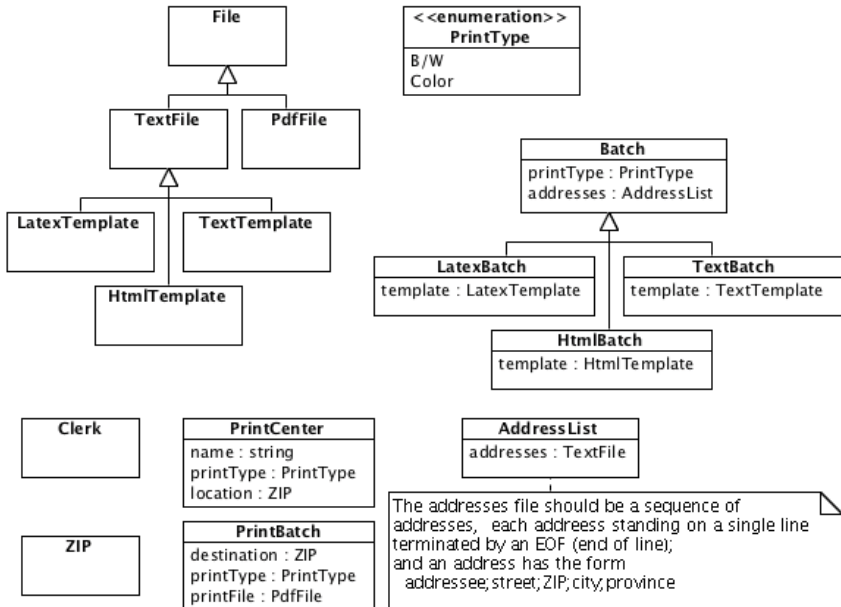


Fig. 3. Toy-HMS case: the domain model

Requirement Specification. In agreement with *KeyP1*, the requirements in the case of the MTF are already split in two parts: the *TranslationRules* and the *DispatchRules*, where the *TranslationRules* should associate a producer and an

input data with a set of output data, and the `DispatchRules` should associate an output data with a consumer.

Technically the `TranslationRules` should define a family of partial functions, called translation functions,

$$trans_k : ID_i \times Prod_x \longrightarrow Set(OD_j) \quad k = 1, \dots, p$$

where $trans_k(id, p) = ods$ means that the translation of id sent by p will result in ods , whereas $trans_k(id, p)$ undefined means that in that case the translation failed⁶.

The `TranslationRules` will be modelled by an abstract UML class, with the same name, having a static operation for each translation function. Obviously such functions should share some common aspects, otherwise the system would be just an aggregation of single-translation subsystems performing) and we should make explicit such commonalities; technically the translation functions should be expressed by composing/combining a set of functions corresponding to basic translation blocks (we call them *translation blocks*).

The decomposition of the translation functions in translation blocks is a form of structuring (*KeyP1*, while providing a precise type to the translation functions and the translation blocks is an application of *KeyP3*).

In the UML model the translation blocks will be modelled by other static operations of the same class `TranslationRules`. The effort of decomposing the translation functions and of extracting the translation blocks and the classes/types needed to type their parameters and their results will be the application of the principle of decomposing and structuring (*KeyP1*) and of high level documentation (*KeyP2*); indeed all these operations should have meaningful names and should be properly documented.

The operations corresponding to the translation operations and to the translating blocks will be defined using the means offered by the UML, such as OCL constraints, methods, and activity diagrams. For example, an operation may be modelled textually in the UML giving its (purely functional) body, or visually by means of an activity diagram, with matching input and output parameters where the basic activity are some translating blocks.

The `DispatchRules` should define a family of partial functions

$$dispatch_h : OD_h \times Set(Consumer) \longrightarrow Consumer \quad h = 1, \dots, r$$

where *Consumer* is an abstract class having as specializations all the consumer classes, and $dispatch_h(od, cons) = con$ means that the dispatch of od when all the consumers $cons$ are available, will result in sending it to con (obviously $con \in cons$), whereas $dispatch_h(od, cons)$ undefined means that in that case the dispatch failed⁷.

The `DispatchRules` will be modelled by an abstract UML class, with name `DispatchRules`, having a static operation $dispatch_h$ for each dispatch function. Again these operations should be decomposed using some basic dispatch operations making explicit any existing commonalities among them.

⁶ For the moment we do not consider the error messages.

⁷ Again, for the moment we do not consider the error messages.

Every operation needed for modelling the `TranslationRules` and the `DispatchRules` may be defined at a varying degree of abstraction and thus the requirements may be as much abstract as needed. The rules may be quite complex and may be written in a way that will help cope with the future change requests.

Another application of *KeyP5* is to avoid duplications in the definition of `TranslationRules` and the `DispatchRules`, that is to avoid that the same function fragment be written several times (same as the hint for avoiding duplicated code). Using the UML this means to introduce auxiliary operations and classes to represent the duplicate fragment. There is no need to say that the same suggestions apply even more stringently if the translation definitions are utterly complex.

If no commonalities or just a little amount of them are found among the definitions of the various translations and dispatching functions, then we should consider if the problem is about the development of a unique product or just a collection of very loose related different products. For example in the Echelon case the part concerning the search of hot words in text should be common to filtering email, chats, social network walls and SMS, whereas the capability of extracting text from PDF and JPEG files will be again used in various of the filtering functions, which are the translation functions in this case. On the other side a product offering a bunch of file compressing operations based on totally unrelated techniques should not be a case of the MTF.

In Fig. 4 we present the requirements for the Toy-HMS case using the UML. To make the diagram more compact we have omitted the classes already described in the Domain Model in Fig. 3, and the definition of the various functions can be found in Appendix A.

Design. The design phase requires to produce the machine, in Jackson's sense, of the problem frame schema, that is the `Multi-Translator`. The software architecture of the `Multi-Translator` will be represented using the UML, where each component will be modelled by a class.

It is possible to provide various architectural schema and to compare them w.r.t. the support for changes (for example using SAAM, a strategy presented in [12,16]). Moreover depending on further information concerning the non-functional aspects, such as the dimension of the input and output data, the frequency of the arrival of data to translate and whether there are real time constraints on when to return the translated data, further architectures may be proposed. However, there are a few general constraints on the possible architecture to be able to cope with the changes, that we summarize below and visually present in Fig. 5.

We use the same notation of the Jackson's frame to present the hints on the architecture of the `Multi-Translator`, and we depict the machine domain icon with dashed lines to suggest that it is not a complete architectural definition. For simplicity in the picture we depict only some representatives of the various domains (input and output data, producers and consumers). The `Executor` is a



Fig. 4. Toy-HMS case: the requirements (classes of the domain are not repeated)

machine (denoted using the Jackson’s notation by the double stripe on the left), the other domains denoted with two lines on the left are designed domains.

The lexical designed domain **Consumers** (composed of: name, printType, location) is a representation of the current situation of the consumers (which are the current available ones and their relevant features), which should be recorded in a persistent way inside the **Multi-Translator**, and there should be the means to keep it updated. Thus we have an **Operator**, a biddable given domain that will take care of updating the consumers, (s)he will know when new consumers will start/stop to operate or will change their characteristics and (s)he will interact with the **Multi-Translator** to update this information. Providing the domain **Consumers** is an application of *KeyP3*: we must give an explicit type structure for representing the consumers. To define this data structure we should look at the requirements to discover what is relevant of the consumers for the dispatching (for Toy-HMS just the type and the location of the printing center).

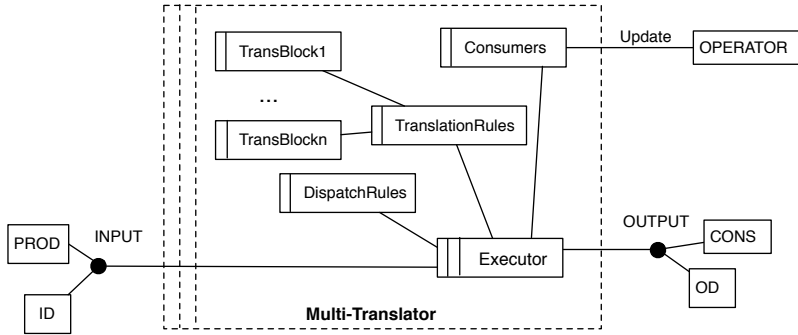


Fig. 5. Multi-Translation Frame: assumption on the design

Then, there will be some parts corresponding to the translating blocks, the translating functions and the dispatching functions appearing in the requirements. They are given domains and may be lexical or causal, indeed they may be sub-machines able to perform such functionalities or descriptions of such functions that another component will be able to execute. Thus we do not fix the kind of domain corresponding to them.

According to *KeyP6* (generative techniques), the translation blocks, the translations functions (and the dispatch rules and their basic sub-functions) are the ingredients that the executor uses to put together the translations to apply to the input data (for example, they may be: – code written in some domain specific languages, – compiled procedure written in some programming language, – software components or services). Thus the Multi-Translator has not some parts that are able to perform the various translations, but it includes an engine (the Executor) able to execute in a very general sense the definition of the various functions, in some sense it will generate the translation functions starting from the various ingredients.

3.3 Coping with Changes in the Multi-Translation Frame

Now we consider, as examples, some of the most probable changes in the MTF and see how the proposed method will help to cope with them.

Changes in context. The considered changes in the context are the appearing and disappearing of producers/consumers and changes in their features. The presence of the Consumers domain, and of the Operator that takes care to update it, allows the Multi-Translator to cope with the modifications in its context. Thus is quite easy to cope with this changes, and the proposed solution is a classical one.

Proactive standard changes. *Simplified versions of a translation.* The simplification may require to drop some of the transformations and/or to receive as input data what was before an intermediate data or to dispatch what was

before an intermediate data. The transformation just requires to introduce as new input and output data the previous intermediate data; it is needed to check that they are suitable for the associated consumers or that they may be sent by the producer.

Unpredictable changes. *Appearing of a new type of input data (without modifications in the output data and in the dispatch rules).* It is highly probable that the new kind of input data will be modelled by a subdomain of the existing ones or that it will be modelled by reusing parts of the existing ones (i.e., already existing UML classes), and so it will be easy to give its model. Then we have to define the corresponding translation functions (one for each possible producer of this kind of input data) reusing the existing translation blocks; it may happen that we have to introduce new translation blocks and/or classes to type their arguments and results. Thus, the modification of the Multi-Translator will just amount to possibly add some new translating blocks and a new values to the translation rules, whereas the Executor is not changed.

Other changes of the same type (e.g., new consumers or new kind of consumers, new dispatch rules) can be handled in the same way.

Disappearing of a type of input data. The input data is eliminated by the domain, and we have to check whether there was some producers producing it and investigating whether they have to be deleted as well. At the requirement level the corresponding translation function will be eliminated, then it has to be checked whether there are some translation blocks used only by itself, in this case also they are to be eliminated together with the relative types. Finally, we have to check if the eliminated translation function was the unique one to produce some kind of output data, that have in turn to be eliminated, and as last steps we need to investigate if there are some consumers that were receiving only such output data, that have to be deleted in turn. Doing this chain of deletions we have to do several checks helping to detect possible inconsistencies in the proposed change. At the Multi-Translator level, it is sufficient to eliminate the useless translation blocks and translation functions, and this is not a difficult operation, since all of them are items of the proper kind.

4 Related Work and Conclusions

To the best of our knowledge, this is the first work that proposes a *frame-driven development approach* able to cope with evolvability problems in a specific domain.

Among the software development methodologies specific for evolvability we can cite, e.g., [4,6,17]. Bastani in [4] presents a new analytical framework, called “Abstraction-oriented Frames” and a method specifically designed to support the requirements analysis and design of open evolvable software systems. To meet flexibility in changing requirements, architecture and design at any phase, the requirements framework presented in that paper consists of dynamics that facilitate subsystem modifications at any level of abstraction or any part of the

system. Another framework able to cope with evolvability problems is presented in [17] by HP. That framework, named ORBlite, provides a substrate that allows systems to be composed of components that can evolve independently over time. ORBlite has been successfully used by HP to build several evolvable real systems. Instead, authors in [6] propose and explain “Goal sketching”: a simple technique used for requirements engineering purposes. This technique starts with the creation of a goal graph which expresses the high level motivations behind the intention to develop the software. Then, a series of developments are planned using the goal graph as a guide (similarly to use Scrum sprints [21]). Authors claim that Goal sketching can be successfully used to develop evolvable systems. Finally, Schmidt [21] presents arguments in favour of implementing evolvable software systems using the SOA paradigm [13]. Indeed, SOA provides an extraordinary mechanism for supporting the evolution and rapid response to change in business rules.

Even if several proposals have been advanced by researchers in this direction, Brcina et al. [7] claim that existing software methodologies do not provide sufficient support for managing the evolvability. For this reason, they present a meta-model based and goal oriented process for controlling and optimizing the evolvability of a given software system. Breivold [8] in his PhD thesis introduces a method for analyzing software evolvability at the architecture level. More precisely, he identifies some sub-characteristics (e.g., Testability) that are of primary importance for an evolvable software system, and outlines a software evolvability model that provides a basis for analyzing and evaluating the evolvability of a given system. Instead Shiri et al. [22] present a novel approach to estimate the effort of potential modification and retesting associated with a modification request, without the need of analyzing or understanding the system source code. The approach is based on Use Case Maps and concept analysis [14].

In this paper we have presented a problem frame based approach to study the problem of developing a large class of software systems able to translate in different ways some inputs in some outputs (e.g., hybrid mail or big brothers filtering digital communications for suspicious words). In particular we have shown the use of problem frames in the case of the **Multi-Translator** with a simple instantiation, the **Toy-HMS**. Also, we have shown that the problem frame approach helps to produce systems that are able to handle predictable and unpredictable changes according to the new coined motto “developing for changes”.

Even if our method may seem too simple for realistic big systems, we believe that is usable and effective. For this reason, we want to apply it in future to obtain a new evolvable version of XYZ (the real huge system of interest of the company that has prompted our investigations). In future works, the capability to cope with predictable and unpredictable changes of the new **HMS** will be empirically analyzed and compared with the actual one using a “what if approach”, i.e., looking at how easy/hard will be to cope with a given list of real change requests following the Software Architecture Analysis Method (SAAM) proposed in [12,16].

References

1. Astesiano, E., Reggio, G.: Knowledge Structuring and Representation in Requirement Specification. In: Proceedings 14th International Conference on Software Engineering and Knowledge Engineering, pp. 143–150. ACM Press, New York (2002)
2. Astesiano, E., Reggio, G.: Towards a well-founded UML-based development method. In: 1st International Conference on Software Engineering and Formal Methods, Brisbane, Australia, September 22-27 (2003)
3. Astesiano, E., Reggio, G.: Tight Structuring for Precise UML-Based Requirement Specifications. In: Wirsing, M., Knapp, A., Balsamo, S. (eds.) RISSEF 2002. LNCS, vol. 2941, pp. 16–34. Springer, Heidelberg (2004)
4. Bastani, B.: A requirements analysis framework for open systems requirements engineering. SIGSOFT Software Engineering Notes 32(2), 47–55 (2007)
5. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change, 2nd edn. Addison-Wesley, Boston (2004)
6. Boness, K., Harrison, R.: Goal sketching: Towards agile requirements engineering. In: Proceedings of International Conference on Software Engineering Advances, Cap Esterel, pp. 71–77 (2007)
7. Brcina, R., Bode, S., Riebisch, M.: Optimization process for maintaining evolvability during software evolution. In: IEEE International Conference on the Engineering of Computer-Based Systems, pp. 196–205 (2009)
8. Breivold, H.P.: Software Architecture evolution and software evolvability, PhD thesis. Mälardalen University Press Licentiate Theses No. 97 (2009)
9. Business Process Management Initiative (BPMI). Business Process Modeling Notation (BPMN) (2004), <http://www.bpmn.org/Documents/BPMN%20V1-0%20May%203%202004.pdf> (website, last access December 13, 2007)
10. Choppy, C., Reggio, G.: A UML-Based Approach for Problem Frame Oriented Software Development. Journal of Information and Software Technology (2005)
11. Choppy, C., Reggio, G.: Requirements capture and specification for enterprise applications: a UML based attempt. In: Han, J., Staples, M. (eds.) Proc. ASWEC 2006, pp. 19–28. IEEE Computer Society, Los Alamitos (2006)
12. Clements, P., Bass, L., Kazman, R., Abowd, G.: Predicting software quality by architecture-level evaluation. In: Proceedings of the Fifth International Conference on Software Quality, Austin, Texas, vol. 5, pp. 485–497. Software Engineering Institute (1995)
13. Erl, T.: SOA Principles of Service Design. The Prentice Hall Service-Oriented Computing Series from Thomas Erl (2007)
14. Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations. Springer, New York (1997)
15. Jackson, M.: Problem Frames: Analyzing and Structuring Software Development Problems. Addison-Wesley, Reading (2001)
16. Kazman, R., Abowd, G., Bass, L., Clements, P.: Scenario-based analysis of software architecture. IEEE Software 13(6), 47–55 (1996), doi:10.1109/52.542294
17. Moore, K.E., Kirshenbaum, E.R.: Building Evolvable Systems: The ORBlite Project (1997), <http://www.hpl.hp.com/hpjournal/97feb/feb97a9.pdf> (website, last access November 7, 2010)
18. Leotta, M., Ricca, F., Reggio, G., Astesiano, E.: Comparing the maintainability of two alternative architectures of a postal system: SOA vs. non-SOA. In: Proceedings of 15th European Conference on Software Maintenance and Reengineering (CSMR 2011), Oldenburg, Germany, March 1-4, pp. 317–320 (2011), DOI: <http://www.computer.org/portal/web/csd1/doi/10.1109/CSMR.2011.41>

19. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM 15, 1053–1058 (1972)
20. Parnas, D.L.: Software aging. In: Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, pp. 279–287. IEEE Computer Society Press, Los Alamitos (1994)
21. Schmidt, V.A.: Using service oriented architecture for evolvable software systems. In: 11th International Command and Control Technology Symposium (2006)
22. Shiri, M., Hassine, J., Rilling, J.: A requirement level modification analysis support framework. In: IEEE International Workshop on Software Evolvability, pp. 67–74 (2007)

A The Translations and Dispatches Blocks for Toy-HMS

Function	Signature	Description and Example
addressControl	AddressList \longrightarrow boolean	checks whether the addresses are correct with respect of the format ForInd and each address is separated from each other by a end-of-line. ForInd format: addressee;street;ZIPCode;city;province
addressCorrection	AddressList \longrightarrow AddressList	examine the input address list correcting various mistakes. For example, in the two addresses below there are two mistakes; the first presents an incoherence between the city and the province; the second presents a misspelling in the city name: Mario Rossi;Via Verdi, 23;00100;Roma;MI Mario Bianchi;Via Verdi, 23;00100;Romw;RM The two addresses are corrected in: Mario Rossi;Via Verdi, 23;00100;Roma;RM Mario Bianchi;Via Verdi, 23;00100;Roma;RM
addressSplitting	AddressList \longrightarrow Sequence(AddressList)	groups the addresses that present the same ZIP Code in separated AddressList, one for each ZIP Code.
latexComposition	AddressList \times LatexTemplate \longrightarrow PrintBatch	creates a PDF file containing the mails created joining the Latex template with the data within the address list (notice that however the print type attribute of the produced print batch is not defined)
htmlComposition	AddressList \times HtmlTemplate \longrightarrow PrintBatch	creates a PDF file containing the mails created joining the HTML template with the data within the address list (notice that however the print type attribute of the produced print batch is not defined)
textComposition	AddressList \times TextTemplate \longrightarrow PrintBatch	creates a PDF file containing the mails created joining the Text template with the data within the address list (notice that however the print type attribute of the produced print batch is not defined)
distribution	PrintBatch \times Set(PrintCenter) \longrightarrow PrintCenter	PrintCsOk = { PC \in PrintCs PC.printType = PB.printType } return PCselect s.t. PCselect \in PrintCsOk and distance(PC.location,PB.destination) = min{ distance(X.location,PB.destination) X \in PrintCsOk}
distance	ZIP \times ZIP \longrightarrow Float	return the distance between the two ZIP

Towards a Framework for Modelling and Verification of Relay Interlocking Systems

Anne E. Haxthausen

DTU Informatics, Technical University of Denmark, DK-2800 Lyngby, Denmark
ah@imm.dtu.dk

Abstract. This paper describes a framework currently under development for modelling, simulation, and verification of relay interlocking systems as used by the Danish railways. The framework is centred around a domain-specific language (DSL) for describing such systems, and provides (1) a graphical editor for creating DSL descriptions, (2) a data validator for checking that DSL descriptions follow the structural rules of the domain, (3) a graphical simulator for simulating the dynamic behaviour of relay interlocking systems, and (4) verification support for deriving and verifying safety properties of relay interlocking systems.

1 Introduction

This paper describes the visions of an ongoing project made in collaboration with Banedanmark (Rail Net Denmark), and gives a survey of a framework of tools being developed within this project.

Background and motivation. A conventional means of keeping the railway traffic safe is to use interlocking systems that control signals and points in such a way that trains are only allowed to pass a signal when this cannot lead to train collisions or derailments. Many Danish interlocking systems are still implemented using complex electrical circuits containing relays. These relay based interlocking systems are documented by diagrams of the electrical circuits, and currently the only way to analyse them is to inspect the diagrams and manually draw conclusions. This is very difficult to do as the number of diagrams for a single system is very high and the logic described in each of them is complicated with many mutual dependencies. Certainly such a manual analysis is not only difficult and time consuming, but may also be error prone. This is not satisfactory for a safety-critical system. To help this, we started a project, the goal of which is to formalise and automate the validation and verification process for relay interlocking systems.

Solution approach. Our solution to the above mentioned problem is to provide a framework of computer-based tools that support the validation and verification process. The tools should be centred around a domain-specific language for expressing the documentation that is usually made for relay interlocking systems,

e.g. track layout and relay circuit diagrams. The idea is that to analyse or verify a relay interlocking system the railway engineer should express the documentation of the relay interlocking system in this domain-specific language, and the framework should provide tools that can be applied to such documentation to analyse and verify the documented relay interlocking system. Prototypes of such tools have already been developed. We have chosen to centre the tools around a domain-specific language rather than a general purpose modelling language, as it is easier for railway engineers to use a language that facilitates concepts already known and used in the railway domain.

Related work. The author and Jan Peleska have developed a framework of tools for the construction and verification of tramway control systems [15]. These tools are also centred around a domain-specific language, however the language and tools are different from those described in this paper. The differences are due to the fact that the controllers in that work are electronic, while ours are implemented using relay circuits. For instance, the tool set described in this paper provides a simulator for the electrical behaviour of relay circuits (which is not relevant for electronic systems) while the other tool set provides a control software generator (which is not relevant for relay systems¹).

For other complementary and competing approaches for the development and verification of railway control systems the reader is e.g. referred to the contributions in [20], and for a survey of results and trends the reader is referred to the paper [5].

Paper overview. First, in Section 2, we describe the railway application domain. Then, in Section 3, we give an overview of the tools framework, and in the subsequent sections we describe the domain-specific language and each of the tool components in more detail. In section 8 we mention how such a framework can be formally developed using the RAISE Formal Method [19]. Finally, in section 9 we draw some conclusions and describe ideas for future work.

2 The Railway Application Domain

In this section we introduce concepts of the railway domain that are relevant for this paper.

2.1 Equipment at a Station

The considered interlocking systems use a variety of track-side equipments to monitor and control trains:

Track circuits: The railway tracks are divided into sections each having equipment (a circuit) for train detection. The interlocking system uses this for monitoring the occupancy status of the individual track sections.

¹ For relay systems the equivalent to a control software generator would be a relay circuit design generator. However, as the purpose of our work is only to support the analysis and verification of relay circuits and not the creation of these, such a generator has not been considered.

Points: Tracks are joined by points² which can guide trains into different directions depending on the position of the points. The interlocking system monitors and controls the positions of points. An operator can switch the points³.

Signals: Signals are placed at the entrance of some track sections. They can show GO and STOP aspects. The interlocking system sets the signals to inform the train drivers whether they are allowed to enter these sections.

Figure 1 illustrates the interactions between the operator, the interlocking system, the trains, and the track side equipment.

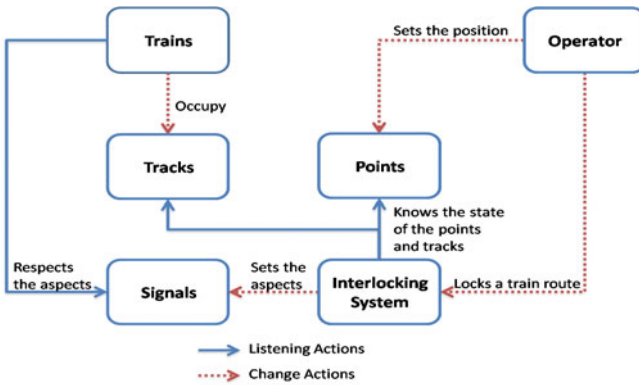


Fig. 1. Relationships between different elements of a station

2.2 Train Routes and Train Route Tables

The stations we are considering in this paper use a *route based* approach to interlocking. The basic ideas of this approach are:

- Trains should drive on *routes* through the network.
- Each route is covered by an entrance signal that informs whether it is allowed for a train to enter the route or not. The trains are assumed to respect the signals.
- Two trains must never be allowed to drive on conflicting (i.e. overlapping) routes at the same time. (*To prevent collisions.*)

² Following the UIC (International Railway Union) railway dictionary available from www.uic.org, this paper uses the term point to denote the assembly of rails, blades and of auxiliaries of which some are movable. In other dictionaries the following terms are used for the same: a set (or pair) of points, a switch and a turnout.

³ In some cases a centralised traffic control system is used to switch the points, but even when this is the case, the operator should still be able to switch the points, e.g. in emergency cases.

- Before a train is allowed to enter a route, the points in the route must be locked in positions making the route connected (i.e. it is physically possible to go from one end of the route to the other end without derailling), and the route must be empty (i.e. there are no trains on the route). (*To prevent derailling and collisions, respectively.*)
- The points of a route must not be switched while a train is driving on the route. (*To prevent derailling.*)

For each station to be controlled by an interlocking system, a *train route table* is used to specify routes and interlocking rules for that station. Such tables define for instance for each train route

- which settings of signals are required for the route to be *opened* (i.e. for allowing trains to enter the route),
- which positions points must have for the route to be *connected*,
- which track sections must be unoccupied for the route to be *empty*, and
- the conditions for unlocking/releasing a route.

The tables also define which train routes are *conflicting*.

2.3 Relay Circuits

The interlocking systems we are considering are implemented by electrical relay circuits. In [13] a formal domain-model for relay circuits is presented. Here we just give an informal description.

A relay circuit is made up of components such as power supplies, relays, contacts, lamps inside signals, and buttons, connected by wires. A *relay* is an electrical switch operated by an electromagnet to connect or disconnect a number of contacts in a circuit. When current flows through the relay, the magnet is *drawn* and some of the associated contacts are connected (these contacts are said to be *upper contacts*) while others (the *lower contacts*) are disconnected. When no current flows through the relay, the magnet is *dropped* and the associated upper and lower contacts will be disconnected and connected, respectively. When contacts are connected/disconnected this may imply that sub-circuits containing these contacts become live/dead. This again may imply that relays of these sub-circuits are drawn or dropped, and so on.

The system can get input from the environment:

- Buttons can be pushed (and later released) by an operator.
- For each track section there is a (track) relay that is dropped/drawn when a train enters/leaves that track section.
- For each point there are two associated (point) relays. One of these is dropped or drawn when that point is moved into a new position.

The track relays and point relays are said to be *external*, while the relays controlled by the interlocking system are said to be *internal*.

2.4 Relay Circuit Diagrams

The Danish railways use diagrams to document the electrical circuits of a relay system.

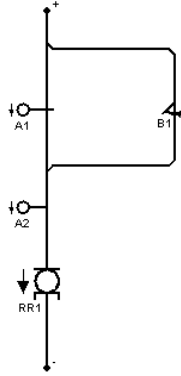


Fig. 2. Diagram for circuit controlling relay *RR1*

For each internal relay one of the diagrams shows the sub-circuit that controls that relay. An example of such a diagram is shown in Figure 2. This diagram shows the sub-circuit controlling a relay named *RR1*. The circuit consists of a number of components connected by wires. The wires are depicted as black lines. At the top is the positive pole and at the bottom is the negative pole of the power supply. Relay *RR1* is shown using this signature:



Signatures for relays contain a circle that may be decorated with some bars. (See relays 33 and 37 in Figure 6 for other examples of such decorations.) The decorations only indicate the logical purpose of the relay – they do not indicate any special physical behaviour. The downwards arrow to the left of the circle informs that in the initial state this relay is dropped. (If it had been drawn the arrow would have been upwards.)

A number of contacts belonging to other relays occur in this circuit. E.g. a contact belonging to a relay named *A1* is shown using this signature:



Generally the signature of a contact is a small version of the signature of the relay to which it belongs. The downwards arrow informs that in the initial state relay *A1* is dropped. The horizontal bar breaks the wire – this indicates that the contact is disconnected in the initial state. If the bar had not been breaking the wire it would have indicated that the contact had been connected in the initial state, as it is the case for relay *A2*. Also a button *B1* is shown on the diagram using this signature:



This signature informs that in the initial state this button is released. A pushed button is shown by this signature:



3 Framework Overview

As mentioned in the introduction our goal is to provide a framework of tools for analysing and verifying relay interlocking systems. In this section we give an overview of this framework.

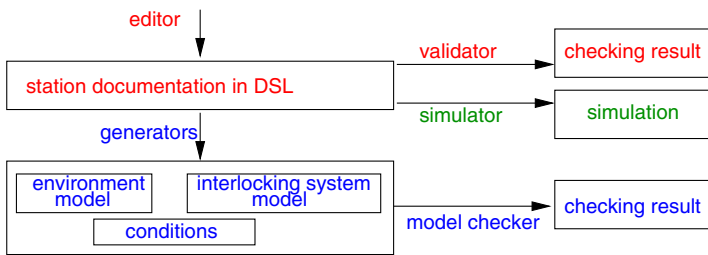


Fig. 3. Framework of tools

To make the framework user friendly for railway engineers we exploit the idea to define and centre the tools around a *domain-specific language*, *DSL*, for expressing the documentation that is usually made for the relay interlocking system of a station. The tools we are developing comprise:

- a (graphical) *editor* for creating DSL descriptions,
- a *validator* for checking that a DSL description follows static (structural) rules of the domain,
- a (graphical) *simulator* that for a given DSL description can simulate the dynamic behaviour of the described interlocking system, and
- *generators* that from a DSL description produce input to (a front end of) the SAL model checker [3]:
 - a behavioural (state transition system) model of the described interlocking system,
 - a behavioural (state transition system) model of the described environment (track isolations, points, and operator buttons), and
 - safety conditions and other kinds of conditions (expressing desired properties) formalised in the temporal logic LTL.

The model checker can then be applied to this to verify that the interlocking system always satisfies the safety conditions and other desired properties.

Figure 3 illustrates this framework of tools. The language and tools will be further described in the next sections.

4 Domain-Specific Language

A *specification D* in *DSL* consists of the following station documentation:

- a *track layout diagram* describing the physical environment,
- a *train route table* specifying the interlocking rules, and
- *relay circuit diagrams* describing the physical implementation.

In Figures 4 and 5 are shown the track layout diagram and the train route table for Stenstrup station (unfortunately in Danish). In Figure 6 a simplified version of one of the circuit diagrams for Stenstrup is shown. (There are too many, too large circuit diagrams for Stenstrup to show them here.)

A detailed explanation of how to read the track layout diagram and table is given in 6. Here we will only explain those details that are needed for understanding the remaining of this paper.

The track layout diagram shows the geographical arrangement of the tracks and track-side equipment such as track circuits, points, and signals. From the diagram it can be seen that Stenstrup has six track circuits (named A12, 01, 02, 03, 04, and B12), three points (named S1/S2, 01 and, 02), and eight signals (named a, b, A, B, E, F, G, and H).

The train route table has one row for each train route. For each route the **Togveje** sub-columns contain basic information about the train route such as its identification number and whether it is an entry or exit route from/to which other station, the **Signaler** sub-columns state the signal settings (**gr** means green/GO and **re** means red/STOP) required for the route to be open (signals that should show red must be set before the signals that should show green are set), the **Sporskifter** sub-columns state required positions of points (+ means straight position and - means branching position) for the route to be connected (and possibly also flank protected), the **Sporisolationer** columns state with an ↑ which track sections must be unoccupied for the route to be empty, the **Ovk** column concerns level crossings (not to be discussed in this paper), the **Stopfald** column specifies that a certain signal (the entry signal of the route) should be switched to a STOP aspect when a certain track section (the first section of the route) becomes occupied, the **Togvejsopl.** columns define conditions (not to be explained here) for when the train route can be released, and the **Gensidige spærringer** marks with the symbol ◦ which routes are conflicting.

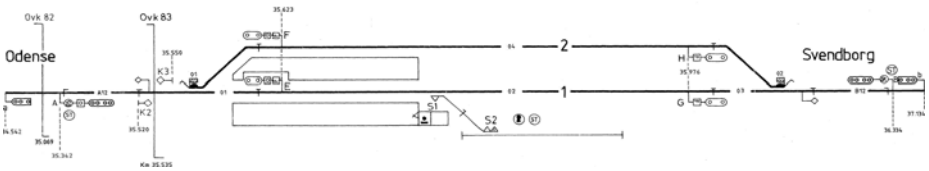


Fig. 4. Track layout for Stenstrup station

Togveje		Signalier		Sporskifter		Sporsolationer		Dvk		Stop		Togvejsopt.		Gensidige spæringer													
nr		Spærforløb	a	b	A	B	E	F	G	H	01	02	04	03	1012	02	03	03	1012	02	03	03	Indt.	Opt.			
2	fra Odense	Indk	1	strækkn	sp ¹	sp ²	ra	ra ¹		+	aff	↑	↑	↑	↑	↑	↑	↑	Ja	Ja	A	↓A12	↓A11	↓A10	2		
3		Indk	2	strækkn	sp ¹	sp ²	ra	ra ¹		+	aff	↑	↑	↑	↑	↑	↑	↑	Ja	Ja	B	↓B12	↓B11	↓B10	3		
5	fra Svendborg	Indk	1	strækkn	sp ¹	sp ²	ra	ra ¹		+	aff	↑	↑	↑	↑	↑	↑	↑			B	↓B12	↓B11	↓B10	5		
6		Indk	2	strækkn	sp ¹	sp ²	ra	ra ¹		+	aff	↑	↑	↑	↑	↑	↑	↑			B	↓B12	↓B11	↓B10	6		
7	til Odense	Udk	1				ra	ra ¹		+	aff	↑	↑						Ja	Ja	E	↓E11	↓E10	7			
8		Udk	2				ra	ra ¹		+	aff	↑	↑						Ja	Ja	F	↓F11	↓F10	8			
9	til Svendborg	Udk	1				ra	ra ¹		+	aff			↑	↑						G	↓G11	↓G10	9			
10		Udk	2				ra	ra ¹		+	aff			↑	↑						H	↓H11	↓H10	10			

Fig. 5. Train route table for Stenstrup station

A graphical editor for creating relay circuit diagrams and track layout diagrams has been implemented, see [9], and recently it has been extended with the ability to specify train route tables, see [10].

5 Data Validation

The data validator tool can be used to check that the station documents follow structural rules of the domain, e.g. that

1. the track layout diagram represents a legal railway network of track elements,
2. the circuit diagrams represent a legal network of circuits,
3. the train route table
 - (a) refers only to track elements in the track layout diagram,
 - (b) describes only routes that are connected paths in the railway network in the track layout diagram,
 - (c) marks overlapping routes as being conflicting,
 - (d) ...

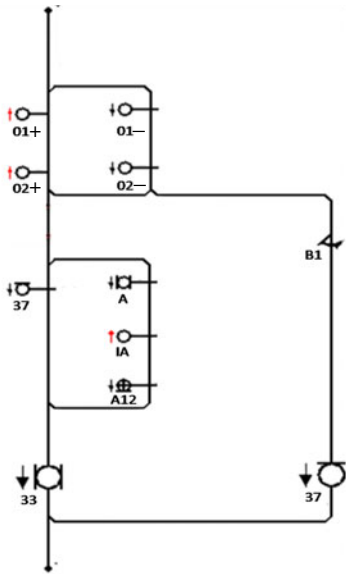
The data validator has been integrated with the editor.

6 Simulation

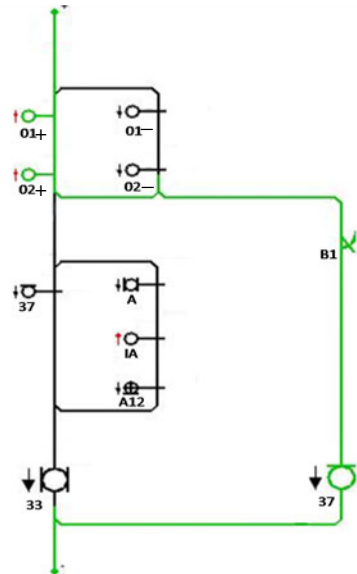
When relay circuit diagrams for an interlocking system have been created by the editor and checked by the data validator, the simulator can be used to visualise on these diagrams how the states of the relay circuits change over time. In this visualisation one can see the state of wires (current carrying or not), the state of relays (drawn or dropped), the state of contacts (connected or disconnected), and the state of buttons (pushed or released).

The user can give input to the system by playing:

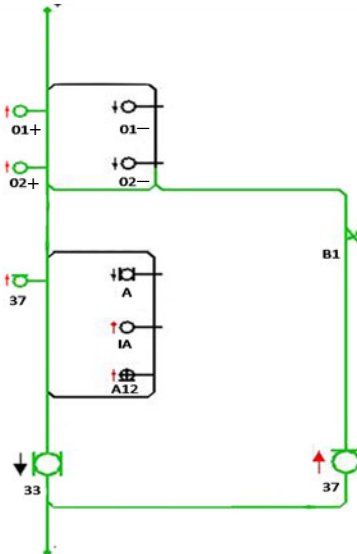
1. an *operator* that pushes a button of the relay circuits (to lock a train route) or switches a point, and
2. a *train* that enters or leaves a track section shown in the track layout diagram.



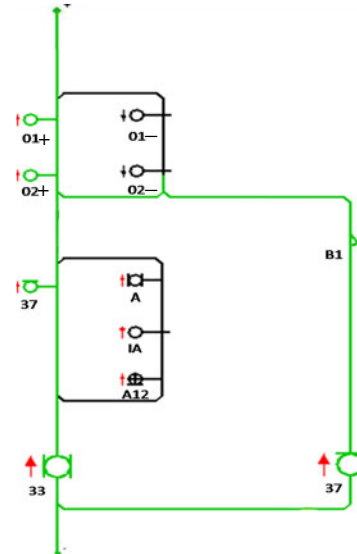
State 0 : initial state



State 1



State 2



State 3

Fig. 6. A state sequence for a circuit

After having given an input the user can step through the sequence of states that the relay circuits will go through after such an input.

In Figure 6 is given an example of a simulation showing how the state of a circuit changes when a button is pushed. Wires that are current carrying are shown by a green colour (seen as a grey colour in black&white print). State 0 is the initial state. In the initial state, no wires are current carrying as there is no path from plus to minus. When the button is pushed, current flows from plus to minus through relay 37, see state 1. As a consequence of this, relay 37 is drawn and its associated upper contact becomes connected, opening a second path of current from plus to minus through relay 33, see state 2. As current flows through relay 33, this will be drawn, see state 3. In state 3 no more internal events can happen.

A detailed description of the simulator tool and its development is given in [9].

7 Verification

This section describes how our framework provides verification support for a relay interlocking system. More details can be found in [14,6].

We have chosen *model checking* as the verification approach as this allows for full automation.

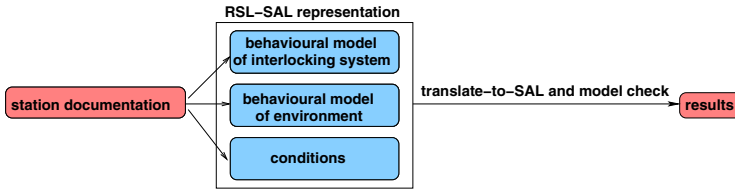


Fig. 7. Verification in two major steps

When the station documentation for the interlocking system has been created in the domain-specific language and checked by the data validator, verification can be performed in two major steps as illustrated in Figure 7. First, generators are applied to automatically generate input to a front-end of the SAL model checker [3]:

- a behavioural state transition system model M_c of the relay interlocking system,
- a behavioural state transition system model M_e of the environment (track isolations, points, and operator buttons), and
- conditions ϕ in the form of formal assertions about the desired behaviour of system.

This input is expressed in RSL-SAL⁴. Then, in the second major step, the RSL-SAL representation is translated (by a front-end translator provided by the RAISE tool set `rsltc` [2][12]) into a representation in the SAL input language [8], upon which the SAL model checker is applied to check that the concurrent composition of the models M_c and M_e always satisfies the conditions ϕ .

In sections 7.1- 7.2 we briefly describe the behavioural models and the conditions that are generated from a domain specific description.

The described verification approach can be used to verify that a given interlocking system ensures desired properties such as trains do not collide and do not derail when some assumptions (like trains drive at a safe speed, trains do not pass signals showing a stop aspect, and physical components are not faulty) are met. More examples of such assumptions are given in section 7.1, while all assumptions are stated in [6]. The verification approach is based on the further assumptions that

- the DSL documentation made by the user faithfully describes the given station and its interlocking system,
- the generated conditions are correct formalisations of the desired properties
- the generated models correctly models the behaviour of the described interlocking system and its environment, and
- the other applied tools are correct.

The verification approach has successfully been applied to verify that Stenstrup station in Denmark is safe. Details of this case study are given in [14][6].

7.1 Behavioural Models

The behavioural models of an interlocking system and its environment consist of a common state space and rules for how the relays and buttons of the circuits can change states. The state space and rules are derived from the track layout diagram and the relay circuit diagrams.

Common state space. The common state space consists of

- a Boolean variable b for each button b and
- a Boolean variable r for each relay r .

When the value of a variable for a button/relay is **true**, it models that it is pushed/drawn, and when the value is **false** it models that it is released/dropped.

Transition rules for the interlocking system. The transition rules for the interlocking system describe the dynamic behaviour of the internal relays. For each internal relay r in the relay circuit diagrams two rules are generated, one for drawing it and one for dropping it:

⁴ RSL-SAL [16][17] is an extension of the RAISE Specification Language [18] with constructs for defining state transition systems and constructs for specifying desired properties of these in the form of formal assertions in the temporal logic LTL.

$$\begin{aligned} [\text{draw}_r] &\sim r \wedge \text{conducting}_r \rightarrow r' = \mathbf{true}, \\ [\text{drop}_r] &r \wedge \sim \text{conducting}_r \rightarrow r' = \mathbf{false} \end{aligned}$$

The first rule expresses that r can be drawn when r is dropped and conducting current, while the second rule expresses that r can be dropped when r is drawn and not conducting current. The condition conducting_r for current to flow through a relay r is a logical formula determined as follows. Current will flow through the relay if there is a path from the positive pole to the negative pole that flows through the relay, and all contacts within this path are connected and all buttons are pushed. Now for a given relay there are several potential paths, p_1, \dots, p_n , for current to flow through it. For each potential path p_i we express the condition cp_i for that path to be conductive. Then the condition for the relay to be conducting is the disjunction of these conditions:

$$\text{conducting}_r = cp_1 \vee \dots \vee cp_n$$

The condition for a potential path to be conductive is a conjunction of conditions for its contacts to be connected and its buttons to be pushed. The condition for a button b to be pushed is b . The condition for an upper contact and a lower contact belonging to relay r to be connected is r and $\neg r$, respectively.

As an example, from the diagram in Figure 6, the following condition for current to flow through relay 37 is derived:

$$\text{conducting}_r37 = (01+ \wedge r02+ \wedge B1) \vee (01- \wedge r02- \wedge B1)$$

Transition rules for the environment. The transition rules for the environment specify assumptions on how external relays (track relays and point relays) and buttons can change state.

A state change of an external relay or a button is considered as an input to the interlocking system. Such an input may lead to a chain of internal events: internal relays that are drawn and dropped. In practise such chains are very short (at most of length 6) and take almost no time. Therefore we have taken the assumption that new inputs can first happen when no more internal events can happen.

The system is said to be in an *idle* state, when it is ready for input, i.e. no internal event is possible. In order to easily keep track of when the system is idle, a Boolean variable *idle* is added to the state space and a rule for the system to become idle is introduced:

$$\sim \text{idle} \wedge \sim c \rightarrow \text{idle}' = \mathbf{true}$$

where c is the disjunction of all guards in the rules for internal relays.

Transition rules for external events should then always take the form:

$$\text{idle} \wedge \dots \rightarrow \dots, \text{idle}' = \mathbf{false}$$

In [14,6] patterns for all rules to be generated are stated. Below we informally explain the assumptions behind the transition rules for track relays and point relays.

Transition rules for track relays. The tracks are divided into track sections, each of which is monitored by an associated track relay. We assume that track sections are only occupied by trains and hence completely controlled by the movement of trains: when a train enters or leaves a track section, the associated track relay is dropped and drawn, respectively. Therefore the rules for track relays should reflect possible train movements and will depend on the track layout for the station and in particular on the placement of signals. The rules reflect the following assumptions about train behaviour:

- Trains enter only a station from entry sections and leave only a station from an exit sections.
- Trains follow the tracks.
- Trains do not pass signals showing STOP.
- Trains do not change direction while using a route.
- Trains do not split.

Transition rules for points. Points can be in one of three positions: plus (straight), minus (branching), and intermediate (between plus and minus). Each point p has its position monitored by two relays, $p+$ and $p-$ that are drawn only when p is in its plus position and p is in its minus position, respectively. Hence, p is in its intermediate position when both $p+$ and $p-$ are dropped.

For each point in the track layout diagram there are four transition rules for switching the point from plus to intermediate, from intermediate to minus, from minus to intermediate, and from intermediate to plus, respectively. Each of these rules reflects the following assumptions about point behaviour:

- A point is only allowed to change state, when the track section of the point is unoccupied, and all routes that include the point are unlocked/released. (The latter condition can be determined as for each route there is a relay that is drawn when the route is unlocked/released).

7.2 Conditions

Three kinds of conditions (LTL formulae expressed in RSL-SAL) are derived from a domain-specific description:

- *Confidence conditions*, i.e. conditions expressing that the circuits are well-designed in the sense that
 - there are no internal cycles where the same sequence of internal relay events is repeated over and over again as the reaction to an input to the system, and
 - there are no critical races (so that the system always reacts in the same way to the same input).

These conditions are derived from the circuit diagrams.

- *High level safety conditions*, i.e. conditions that directly express what it means for the domain under control to be safe, i.e. that express that there are no train collisions and no derailling of trains. These conditions are derived from the track layout diagram and characterised by being independent of the chosen interlocking protocol (that is why they are called *high level safety conditions*).
- *Low level safety conditions*, i.e. conditions expressing that the interlocking rules specified by the train route table are satisfied. These conditions depend on the chosen approach to interlocking and are derived from the train route table. For instance, for each route in the train route table there is a condition expressing that the entry signal of that route must be switched to a STOP aspect when the first section of that route becomes occupied. The signal and the section are specified in the **Stop fald** column of the train route table.

In [6] patterns for all these conditions are given. Here we just give a few examples of some of the conditions derived for Stenstrup station.

Example of a confidence condition. In our context, the absence of internal cycles is equivalent to requiring that the system will always eventually become idle:

$$G(F(\text{idle}))$$

Example of a high level safety condition. The following condition is derived from the track layout in Figure 4. It expresses that when a train is occupying point 01, the point must not be in a switching state:

$$[\text{no_derailing_01}] G(\sim t01 \Rightarrow (01+ \vee 01-))$$

Here $t01$ is modelling the track relay associated with point 01, and $01+$ and $01-$ are modelling the two point relays associated with the point.

Example of a low level safety condition. The following condition is derived from the **Stop fald** information for route 2 in the train route table in Figure 5. It expresses the rule that when section $A12$ becomes occupied, the green lamp in signal A must be turned off and the red lamp turned on:

$$G(\text{idle} \wedge \sim tA12 \Rightarrow \sim A\text{green} \wedge A\text{red})$$

Here $tA12$ is modelling the track relay associated with track section $A12$, and $A\text{green}$ and $A\text{red}$ are modelling relays that are drawn when the green lamp is on in signal A and when the red lamp is on in signal A , respectively. In the condition it is necessary to add *idle* on the left-hand side of the implication in order to give the system time to react on the occupation of $A12$, cf. [6].

8 Development of a Domain-Specific Language and Tools

We are using the RAISE Specification Language, RSL, [18] to specify the domain-specific language and most of the tools. Examples of this can be found in [6]. As implementation language of the tools we are using Java.

The specifications are typically developed starting with a property-oriented specification and ending with an executable specification. Some of the advantages of this are:

- It is easier first to make an abstract specification in which e.g. only the properties of functions are given, and then later make an executable specification in which algorithms for the functions are given.
- It is easier to define data types and algorithms in an RSL executable specification and then translate these into Java, than coding directly in Java.

9 Conclusions

In this paper we have given an overview of a tool set that is intended to help railway engineers to analyse and verify relay interlocking systems. To describe a system to be analysed or verified, the railway engineer just has to use an editor for a domain-specific language to create documents that railway engineers are already used to creating: track layout diagrams, train route tables, and relay circuits. The tool set also includes a data validator, a simulator, and model and condition generators that all take such documents as input. The data validator can be used to check that the created documents follow the structural rules of the domain, increasing the confidence that the documents describe a station and its associated interlocking system. The simulator can be used to give confidence that the electrical circuits behave as expected. The model and condition generators can be used to generate input to a model checker that can then be used to automatically verify that the described relay interlocking system satisfies a number of conditions (like trains do not collide or derail) when some assumptions (like trains not passing red signals and operating at safe speeds) are met. To use such automated tools is a great improvement compared to manual inspections of diagrams: it is faster and easier to do, and it reduces the risk that errors or omissions are made.

Prototypes of most of the tools have been implemented, while a few of them are currently under development.

The framework has successfully been applied to verify that Stenstrup station in Denmark is safe. In future work it should be tested whether the framework can be applied to larger stations without problems such as state space explosion during model checking.

In future work we plan also to experiment with other editors, simulators, visualisations, and model checking approaches to see what is most valuable and most efficient. For instance, we plan to make a generator that maps descriptions in the domain specific language into behavioural models in the form of SyncCharts [4] and then use the KIELER (Kiel Integrated Environment for Layout Eclipse Rich Client) framework [111] to graphically visualise simulation runs of the generated SyncCharts.

As an example of another experiment, we are currently exploring how the Maude [7] term rewriting system and model checker can be used for the verification of safety. The behavioural models are expressed as Maude rewrite theories

and the safety conditions are assertions in the temporal logic LTL expressed in the Maude Language. Details of some initial investigations are described in [10], but the investigations are not yet finished.

In the future it could also be interesting to transfer the conceptual ideas of this paper to other application areas such as air traffic control systems.

Acknowledgements. I would like to thank Kirsten Mark Hansen, Banedanmark, for providing the initial idea for this project and for many valuable discussions and suggestions. My thanks also go to my former students Louise Elmose Eriksen, Boe Pedersen, Marie Le Bliguet and Andreas A. Kjær, who have all contributed to this project in their bachelor and master theses, and who have helped produce some of the figures in this paper.

References

1. KIELER Eclipse project, home page, <http://www.informatik.uni-kiel.de/rtsys/kieler/>
2. RAISE Tools, home page, <http://www.iist.unu.edu/newrh/III/3/1/page.html>
3. Symbolic Analysis Laboratory, SAL (2001), home page, <http://sal.csl.sri.com>
4. André, C.: Synccharts: A visual representation of reactive behaviors. Technical report, I3S Laboratory, Sophia-Antipolis, France (April 1996)
5. Bjørner, D.: New Results and Current Trends in Formal Techniques for the Development of Software for Transportation Systems. In: Proceedings of the Symposium on Formal Methods for Railway Operation and Control Systems (FORMS 2003), Budapest/Hungary, May 15-16, L'Harmattan Hongrie (2003)
6. Le Bliguet, M., Kjær, A.A.: Modelling Interlocking Systems for Railway Stations. Technical Report IMM-M.Sc.-2008-68, Informatics and Mathematical Modelling, Technical University of Denmark, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, Master thesis supervised by Anne Haxthausen (2008), <http://orbit.dtu.dk> (search under department records)
7. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS. Springer, Heidelberg (2007)
8. de Moura, L., Owre, S., Shankar, N.: The SAL Language Manual. Technical Report SRI-CSL-01-02, SRI International (2003), <http://sal.csl.sri.com>
9. Eriksen, L.E., Pedersen, B.: Simulation of Relay Interlocking Systems. Technical Report IMM-B.Sc.-2007-04, Informatics and Mathematical Modelling, Technical University of Denmark, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, Bachelor thesis supervised by Anne Haxthausen and Hubert Baumeister (2007), <http://www2.imm.dtu.dk/pubdb/p.php?5306>
10. Eriksen, L.E., Pedersen, B.: Verification of Safety Properties for Relay Interlocking Systems. Technical Report IMM-M.Sc.-2010-57, Informatics and Mathematical Modelling, Technical University of Denmark, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, Master thesis supervised by Anne Haxthausen (2010), <http://orbit.dtu.dk> (search under department records)
11. Fuhrmann, H., von Hanxleden, R.: Taming graphical modeling. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 196–210. Springer, Heidelberg (2010)

12. George, C.: RAISE Tools User Guide, <http://www.iist.unu.edu/newrh/III/3/1/page.html>
13. Haxthausen, A.E.: Developing a Domain Model for Relay Circuits. *International Journal of Software and Informatics* 3(2-3), 241–272 (2009)
14. Haxthausen, A.E., Le Bliguet, M., Kjær, A.A.: Modelling and Verification of Relay Interlocking Systems (Invited paper.). In: Choppy, C., Sokolsky, O. (eds.) *Monterey Workshop 2008*. LNCS, vol. 6028, pp. 141–153. Springer, Heidelberg (2010)
15. Haxthausen, A.E., Peleska, J., Kinder, S.: A Formal Approach for the Construction and Verification of Railway Control Systems *Formal Aspects of Computing*. *Formal Aspects of Computing* 23(2), 191–219 (2011); Special issue in Honour of Dines Bjørner and Zhou Chaochen on Occasion of their 70th Birthdays; The article is also available electronically on SpringerLink, <http://www.springerlink.com/openurl.asp?genre=article&id=doi:10.1007/s00165-009-0143-6>
16. Perna, J.I., George, C.: Model checking RAISE specifications. Technical Report 331, UNU-IIST, P.O.Box 3058, Macau (November 2006)
17. Perna, J.I., George, C.: Model Checking RAISE Applicative Specifications. In: *Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods*, pp. 257–268. IEEE Computer Society Press, Los Alamitos (2007)
18. The RAISE Language Group: *The RAISE Specification Language*. The BCS Practitioners Series. Prentice Hall Int., Englewood Cliffs (1992)
19. The RAISE Method Group. *The RAISE Development Method*. The BCS Practitioners Series. Prentice Hall Int., Englewood Cliffs (1995)
20. Schnieder, E., Tarnai, G. (eds.): *Proceedings of Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2007)*, Braunschweig, Germany. GZVB e.V (2007) ISBN 13:978-3-937655-09-3

Trust Of, In, and among Adaptive Systems

Douglas S. Lange

Space and Naval Warfare Systems Center Pacific
San Diego, CA 92152
doug.lange@navy.mil

Abstract. Adaptation brings with it challenges related to trust. Mission critical and safety related systems usually require very predictable and repeatable behavior in order to be deployed. However, adaptation can satisfy many of the newer requirements of these systems. Service oriented architectures that adapt can enhance robustness, yet they bring the issue of choosing which of several services to trust for a particular need. Some adaptive systems require human interaction or must observe their human users in order to effect the adaptation. Users must develop trust for these systems so that they feel their time and energy are not wasted and that the system will in fact yield increased efficiency and effectiveness. This paper discusses these challenges and suggests some solutions for ensuring trust in adaptive systems.

Keywords: adaptive systems, machine learning, trust, mission critical, service oriented.

1 Introduction

Earth's fossil record is littered with evidence of species that failed to adapt successfully to environmental changes. The mechanism of evolution is the random mutation of genes during reproduction. Organisms then go through an evaluation process: *life*. Those that are successful enough to reproduce enhance the chances that their genes will be propagated since mutation of a gene is less likely than perfect reproduction of that gene. As the environment changes, different gene configurations have different advantages than they did previously and species adapt.

Nature doesn't demand that adaptations be successful. There is no preference for success or failure of a particular individual or species. The earth goes on and the composition of life forms change. While evolution has produced extremophiles that can be seen surviving in areas where other species cannot, it has also produced species that became extinct in environments that many other species find easy to inhabit.

Nature has no preference, but we do. We do not usually have the time and money necessary to try out millions or billions of system configurations over hundreds or

thousands of generations and see which combinations suit our needs in our environment. Nor do we have the luxury of allowing many of our mission-critical systems to fail as their environment changes so that we can make use of the information of those that remain to make the product line stronger. Our systems must give us a promise of success with a typically high degree of certainty.

Yet adaptation may still have a role to play in achieving that high degree of certainty of success. Software engineers simply cannot count on adaptation alone, and random mutation without some regulating mechanism cannot be our sole tool.

In this paper, we will explore some ideas for using and regulating adaptation in such a way as to allow trust in those systems that use them. The first employs redundancy. By using multiple redundant components of differing architectures, some of which adapt, while some evaluate trends in performance, and some enforce invariants, we may be able to regulate how adaptation is allowed to express itself.

Second, we will look at how the environment, which includes other systems that may be impacted by an adapting component, evaluates whether or not to make use of the component. In a complex system, the improper adaptation of one component can cause a lack of success for another component, and possibly the entire system. An example of this is found in *service oriented architectures*. As services evolve, we must provide information to other components to allow them to decide how much to trust the services.

Finally, we will discuss how adaptive systems can be prepared to enter the environment in a manner that improves their chances of success, and allows the users of such systems to gain confidence in their use. It can be shown that because human users are not neutral to success or failure in the way that nature is, an adaptive system can fail simply because of a lack of trust, or due to an undervaluing of the systems long-term performance.

2 Adaptation in Mission Critical Systems

Are we sure that we want adaptation in mission critical systems? Directions in systems within the United States Department of Defense indicate that we are in fact beginning to deploy adaptive capabilities in critical settings. A few examples can illustrate the early moves.

The Defense Advanced Research Projects Agency (DARPA) sponsored the Personalized Assistant that Learns (PAL) program that brought together researchers in artificial intelligence, machine perception, machine learning, natural language processing, knowledge representation, multi-modal dialog, cyber-awareness, human-computer interaction, and flexible planning. The single research focus was to create an integrated system that can “learn in the wild”—that is, adapt to changes in its environment and its user’s goals and tasks without programming assistance or technical intervention. The goals of PAL are illustrated in the figure below.

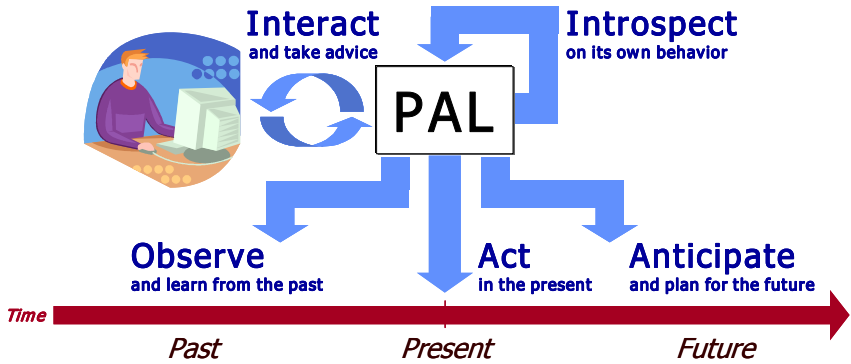


Fig. 1. PAL Goals [From 17]

This goal also implies that the system will adapt without formal verification, validation, or certification. PAL is expected to learn new tasks by observing a user or being taught by that user. While the targeted domain of this capability is not in weapons systems where direct safety and efficacy concerns might exist, it is intended to operate in decision-making environments. Mistakes made by decision-makers can lead to catastrophic results just as faults in direct safety-critical systems can.

Several levels of adaptation were studied and developed within the PAL program. The figure below illustrates the varying levels of human involvement with the system's adaptation.

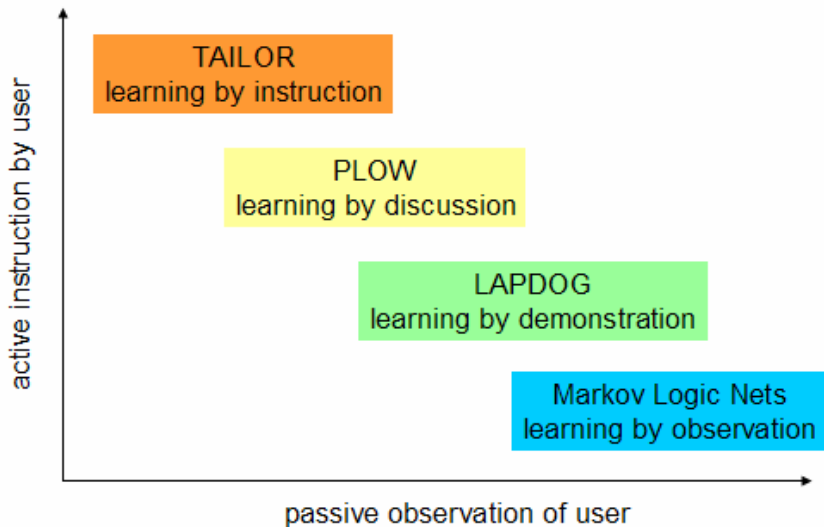


Fig. 2. Levels of Training an Agent [From 17]

While we may be more comfortable with adaptation that has high active involvement from the user, it is also true that we rarely allow the deployment of software developed by a human without some form of evaluation. Methods for evaluating the new possible behaviors of systems that learn in addition to mechanisms for protecting against learned aberrant behaviors are necessary.

3 Unlike Redundancy

It can be easily shown that redundancy can be employed to improve system reliability. The Space Shuttle avionics system contains five identical computers to perform flight-critical functions [1]. Redundant systems are often utilized with a voting scheme so that components that are failing can be identified when they come up with answers that don't agree with those of the others. The National Aeronautics and Space Administration (NASA) uses five computers so that even after two failures there will be still be enough computers to vote and not end up in deadlock. The main assumption of this approach is that failure of one of the components is an independent event and is not due to a design flaw [2].

Sometimes it is better to use *unlike redundancy*, where differently designed components perform the same function [3]. This is done when one cannot be certain that failure of components will be independent events and not due to a design flaw. This will reduce the likelihood that the components will not all fail simultaneously due to a common fault.

If adaptive systems are to be used in mission critical environments, there may be a benefit to using unlike redundancy. In addition, adaptive systems offer opportunities to provide such redundancy naturally.

3.1 An Autonomous Vehicle Problem

Suppose that we want an autonomous vehicle to operate with minimal human intervention while performing a mission critical and safety critical task. The United States Department of Defense (DOD) has stated in solicitations for proposals that armed unmanned systems will need to be able to operate autonomously, and collaboratively engage hostile targets "within specified rules of engagement". While the solicitation includes that final decisions on engagement will be "left to the human operator", it goes on to say that "Fully autonomous engagement without human intervention should also be considered, under user-defined conditions..." [4]

How will such robotic systems be designed? There is research in many areas that may provide the technology necessary for autonomous systems to make such critical decisions as to whether or not to fire a weapon. Programming the tactics required for all situations is a difficult and imprecise process. Researchers at Stanford have shown that unmanned air vehicles can be *taught* behaviors more effectively than they can be programmed in some situations [5]. Similarly, teams of autonomous agents have been taught tactics using neuroevolution in a video game environment. In the NERO game, players train teams of virtual robots. This is accomplished by training and running successive generations the robot teams through training games and selecting those robots

that exhibit elements of the desired behavior and using them to create the next generation. These teams are used for combat against teams developed by other players [6].

But will we trust robots that have learned tactics and decision making in a simulated environment and through previous operations. While they likely will have an advantage over rigidly programmed vehicles when the environment changes, we will not necessarily know how to predict vehicle responses to new situations. We expect emergent behaviors, yet still must make sure that they are good.

One method proposed for dealing with the robots allowed to make important, or indeed lethal decisions, is to program in a conscience [7]. In this approach an *ethical adaptor* is employed. The robot predicts the outcome of each engagement. If the decision is to engage, then after the engagement, the robot receives feedback on the results (e.g., the existence of civilian casualties). If the result was worse than expected, the robots confidence in its evaluation is degraded and it loses the authority to use those weapons where the results are less certain, or loses some degree of autonomy. The vehicle may now only use those weapons that have less possibility for unexpected casualties or may have to receive additional checks from a human operator to use riskier weapons.

Finally, there are invariants that can be programmed into an autonomous vehicle. Whenever a known situation can be described programmatically, this approach can be employed. Autonomous vehicles can be programmed not to fire weapons within some radius of known schools, hospitals, and etc. This approach is of course brittle in the face of a changing environment, but can provide safeguards in situations where a robot has learned incorrectly.

3.2 A Mixed Approach May Improve Trust

Imagine using all three of the above methods to provide unlike redundancy and allowing the three subsystems to vote on the final decision. The machine learning component will evolve to learn when to engage and when not to within the changing environment. It will employ tactics taught by its commanders and through simulations. But it may learn some wrong things.

The ethical adapter will serve as a check. If the decisions that the three modules have voted to proceed with have gone badly, it will more-and-more often vote against dangerous action. This is independent of the learning system, which should also learn from mistakes and make adjustments. By having two different approaches, mistakes should be fewer and we could have more trust in the system.

Likewise, the invariant component has the third vote and whenever a constraint is recognized, it will vote against action. To err on the side of caution, one might give the invariant component and the ethical adapter components two votes each, allowing each to abstain as well when the environment does not offer them any information. Unanimity may also be required for the most serious decisions, such as weapons release.

We cannot allow systems that make mission critical or safety decisions simply to adapt and improve without some form of check. Unlike nature, we must judge mistaken adaptation as being undesirable and cannot easily accept failures.

4 Graduated Certification

Large systems of systems (SOS) must adapt. With very large numbers of components, they are typically developed in an adaptive manner, rarely remaining static. In particular large SOS are being developed using service oriented architectures (SOA) specifically so that new components can make use of previous services without the original services necessarily having been designed for that purpose [8]. Similarly, large networks of services are being developed for large sensing networks like the *Ocean Observatories Initiative* [9] and for military command and control networks [10].

Beyond human-driven adaptation of the network through the addition, modification, and deletion of services, automated service composition is possible to some degree now and presumably more so in the future [11]. SOA is being employed precisely because many other architectures have been considered too static to handle changing environments. SOA are considered to be more adaptable.

4.1 Problem of Trust in an Adaptive SOA

There has been a great deal of research into how to ensure that a service knows who is asking it to perform. SOA can be even be implemented across security domains provided sufficient authentication and system-level security are present [12]. Policy frameworks are being defined and developed that will allow access policies to be developed and adapted as the network changes [13]. Services can determine whether or not they can trust a requestor.

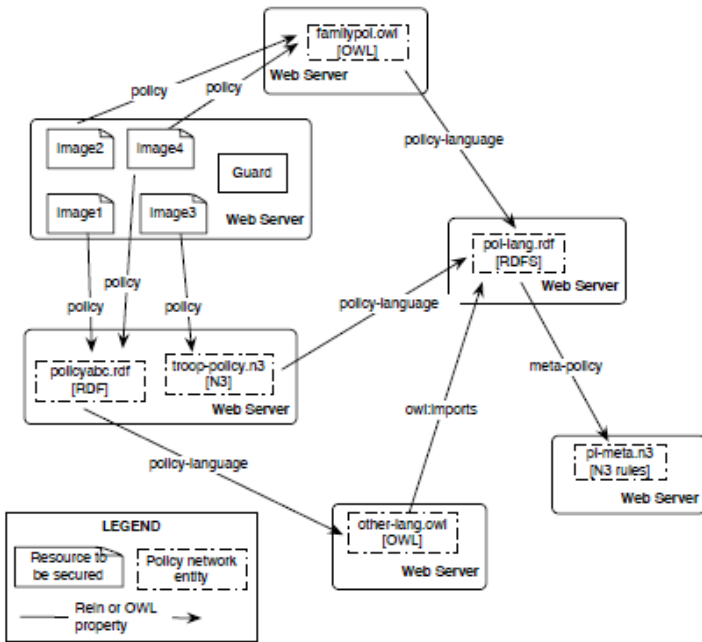


Fig. 3. A Rein Policy Network [From 13]

But the reverse problem is more complicated. As the network adapts, new or upgraded services may appear that provide essentially the same function as previously available services. Some of these services may be automatically generated. Services may have gone through different levels of quality assurance. How can a person or application know which services to trust, not only with regard to more straightforward aspects of network security, but in terms of quality of service: how accurate will the answer be, how long will it take to get a response, and etc?

The simple answer would be to have a uniform policy for evaluating new services and preventing the introduction of services that have not yet met the criteria for deployment. This would preclude the automatic generation of new services and is likely to cause stagnation of very large SOA networks.

4.2 Employing Policy Languages

The Rein policy framework [13] is an example of a capability for providing service access policy. In the figure below, the framework is used to protect images on a web server.

We propose using it the other way around. We would like to define a graduated certification scheme. Possible levels of certification are shown below.

Table 1. Certification Grades

Level	Description
0	Automatically Generated Service or Human Generated services through choreography system
1	Automated test module executed and passed
2	Component Testing performed at DOD Service lab
3	Workflow testing performed at DOD Service lab
4	Performance experience mature

A policy network can be developed based on Rein to allow applications to decide which services they trust enough to use. Applications that require more trust will opt to avoid newer less mature services, while those that are not as sensitive but require the information provided by newer components will make use of the new information. Similarly, security enclaves may be automatically created around services based on their level of trust from the standpoint of information assurance. More trusted services will be allowed access and placement within areas of the network that may be more sensitive.

Where there is more than one choice, applications might trade-off performance with certification. This type of certification policy framework will allow the network to evolve, while ensuring that capabilities on the network are using services that are appropriately mature.

5 Preparing Adaptive Systems for the Real World

Humans are adaptive systems, yet we do not get deployed without training. The Defense Advanced Research Projects Agency (DARPA) implemented a program to

build the first instance of a complete cognitive agent based on integrating multiple machine learning adaptation methods. The program, called Personalized Assistant that Learns (PAL), has yielded new cognitive technology of significant value not only to the military, but also to the business and academic sectors [14].

A significant question came up early in the PAL program. “When is a PAL ready to be fielded?” In standard military programs, deployment occurs after the system has passed a formal test and evaluation phase. If the system has met the requirements and is found to be operational useful when tried by the users, it is ready to go. The PAL program suggested a different approach was needed. Some of the learning technologies were designed to allow a PAL to learn many of its capabilities after being deployed. The users’ environment is guaranteed to be different than it was when the system was first specified, so PAL was created to learn *in-the-wild*.

Trading off cost (including risk) with benefit is the basis of deployment decisions. As stated above, software that adapts through machine learning may start out with little benefit to go with whatever risk is present in installing it. Learning systems are also likely to appear more risky by their very nature because their future behavior is not entirely predictable. If there isn’t sufficient measurable benefit to weigh against the risk, the decision is likely to be negative.

The anticipated advantages of learning in the wild are that capabilities that are difficult to specify will be achieved, and that new capabilities that are needed due to the rapidly changing environment can be provided without returning to the laboratory. PAL will learn entirely new tasks, without new software being developed by engineers. Therefore, PAL would not necessarily pass a formal test against a set list of requirements, but PAL has capabilities to learn overtly from the user and by observing the users actions [15].

5.1 Three Alternative Results

If we are to imagine deploying a PAL, there are three basic results that can be envisioned, and they are likely to be determined by the behavior of the human users as well as the learning ability of the software. The initial knowledge state of the PAL is also likely to be important.

The three possible results are:

1. *Expected (Success)*. The PAL is capable enough at the start and quick enough at learning to make it useful to its human user. The user’s initial investment (at the cost of efficiency and effectiveness) is not too expensive and pays off so that effectiveness improves over time. Eventually the learning plateaus but not before the investment pays off.
2. *Abandonment*. The user’s performance suffers at the beginning, and it does not improve quickly enough so that the user will see it as useful. The user abandons the PAL and effectiveness goes back to pre-PAL levels.
3. *Disaster*. The user’s performance suffers at the beginning, and it does not improve. PAL does not learn, or learning does not benefit the user, but the user continues to use PAL resulting in a permanent drop in effectiveness.

The figure below illustrates notional performance curves for the human-computer team for each of the scenarios above. From the introduction of a PAL at ‘A’, we see a

degradation of performance initially. We might see a bottoming or at least slowing of this cost by the time labeled 'B'. If the user sees improvement early on, we may be on the success curve, and performance could continue to improve and overtake previous levels of performance at point 'E'. Learning tends to plateau at some point, which for the graph below we label as 'F'.

Abandonment is the likeliest outcome if performance does not improve quickly enough. We can imagine the user becoming frustrated and needing to return to old ways of performing the job in order to maintain required levels of effectiveness and efficiency. In the notional graph below, abandonment occurs at point 'C', and the user returns to levels of performance that preceded deployment of PAL by point 'D'.

The final scenario is depicted by the green dotted line. Performance does not bounce back up, and a determined user continues to use the assistant, with continued decreases in performance due to efforts to train the assistant that are not working.

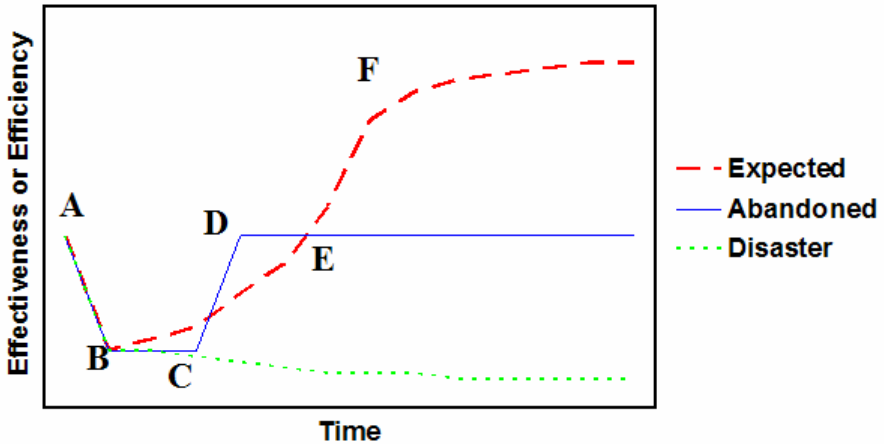


Fig. 4. Notional Performance Curves

If these scenarios are believable, then assessing the benefit of a PAL or similarly adaptive system becomes a problem of determining the likelihood of success. In order to do that, we might be able to make use of models of behavior, learning, and the resulting performance changes. We also may want to develop processes that improve the likelihood of success, and we will need models of how to measure progress during these processes.

If we accept that we cannot evaluate a PAL for operational capabilities prior to its use, we must still be able to determine when a PAL is ready to be sent out into the field. Two thresholds must be surpassed. First, the PAL must know enough to learn from the user and from observation. Second, the PAL must be useful enough in order for a human to be willing to have it around. These were shown in experiments using simulations of the learning capabilities and randomized task lists from a test domain [16]. There were many other factors, but the statistical analysis showed that these

were the keys. So this becomes the survival test for our adaptive and adapted system. Will the user tolerate and teach the system?

5.2 A Boot Camp

Anthropomorphic descriptions are difficult to avoid when thinking about the PAL program. The inspiration behind the program, quite literally, is the benefit gained by those fortunate enough to have a human assistant. There are however, differences that immediately come to mind when one delves deeper into the possible capabilities of a PAL when compared with a human assistant. An example is that two or more PAL may be able to directly transfer knowledge among one another, rather than first describing that knowledge in some language that can only provide an approximation of the needed information with the hope that the receiver correctly interprets the message.

However, there are many similarities between a PAL and a human assistant. For example, both must learn how their boss wants to conduct business, their performance is best measured by the change in their boss's performance, and if not found useful will get cut out of processes where the boss feels that it is more expedient to just do it him or herself. Because of the many similarities, it is useful to consider how one deals with the introduction of humans into the operational environment and decide if similar processes are useful for PAL.

Structured training is one technique used to prepare humans for their jobs. Most famous is military basic training, also known as boot camp. New recruits in the military already have been to school (e.g., high school or college), but need to learn the basics of the military domain. For a PAL, it may be difficult to program in all the knowledge it can gain from the target domain within a training setting.

Another technique used in the military for training humans is simulation. For instance, naval officers moving to a joint staff tour are taught crisis action planning (CAP) processes in training driven by simulation systems. To be successful, the officers need to have certain amounts of prior knowledge, and can have their knowledge evaluated before leaving. There is no thought that the real life situations will exactly mirror those they have encountered in training. Rather it is believed that the skills they gain will allow them to learn the rest of what they need to know while on-the-job.

This approach parallels the goals of the in-the-wild learning by PAL. The assistant must learn within its operational environment, but in order to do so, must have some basic knowledge about its domain and the basic processes that are likely to be encountered.

The approach proposed for a boot camp for cognitive systems consists of three phases. In the phase prior to the boot camp, knowledge is being added through programming. This can be either using traditional programming languages or as statements in a logic-based knowledge base, or by any number of means. The distinction is that the method is one of designed and engineered programming. Learning algorithms may be employed, but through a controlled method, and through training data sets. Knowledge is primarily programmed into the assistant.

Prior to entry into the boot camp there is an opportunity to measure the knowledge currently held in the assistant. In the PAL program, this was done through an adaptation of traditional standardized tests for human students. First, a task analysis was conducted, and then necessary skills and knowledge were determined. Finally, exam

questions were generated to test for the necessary skills and knowledge [18]. PAL is meant to interact with human users, so using test techniques that are applied to humans seems appropriate. This fits, in general, with capabilities that are intended to be of assistance to humans.

The consequences of inadequate programming are that the assistant cannot function well enough to learn in the simulation environment of the boot camp. This is the same issue we are hoping to avoid in operational use. Therefore, the measurements that support the decision process are equally valid in both environments. It is a simpler matter to measure performance changes in the boot camp environment than in the operational environment and if we can use it to approximate the operational environment, we will be well off.

The next phase, the boot camp, is characterized by the use of simulation systems. Players using the simulation are cooperating to create and execute plans that adequately respond to the game situations presented in scenario. As the game unfolds, reports are generated by the simulations that drive the decision support displays. These communications are monitored by PAL, as are collaborative chat and email messages among the players. The goal of the players is to recognize the situation, develop a plan, and then execute it by composing and executing a plan in the simulation. It is the assistant's job to help where possible to improve the collaboration and even to help manage the task of composing and executing plans. These tasks are learned through observation and in some cases explicit instruction by the users.

There are many measurement opportunities in this process. The efficiency and effectiveness of the players can be judged. Efficiency can be measured through the time it takes from a particular point in the scenario until a new plan is composed and completed. Effectiveness can be measured by the number of successfully developed operational plans, each with different reward values, and deadlines.

Following work in the simulation environment, the opportunity exists to conduct a graduation exam. The mechanism for this can be the same as the test used prior to initiating use in the simulation environment of the boot camp, and it could well be the same exam in order to compare the changes in the performance on the exam with learning that occurred in the boot camp. The results are useful to correlate with the results of operational use to help determine if the boot camp is successful.

The final step is operational use. Here the assistant is paired with a user. If the assistant has been trained sufficiently, the human will be willing to use the assistant and the assistant can continue to learn. If the assistant has been inadequately trained, it either will be abandoned or will degrade the human's effectiveness and efficiency.

Effectiveness and efficiency are more difficult to measure in an operational environment because we cannot set up repeatable controlled experiments. Here we have to rely on monitoring the behavior of the user in terms of use of the assistant.

6 Summary

Adaptive systems present a much more complicated problem than biological entities who participate in evolution. While the consequences of poor adaptation are dire for an individual, it means nothing to the earth. There is no judgment of a good or bad result to nature.

Our systems must adapt to their environments in order to stay useful. Development timeframes are much slower than the changes to the environment in many domains. Trust is an important factor for success of adaptive systems. Adaptive systems bring their own kinds of risks and in safety critical environments should perhaps be surrounded by other types of technologies in order to ensure that adaptation doesn't lead to catastrophe. As networks and their component services evolve, so too our ability to decide how much trust to give recently altered services needs to evolve. Finally, some adaptation requires a human to either explicitly teach or to at least allow a machine learning capability to participate and observe. The user must trust the system to increase effectiveness and/or efficiency, and must trust that the system will successfully learn in order to be willing to invest time and energy into aiding the adaptation. These are among the many challenges we face as we bring adaptation into our systems.

References

1. Sklaroff, J.R.: Redundancy Management Technique for Space Shuttle Computers. *IBM J. Res. Develop.* 20(1), 20–28 (1975)
2. Blanchard, B.S., Fabrycky, W.J.: *Systems Engineering and Analysis*, 3rd edn. Prentice-Hall, Inc., Upper Saddle River (1998)
3. National Aeronautics and Space Administration: Redundancy in Critical Mechanical Systems. Practice No. GSE-3003 (1995)
4. U.S. Army SBIR Solicitation 07.2, Topic A07-032: Multi-Agent Based Small Unit Effects Planning and Collaborative Engagement with Unmanned Systems (2007)
5. Coats, A., Abbeel, P., Ng, A.Y.: Apprenticeship learning for helicopter control. *Communications of the ACM* 52(7) (2009)
6. Stanley, K.O., Bryant, B.D., Miikkulainen, R.: Real-Time Neuroevolution in the NERO Video Game. *IEEE Transactions on Evolutionary Computation* 9(6) (2005)
7. Arkin, R.C., Ulam, P.: An Ethical Adaptor: Behavioral Modification Derived from Moral Emotions. Technical Report GIT-GVU-09-04. Georgia Institute of Technology (2009)
8. Broy, M., Kruger, I.H., Meisinger, M.: A Formal Model of Services. *ACM Transactions on Software Engineering and Methodology* 16(1) (2007)
9. Ocean Observatories Initiative, <http://www.oceanleadership.org/programs-and-partnerships/ocean-observing/ooi/>
10. Galdorisi, G., Clarkson, J., Grossman, J., Reilley, M.: Composeable FORCEnet Command and Control: The Key to Energizing the Global Information Grid to Enable Superior Decision Making. In: Ninth International Command and Control Research and Technology Symposium. CCRP, Copenhagen (2004)
11. Pistore, M., Traverso, P., Bertoli, P.: Automated Composition of Web Services by Planning in Asynchronous Domains. In: International Conference on Automation Planning and Scheduling, Monterey, California (2005)
12. Raney, C.J.: Integrating Multilevel Command and Control into a Service Oriented Architecture to Provide Cross Domain Capability. In: Command and Control Research and Technology Symposium. CCRP, San Diego (2006)
13. Kagal, L., Berners-Lee, T., Connolly, D., Weitzner, D.: Using Semantic Web Technologies for Policy Management on the Web. In: 21st National Conference on Artificial Intelligence. AAAI, Boston (2006)

14. Gunning, D.: Beyond Science Fiction: Building a Real Cognitive Assistant. DARPA/Tech, DARPA, Anaheim (2004)
15. Lange, D.S.: Boot Camp for Cognitive Systems. In: 21st National Conference on Artificial Intelligence. AAAI, Boston (2006)
16. Lange, D.S.: Boot Camp for Cognitive Systems: A Model for Preparing Systems with Machine Learning for Deployment. Ph.D. Dissertation. Naval Postgraduate School (2007)
17. Myers, K.: Building an Intelligent Personal Assistant. In: 21st National Conference on Artificial Intelligence, AAAI, Boston (2006) (an invited talk)
18. Cohen, P., Pool, M.: The CALO, Experiment Data Analysis. (2005), <http://calo.sri.com> (accessed January 15, 2007)

Software Certification: Is There a Case against Safety Cases?

Alan Wassying*, Tom Maibaum*, Mark Lawford*, and Hans Bherer*

McMaster Centre for Software Certification
Faculty of Engineering
McMaster University, Hamilton, Ontario, Canada L8S 4K1
{wassying,lawford,bhererh}@mcmaster.ca, tom@maibaum.org

Abstract. Safety cases have become popular, even mandated, in a number of jurisdictions that develop products that have to be safe. Prior to their use in software certification, safety cases were already in use in domains like aviation, military applications, and the nuclear industry. Argument based methodologies/approaches have recently become the cornerstone for structuring justification and evidence to support safety claims. We believe that the safety case methodology is useful for the software certification domain, but needs to be tailored, more clearly defined, and more appropriately structured in analogy with regulatory regimes in classical engineering disciplines. This paper presents a number of reasons as to why current approaches to safety cases do not satisfy essential attributes for an effective software certification process and proposes improvements based on lessons learned from other engineering disciplines. In particular, the safety case approach lacks the highly prescriptive and domain specific nature that can be seen in other engineering specialties, in terms of engineering and analysis methods to be applied in generating the relevant evidence. Safety case approaches and corresponding methods should aim to achieve the levels of precision and effectiveness of engineering methods underpinning regulatory regimes in other engineering disciplines.

1 Introduction

Software certification is in the news; see, for example, [1] for an academic account, and the following online discussions for “popular” reaction: an infusion pump that had to be removed from the U.S. market [2]; a software defect that prevented the emergency stop on the Gamma-Knife [3]; and two instances in which software failure resulted in horrific damage caused by radiation machines [4,5]. From automotive recalls to radiation device malfunctions, actual and potential deaths caused by faulty software have woken people up to the fact that software embedded in devices of all kinds has the capability of both helping us

* Supported by the Ontario Research Fund, and the National Science and Engineering Research Council of Canada.

and killing us, or, less dramatically, having serious consequences for individuals and the environment. It is quite obvious to many, if not most, people that software is an incredible enabling technology. It is so good, in fact, that there is now almost no new device/technology on the market that does not depend on software in some way, either for its function or for its design. We have also been remarkably successful in building huge numbers of software enabled devices, with a rather limited number of known serious problems. However, this is quite misleading, and has resulted in severe over-confidence, both on the part of manufacturers and the public at large. As software and devices become increasingly complex and safety features get further intertwined with functional features, the chance of creating serious disasters also dramatically increases. Every now and again, just to remind us, software faults in critical applications feature prominently in the world's news. In contrast, establishing software certification and regulation as the norm is not on most people's horizon, never mind establishing real improvements in the regulation of software based devices.

Motivated by the developments above, we have become interested in promoting the concept of licensing for software and systems containing software in order to provide the public and governments with greater confidence in the safety (and efficacy) of such products. We want to put software certification on a proper engineering footing, analogous to other engineering domains. A reasonable characterization of *Software Certification*, in the context we are referring to, is that it is a demonstrated assurance that the system has met relevant technical standards and specifications designed to ensure it will not endanger the public, that it can be depended upon to deliver its intended service safely and securely and that it is effective [6]. We also contend that the goal of certification is to systematically determine, based on the principles of science, engineering and measurement theory, whether an artifact satisfies accepted, well defined and measurable criteria [6].

Although there are some regulatory regimes applicable to software, experience has shown that most of these regimes have struggled with how software can and/or should be regulated. These existing regimes are often based on assuring the quality and safety of the software based on the fact that the producer of the software adheres to some defined process, often one that is realized in an international standard for development processes. Recently, we have detected growing awareness that this so-called *process based software certification* cannot deliver the assurance we need, and *product focused certification* is viewed as a significant advance.

In the past few years, we have seen *safety cases* assuming a more prominent role in regulatory regimes for software based systems. They are touted as *the* way to go for certifying the safety of systems, especially those enabled by software [7]. A good introduction to safety cases, their structure and to a methodology for their development can be found in [8]. Safety cases are touted as *the* way to go for certifying the safety of systems, especially those enabled by software [7]. Accordingly, recent years have seen tremendous effort expended on safety cases. For example, a start has been made to develop systematic processes and

formalisms that would help increase the level of confidence in the soundness of a safety case [9]. There is also interesting work on how to characterize the chain of evidence, and how to produce it, in making the argument in a safety case [10]. Work is also being done to extend safety claims to more general software properties [11], and a number of researchers have broadened the safety case into an *assurance case*, see [12] as well. To complete this brief review, approaches for using safety/assurance cases to certify adaptive systems [9], as well as generic software based systems [13], have also emerged in recent years.

We were impressed by what we read about safety cases and started looking for more information on them: theory as well as their application in practice. While doing this we started developing some misgivings about them. This paper re-examines the safety case approach, using lessons we should learn from other engineering disciplines as a criterion for evaluation - and we take as our example, Civil Engineering. This examination leads us to identify a number of potential problems in the use of safety cases, so that we can make suggestions on how to modify this approach so that it can be even more useful in certifying systems containing software.

2 Software Certification Approaches

In many cases, when we speak about *software certification*, we often mean that, in certifying the system behaviour of a software driven device, we need to pay specific attention to the behaviour of the software components and their interfaces to the physical device. In other cases, it may be that the software to be certified runs on generic hardware rather than embedded in a device.

Different software certification approaches exist and are usually characterized by their underlying philosophy as being either *process oriented* or *product oriented*. Nevertheless, the process versus product classifications need not be mutually exclusive, and attempts to partition software certification approaches based solely on the two properties could be misleading. Nothing prevents the use of both in a certification process. For example, the certification process could check that an approved process was followed and also provide an argument based on product evidence. Since process oriented certification regimes usually require compliance with some set of established standards, they are sometimes referred to as *standards based approaches*. Again, this could be misleading because nothing precludes a standard from being product focused. Similarly, it is not unusual to refer to product oriented certification regimes as *argument based approaches* because one has to come up with a convincing argument about one's product, often in the context of an argumentation theory based framework for presenting evidence; but, again, nothing *a priori*, excludes process considerations in the arguments.

In the context of argument based approaches, the way to construct the argument is surely not unique and is still under intense debate. In this context, we see recent proposals for goal oriented and template based presentations of arguments [14] (methods for realizing argumentation based approaches). Whether

the argument should be product focused, goal based, or evidence based is still an open question and this issue has given rise to more specific argument based approaches. In the balance between process and product emphasis in a given certification approach, we believe that the certifying agent should increasingly favour the product, as the criticality of the software increases, not because processes are irrelevant for the development of the product, but because they are not sufficient to prove anything definite about safety, effectiveness or correctness.

2.1 Process Based Approaches

Process based certification is common because we have been “able to do it”: it gives us the illusion, rather than a guarantee, that we have produced a good quality product and, therefore, a safe system. It is a lot easier to evaluate conformance to a development process than it is to decide on attributes that distinguish between dependable and unreliable software products and to be able to measure the relevant attributes effectively. Because process based standards model the products of the process superficially, if at all, it is impossible to characterize properly the properties of the entity we are actually interested in, namely the application we have built. Any process based definition of quality ends up guaranteeing only that certain steps and activities were undertaken, but does not offer direct evidence of the presence of desirable properties. A high quality process is not necessarily a reliable indication of a high quality product. Nevertheless, it is important to note that we believe that it is essential that the product be built by an organization that has excellent processes and excellent people, because this is likely to result in good products. It follows that certifying agents will be interested in these aspects, but they can usually be audited in a straightforward manner, often by a third party with no specific knowledge about the products developed by the audited organization, or its potential problems in relation to safety. The fact that a process standard has been adopted by some regulators may be evidence that the regulator believes that the standard is based on an implicit (generic) safety case that justifies the quality attributes of products produced using the process. However, the fact that this (generic) safety case is implicit contributes directly to the problems regulators are experiencing in evaluating applications based on adherence to the process standard. The developer and the regulator end up talking at cross purposes because of the lack of this common understanding. See below for further discussion of this point.

As a result, because process based certification approaches are inherently unable to guarantee the quality (in many cases, safety) that a regulator requires in the actual product, we need to focus on evaluating attributes of the product itself. This has generated interest in product focused certification approaches.

2.2 Product Focused Approaches

We might characterise process based approaches as providing indirect justification of a product’s required attribute values, as needed by regulators. In contrast

to the indirect justification of process based standards, the product(s) of the process contain the basis for the direct justification of claims to regulators. We refer to this focus on the attributes of the product, providing the basis for the direct justification of safety claims, as *product focused*, in contrast to the process based approaches discussed above. After all, it is the safety of the final product in which we are interested and not the efficacy or otherwise of the process used to produce the product. A product focused approach requires that the relevant attributes of the product can be modeled and that there are precisely defined measurement procedures in place to determine the values of these attributes for a specific instance. As we will see below, typical engineering domains have a very product focused approach to regulation, concentrating on the product itself, or on design artifacts directly related to the final product. This use of models and other design artifacts is made possible by the direct relationship of such design artifacts to the final product, underpinned by physical laws and the predictability inherent in these laws, as well as by the predictability of engineering methods in ensuring that crucial design properties are maintained.

It is often said that classical engineering is underpinned by the continuity principle, enabling the predictability of the artifact's properties from those of the models and design artifacts. While this may well be the case, the predictability of the typical engineering method is at least as crucial in ensuring that properties predicted on the basis of design artifacts will be realized in the engineering product. In this regard, the process standards often used in regulatory settings related to software are not proper engineering methods. They do not have the required detail and prescriptive qualities to support the requirement for maintaining essential properties of design artifacts through to final product. (An international process standard does not an engineering method make!)

3 What Do Civil Engineers Do?

For many years a debate has raged as to whether software engineering is really *engineering*. That debate still continues. We are convinced that software engineering should be a real engineering *profession*. It should be a *speciality* within engineering. Why? To answer that we have to understand a little about the differences between engineering and science - and the very practical reasons that led to the creation of engineering.

There are a number of differences between science and engineering that are of particular importance to any discussion related to software engineering [15].

- Scientists learn and/or discover what is true of our physical world. They learn how to confirm that these truths are, indeed, valid. They also learn how to extend this knowledge. Engineers also need to learn what is true of our physical world. They often learn about this from scientists. They learn how to apply that knowledge in the construction of artifacts that are useful, effective and safe for the general population to use. This often requires additional knowledge from many different domains. They also need to learn

the *engineering principles* applicable to their speciality that then enables them to build safe and effective products.

- Scientists have the validity of their work judged by their peers. Scientific results may be used by other scientists or by engineers who use the results to build something. The results on their own usually do not represent a danger to the public. On the other hand, one of the primary roles of the engineer is to build products that will be used by other people. In many cases, these products have to be constructed so that they do not pose undue risk to the public.
- A direct result of the danger inherently posed by the construction of engineered artifacts was the idea of the *licensed professional engineer*. Society recognized the problem of having unqualified people (people who did not know enough about the specific engineering domain) build potentially dangerous products. There is no similar regulation of scientific personnel.
- In many jurisdictions, not only are the people who build the products regulated (by law), but the products themselves may be subject to regulation that requires “proof” that the product is *safe for its intended use*. In addition, some jurisdictions (such as medical device regulation in the United States) require “proof” that the product is *effective*, i.e., it works as advertised.

It is also apparent that software products, both embedded in devices, and standalone applications, may pose significant risk to society. It is also obvious that software is pervasive, and the modern world depends on software for the functioning of commerce, transportation, entertainment, medicine, and the generation of electricity. Thus, an engineering speciality, called *software engineering*, built on knowledge from *computer science* and other domains makes perfectly good sense. So, why the debate? We believe that the debate is largely fuelled by our failure to model software engineering on other engineering specialities. This is not surprising since software engineering is relatively new, and the fundamental science it is dependent on is also relatively new. However, we have managed to make incredible progress in building software products that have literally changed the world we live in. So, we should not use its “newcomer status” too much as a reason not to have more consensus on the fundamentals of software engineering.

To help the discussion along, it will help to look at an engineering speciality that has a long history of building artifacts that are useful/essential to society, have to be built at reasonable cost, and have to be safe. We will regard this engineering speciality as representative of other engineering domains. So, after centuries of experience, how do *Civil Engineers* manage to build the modern infrastructure required by our civilization, protect us from environmental hazards, and do this at reasonable cost with an excellent (modern) safety record?

Well, Civil Engineers are famous for their development and use of *Engineering Codes*. For example, the *CSA Standard CAN3-A23.3, Design of Concrete Structures for Buildings* [16] (in conjunction with other standards) is used by Civil Engineers to ensure that concrete structures are safe for use. Before we look at some of the details regarding concrete structures, it is interesting and important to note a reminder in Section 1, General Requirements: “(3) The

National Building Code of Canada requires that certain reviews be carried out by the designer or another suitably qualified person to determine conformance with the design.” This is clearly a built-in certification/verification.

The following examples, taken from *CAN3-A23.3*, are instructive because they enable us to highlight essential differences between typical engineering regulatory practice (as exemplified by civil engineering) and typical software regulatory practice.

Example 1 refers to Section 15 which relates to *Footings*. In particular, isolated footings and (some) combined footings and mats. In Section 15.4.1 we find guidance on loads and reactions, and details relating to shaped columns and pedestals. Rules for determining the moment in footings are included. The rules are clearly prescriptive, and have been calculated to be conservative. They also allow for simplified calculations in the case of more complex shapes - also pre-determined to be conservative. Section 15.3 is note worthy, in that it clearly does not preclude a more specific analysis. A more specific analysis may be less conservative and allow for a more “creative” approach. There are other examples in which the more conservative approach is mandated, even if a more “accurate” method is available. By comparison, software regulation is hardly ever prescriptive with respect to analysis methods. This is partly due to the lack of consensus with regard to *standard* analyses in software engineering.

Example 1

- Section 15.4.1 *“The external moment on any section of a footing shall be determined by passing a vertical plane through the footing and computing the moment of the forces acting over the entire area of the footing on one side of that vertical plane.”*
- For example: 15.3 *“In lieu of detailed analysis, circular or regular polygon shaped concrete columns or pedestals may be treated as square members with the same area, for the location of critical sections for moment, shear, and development of reinforcement in the footings”.* □

Example 2 refers to Section 19 which relates to *Shells and Folded Plates*. Interestingly, as we can see in this section, the standard is also prescriptive with respect to the analysis assumptions that must be used. It also includes specific requirements on materials. By comparison, if software regulation does not prescribe analysis methods, it is certainly not likely to prescribe assumptions related to analyses. There are a number of software analogies we could use that relate to material. For instance, we could think of the programming language(s) used in an application as a fundamental component in much the same way as material may be thought of in constructing something physical. Using this analogy, it may be surprising that although most software regulation may not prescribe a programming language, many jurisdictions and manufacturers have imposed restrictions on programming languages to protect against known unsafe programming constructs.

Example 2

- Section 19.2.1 “Elastic behaviour shall be an accepted basis for determining internal forces and displacements of thin shells. This behaviour may be established by computations based on an analysis of the uncracked concrete structure in which the material is assumed linearly elastic, homogeneous, and isotropic. Poisson’s ratio of concrete may be assumed to be equal to zero.”
- 19.3.1 “The specified compressive strength of concrete, t'_c , at 28 days shall be not less than 20MPa”; and,
- 19.3.2 “For nonprestressed reinforcement the yield strength used in calculations shall not exceed 400 MPa.” □

Finally, Example 3 refers to Section 21 which relates to *Special Provisions for Seismic Design*. This rather lengthy section starts with three columns of notation and definitions, indicative of more complexity. Our motivation for selecting this example is that one of the reasons given for treating software differently from other engineering specialities, is the fact that software systems are typically quite complex. In fact, software systems are frequently amongst the most complex of human endeavours. Section 21 now reflects the additional complexity required to deal with constructing buildings that can withstand seismic events. For example, Section 21.4.4 deals with *Transverse Reinforcement*, and Section 21.4.4.2 specifies very prescriptive compliance requirements, for a problem that is inherently complex and could be solved in a large number of ways. In addition, the product can be checked for compliance both during and after implementation.

Example 3

Section 21.4.4.2

Transverse reinforcement, specified as follows, shall be provided unless a larger amount is required by Clause 21.7:

- (a) the volumetric ratio of spiral or circular hoop reinforcement, ρ_s , shall not be less than given by

$$\rho_s = (0.12f'_c/f_{yh}) \quad (21-2)$$

and shall not be less than that required by Equation (10-7);

- (b) the total cross sectional area of the rectangular hoop reinforcement shall not be less than the larger of the amounts given by Equations (21-3) and (21-4)

$$A_{sh} = 0.3 \frac{sh_c f'_c}{f_{yh}} \left(\frac{A_g}{A_{ch}} - 1 \right) \quad (21-3)$$

$$A_{sh} = 0.12 \left(\frac{sh_c f'_c}{f_{yh}} \right) \quad (21-4)$$

- (c) transverse reinforcement may be provided by single or overlapping hoops. Cross ties of the same bar sizing and spacing as the hoops may be used; and
- (d) if the factored resistance of the member core is greater than the factored load effect including earthquake, then Equations (10-7) and (21-3) need not be satisfied outside the joint.” □

The above examples show that *CAN3-A23.3*, like most engineering standards, imposes constraints and requirements on the product, and is prescriptive on the analysis process too. In comparison, most standards found in the software engineering domain adopt a risk-based approach and define a process-based generic approach for the software development lifecycle phases. One such standard that is emerging as a key standard for the functional safety of electrical/electronic/electronic programmable (E/E/EP) safety-related systems is IEC-61508, specifically parts 3 and 7, which relate to software requirements and techniques, respectively [17]. This standard assumes that it is not possible to prescribe techniques and measures that will be correct for any given application. It does, however, state that a primary objective of the IEC 61508 series is to facilitate the development of product and application focused standards, perhaps quite similar to *Engineering Codes* such as the one discussed above. Since testing is the most common way in which software engineers evaluate their products, let us examine what IEC 61508 says regarding the *Requirement for Software module testing* (IEC 61508-3, Section 7.4.7), which is part of the *Software design and development phase* (IEC 61508-3, Section 7.4). First, *outputs* of the given phase (or activity) are identified, such as the *test result*. Second, properties are identified:

- (1) completeness of testing with respect to the software design specification;
- (2) correctness of testing with respect to the software design specification;
- (3) repeatability;
- (4) precisely defined testing configuration.

Next, a list of techniques is *proposed*. For example, *dynamic analysis and testing* (DAT) is proposed. Then, the links between the technique and the properties are highlighted. In this case, it identifies that DAT will positively contribute to properties (1) and (2) but not to properties (3) and (4). This is followed by a more detailed description of suggested techniques related to DAT. For example, *test case execution from boundary value analysis* is proposed. Finally, specific guidelines drawn from experience, are given in Part 7 of IEC 61508. For example, *"the use of the value zero, in a direct as well as in an indirect translation, is often error-prone and demands special attention to zero divisor, blank ASCII characters, empty stack or list element, full matrix, or zero table entry."* As this example shows, the evolution of IEC 61508 is a small step towards more prescriptive and product focused standards or codes. However, as can be seen in the above discussion, even though this modern software standard sets out to be product and application focused, it falls far short of the type of prescription we see in other engineering disciplines.

The next section stresses some lessons we can learn from these examples.

4 What Can We Learn from Civil Engineering?

The kind of domain specific, tightly prescribed standard described above is essentially unfamiliar territory for software engineers and regulators working in

the area of software. We want to understand why analogous standards do not exist for software and we want to make the case that they should.

The following points relate directly to lessons we can learn from: i) the standard described in the previous section (and whenever the “standard” is mentioned in this section, it is that CSA standard on concrete design that we are referring to); and ii) civil engineering practice in general. In many cases throughout this discussion, Civil Engineering simply stands as an example of typical engineering disciplines. We also include a brief but pointed comparison with software engineering practice. The point of the comparison is to promote the idea that we can do better in the software domain and that, if we really want to have proper safety standards in software engineering, we should do better.

- In the balance between *safety* and *creativity/efficacy*, safety always wins. Architects and engineers are constrained, generally by statute, in what they are allowed to do. This is simply accepted as a way of life for everyone in the profession. This prescriptive regulation is also updated frequently to keep up with advances in the field. We should not underestimate the positive effect of smart prescriptive regulation. To comply with Canadian nuclear regulations, developers have to separate control and safety functions. The safety system must be completely independent of the control system. This has the effect of severely constraining the complexity of the safety system, leading to systems that can be mathematically defined and analysed. Contrast this with the almost complete absence of control of creativity in software and in safety cases! (See Bloomfield in Section 5.1.)
- The standard imposes constraints and requirements on the product. Compliance with these constraints and requirements can be determined objectively during and after completion of the implementation. This is because “compliance” is defined in the context of the standard scientific/engineering measurement framework (i.e., the MKS system of measures). There are well defined measurement procedures that can be used to determine whether or not some constraint or requirement has been met. Engineers typically do use them. Software professionals typically do not. More importantly, software standards impose much more stringent requirements on the development process than they do on the product.
- Although the standard is predominantly product focused, it is sometimes unashamedly prescriptive on the analysis process as well. In software we seldom encounter anything similar. There are some exceptions, as usual. For example, Modified Decision/Condition Coverage testing is mandated by a civil aviation software standard.
- The standard applies to all (concrete) products.
- Although this is not obvious from the referenced standard, Civil Engineering (like other engineering disciplines, e.g., Electrical Engineering) defines very limited interfaces. Even when very general interfaces are quite possible, standard interfaces cut the complexity of designs and implementations tremendously. Those simple interfaces allow technicians to assemble products without full-time guidance of the engineers. Software Engineering has

done almost nothing to limit our freedom to create interfaces. There has been excellent work done to cope with the complexity of software interfaces, but it may have been wiser to look also for ways of defining more standard interfaces. This may still be worth examining.

To some degree, in contrast to the lessons we have been trying to learn from the classical engineering disciplines, safety (and assurance) cases have been put forward as a kind of panacea for software engineering. We look more closely at safety cases in the next section.

5 Safety Cases

One approach that exhibits elements of both process based and product focused certification, depending on the nature of the evidence being adduced in the case, is that of the safety case. A safety case provides a structure in which the producer makes *claims* related to the safety of the product, and presents a *justification* as to why the claims are valid, using *evidence* related to and/or derived from the product. Latterly, argumentation theory has been proposed as a way of better presenting/structuring the safety case. This may make the safety case more comprehensible, but there is no reason to believe that it adds anything to its success in demonstrating safety or efficacy. This last claim requires justification. Before going any further, we should clarify what we mean by safety case (or assurance case, as for the moment we will not distinguish them). The literature on safety cases unfortunately exhibits a confused usage of the phrase. A safety case can obviously refer to an artifact, or product in the sense we use it in this article, namely the document containing the safety claims and the justification of those claims. The phrase can also be used as a name for an approach to demonstrating safety claims, as in the “safety case approach”. This usage refers to a methodology, i.e., a set or system of methods, principles, and rules for regulating a given discipline, as in the arts or sciences. For the safety case approach, the elements include hazard analysis, the use of claims, adducing evidence, argumentation principles, and so on. Finally, a well defined and prescriptive engineering method for constructing a safety case in a well defined domain may qualify as a method that falls within the safety case approach. An example of this usage could be the method of goal structured argumentation using argument templates, were it not for the fact that we do not believe that the existing literature describes anything amounting to a proper engineering method.

The certification goal in the Introduction points to what is missing from safety cases at all levels. The safety case approach, i.e., the methodology, is not sufficiently clear on the scientific and measurement foundations of the methods and principles of safety cases. The specific goal structured, argumentation method for safety cases is also not well founded in terms of scientific and measurement principles. Furthermore, it is not an effective engineering method, whose aim is to engineer safety cases in a reliable and predictable manner. Of course, the safety case artifact is itself deficient as it will not contain the relevant measurements and scientifically based justifications required to make the safety case

“sound”. It is this lack of proper scientific and measurement principles underlying safety cases, at all three levels, which justify our scepticism about efficacy of safety cases as noted above. We would make the same comments about assurance cases, which subsume safety cases and address issues other than just the safety of the system under consideration.

5.1 Why Safety Cases?

In stark contrast to the evidence from the classical engineering disciplines, Bloomfield and Bishop present five points as to why their notion of the safety case approach is to be preferred to more prescriptive regulation [13]:

- To prevent safety from being seen as the responsibility of the regulator rather than the service provider.
- Prescriptive regulation typically comes from past experience and this may be inappropriate in technically innovative industries.
- Prescriptive regulation encodes best practice at the time it was written, and may eventually prevent developers from adopting best practice.
- Overly restrictive regulation may be viewed as a barrier to open markets.
- Prescriptive regulation can adversely affect the cost and technical quality of products developed to comply with that regulation.

We will consider these points in turn.

“To prevent safety from being seen as the responsibility of the regulator rather than the service provider.” We have no quarrel with the first point at all! It is absolutely clear to us that the developer of a safety critical artifact must take responsibility for its safety. In fact, legally the developer has no choice, whether regulated or not. However, we do have a quarrel with the role that this statement plays in support of the argument against overly prescriptive standards. Prescriptive standards are perfectly consistent with developers taking responsibility for the safety of their product. We do not see why this is different for software compared with other specialties. Regulation occurs in all disciplines (and it is the system that gets certified, not the software). Prescription should cover a minimum safe set deemed necessary for ensuring the safety of the product.

“Prescriptive regulation typically comes from past experience and this may be inappropriate in technically innovative industries.” Most engineers would be offended at the accusation that, just because they have highly prescriptive regulatory regimes, they do not innovate. However, as Vincenti points out [18], there is innovation, and then there is innovation. In the distinction he makes between normal design and radical design, we see an inherent distinction between controlled and predictable innovation in normal design, and some or all bets are off innovations in radical design. Normal design embodies the experience based prescriptive design principles beloved of engineers. They are beloved exactly because they are reliable and predictable. However, as Vincenti points out, Just because they are prescriptive does not mean that innovation is stifled. The kind of innovation that is well supported in normal design is incremental, controlled innovation. For example, improving engine performance in a car or an airplane

by 2% is almost always an example of such controlled innovation. Replacing the usual car engine with a Wankel engine is not. This is an example of radical design and the predictability of the innovation's properties is much lower than for normal design.

So, in actual fact, engineers should avoid innovation, at least of a certain kind, exactly because they want predictability and safety. Radical innovation should have a very high price attached to it and should be stifled to some degree so as to prevent tragic accidents.

“Prescriptive regulation encodes best practice at the time it was written, and may eventually prevent developers from adopting best practice.” The remarks made above about technical innovation in products apply equally to the engineering methods used to produce them. The guarantees offered by normal engineering methods are well worth their value in providing support for safety engineering and do not prevent normal design innovation in the method. The engineering ethos encourages such improvement. However, the price of radical innovation in engineering methods is made clear; the burden of responsibility for radical changes, with respect to safety properties of systems, is clearly placed on the innovator. As it should be.

“Overly restrictive regulation may be viewed as a barrier to open markets.” As argued above, there should be some restrictions and barriers on open markets to make sure that the chances of damage to individuals and society are minimised. One has only to point to financial regulation, or the lack thereof, that was the major contributor to the world's recent economic catastrophes. The whole purpose of certification is to prevent the open markets from foisting dangerous products on us.

“Prescriptive regulation can adversely affect the cost and technical quality of products developed to comply with that regulation.” In the interest of keeping our response polite, we will only say that safety trumps productivity, and that prescriptive regulation does not automatically imply bad or overly prescriptive regulation. Also, as noted often, other engineering specialities are usually regulated through much more prescriptive standards than is software.

5.2 Some Weaknesses of Safety Cases

The freedom inherent in safety cases is appealing to software experts. It is appealing for exactly those reasons we believe that software engineering has resisted the constraints and discipline imposed in other engineering domains. As Vincenti [18] points out, engineers classically rely on established and recognised methods for designing artifacts, as the established method offers certain assurances about efficacy and safety. These assurances are backed up by standard analyses and measurement procedures associated with the relevant, specific engineering method being used for the design Vincenti calls this *normal* design, in contrast to *radical* design, where some element of a normal design method is absent, say because untried technology is being used. Software engineers have made very little attempt to develop a normal design culture, valuing their freedom to make the same errors over and over again! And to not learn from other software

engineers' experiences! From the regulators' point of view, this also makes their job well nigh impossible. Every submission is different, and so very difficult to evaluate in a systematic way. Regulators also need "normal evaluation" methods to work effectively and efficiently. Just as manufacturers have to use normal design principles for predictability of design, so regulators need to be able to apply normal evaluation procedures to reach reliable evaluation outcomes. It is not that radical designs cannot be submitted, but the evaluation process is then much more complicated and less certain in its judgement (see the discussion on stifling innovation in [13]). There is an obvious and important role for the safety case approach to play. In fact, we believe that safety cases should play this role in the certification of all products, be they software-enabled or not. This role is primarily as an overseer and organizer of safety-related principles and methods relating evidence and artifacts. Individual components within the safety case method should be domain specific and prescriptive certification strategies. For example, a safety case document for a medical device is likely to contain components that are software certification specific, electronic certification specific, etc.

A typical safety case document would be extremely time and resource consuming and potentially costly for certifying agents to evaluate. The fact that each safety case may be a *one-off* example means that certifiers would have to spend considerable time understanding the evidence and the importance of the evidence in each individual case. That is, the certifiers are left with the problem of rationally reconstructing the safety case method and the methodology/approach used in creating the safety case document. So, each safety case presents the certifier with a double scientific induction problem: one from safety case document to safety case method and the other from safety case method to safety case approach. This kind of induction problem is one of the most difficult kinds of problems for scientists to solve! They also have to spend significant amounts of time understanding the safety case structure, in each individual submission. Although ideas from argumentation theory have been proposed as a way of structuring safety cases, this does not actually address the issue at hand, as there is no concept of a "normal" or standard argumentation structure for a particular class of safety cases, domain specific or not. This may also result in an unpredictable submission process, much as we have at this time in many regulatory regimes. This difficulty may be ameliorated somewhat, but not completely, with the advent of safety case templates [19][13]. One might add that in those cases where a regulator relies on a process or quality standard, or both, as in the EU and Canada for medical devices, there is a third scientific induction problem faced by developers and regulators. The process/quality standard used is a stand in for a safety justification that motivated the structure and content of the standard. The idea is that if the standard is followed, then all the requirements of this implicit safety justification would be met. However, both the developer and the regulator have to guess why elements of the standard are required in relation to this unstated safety justification. This results in the need for the implicit safety justification to be induced by the developer and the regulator separately, possibly resulting in diverging interpretations of the implicit

case. This may explain to some degree the need for regulators, e.g., the U.S. FDA, to issue guidelines to accompany these standards - one can interpret the role of the guidelines as narrowing the gap between the interpretations of developer and regulator in relation to the implicit safety case. However, the present process/quality standards based approaches leave both regulator and developer with a triple scientific induction problem! How bad is that?

It is not good enough that the producer of the product supplies the evidence and the argument(s) in the safety case. What does matter is that the certifying agent cannot expect the same type of evidence and the same type of argument each time. The certifying agent thus has less chance of building expertise that will help in future submissions. This lack of expertise, or lack of appreciation of some of the finer points perhaps in the argument, may easily lead to the certifying agent not detecting a subtle flaw. Again, safety case templates could help alleviate this problem .

Argumentation frameworks, suggested for use in presenting safety cases, have some inherent flaws. An argument is not a derivation, as in logic. Hence, the methods of logic cannot be used, on their own, to decide whether an argument “demonstrates” its intended result. An argument has to be refined from its informal presentation in the safety case into a form that can be formally analysed [20]. There may be several or even many such refinements - there are inherent ambiguities in natural language based presentations of arguments. Does one check all of these refinements for “soundness” of the argument? If not all, then how do we choose which one? If we manage to analyse the refinement(s) and determine that we have a sound argument, how can we be sure that the argument formulation is appropriate to support the guarantee of safety? How do we judge that the safety of the artifact has been established by the argument? It seems to us that argumentation is essentially value free; here “value” is used in the moral or subjective sense. It may be that the concept of “explanation” from the epistemology of science may be a better basis for presenting safety cases. In scientific explanation, the issue is how to present a case for establishing in a scientific manner that some observed phenomenon can be logically demonstrated by certain known assumptions, certain scientific laws and scientifically understood procedures. On the other hand, a scientific explanation can also be used predictively. This may be the role related to safety cases - predicting a significant property of our software artifact and relying only on scientific/engineering principles.

Safety cases were designed to present compelling evidence of safety. In many instances, efficacy is also extremely important. The U.S. FDA, requires medical devices not just to be safe, but at least as effective as a similar device already on the market. The initial such device has to demonstrate, through clinical trials, that it does what it claims to do. The relationship between demonstrating these two properties is an interesting issue. There is almost always some tension between efficacy and safety, and safety cases were not designed to deal explicitly with this complication. In developing the shutdown systems for the Darlington Generating Station, demonstrating both properties was aided tremendously by

the separation of safety and control functions, significantly reducing the complexity of the safety system.

5.3 Standards Combining Process and Product Focus with Implicit Safety Cases

Another example that exhibits elements of both a process based and a product focused approach is the *Common Criteria* [21], developed for security certification. In the Common Criteria approach there are product definitions of varying degrees of exactitude. There are also definitions of measurement and evaluation procedures. What is missing is a proper product focus, due care and attention to the demands of measurement theory, and the demands of scientific explanation. Taking into consideration its included *evaluation methodology* [22], it may be regarded as an implicit safety case approach. However, the Common Criteria is more prescriptive than a typical assurance case approach, in that it specifies what products have to be produced for certification at a particular level, and provides the certifier with an evaluation methodology in [22] as part of the standard. Unlike engineering standards though, the evaluation methodology states that the “target audience . . . is primarily evaluators applying the CC [Common Criteria] and certifiers confirming evaluator actions; . . . developers . . . may be a secondary audience.” The developer can infer from [22] what evidence to produce, though not necessarily how it should be produced. To the developer, the safety case for the product behind the standard remains implicit and is effectively done by the evaluator as part of the evaluation process.

An approach that we have championed is an extension of some of the ideas and approaches used in licensing the Darlington Nuclear Generating Station’s Shutdown Systems in Ontario, Canada. The work done on the Darlington Shutdown Systems has actually been quoted as an example for what safety cases are designed to prevent [13]. In reality, the Darlington licensing activity that Bloomfield and Bishop refer to, was the original licensing of the systems in 1989/90. The licensing process was difficult for many reasons. Prime amongst these was that this was the first software based nuclear safety system to be built and licensed in Canada, and that both the regulators and the manufacturer had not developed a plan to deal explicitly with software issues. It thus transpired that regulator involvement in the software verification activities started after the system, including all the software, had already been developed. This initial licensing of the Darlington Shutdown Systems was described from the point of view of the regulators [23], and also by some of the team who performed the verification [24]. Our contention is that a safety case approach introduced at that stage would have fared no better. In fact, this is a good example of why we believe that of all the safety case approaches we have seen, the *assurance based* approach advocated by Graydon, Knight and Strunk [12] is likely to be the most successful. The difference here is that the safety case is used to drive development as well as build the safety argument. Nowadays, when we refer to the Darlington approach, it is to the methodology that was researched and implemented subsequent to the original verification. The software development (and verification) approach was

described briefly in [25]. As that approach was developed, it was discussed in detail with the regulatory authority in Canada. The resulting methodology enabled the regulators to conduct audits of evidence produced during the development process, significantly simplifying the certification process, and making it much more predictable for the manufacturer.

The foundations of that approach were laid in a *standard*, called the “Standard for Software Engineering of Safety Critical Software” [26]. This standard is reasonably prescriptive in defining the (quality) attributes that must be present in each of the major software documents that are mandated. This effectively defines what major steps in the software development process must occur, as well as how to judge whether the required attributes are present in the specified documents. Although it does not specify/mandate the actual process to be used, it does mandate that specific processes must be described in *lower-level process documents*. For example, [26] has a list of process documents that must be produced, that then govern the production of a specific project output. A small sampling of these includes: the *Software Requirements Specification (SRS)*, the *Software Design Description (SDD)*, the *Design Review Report (DRR)* and the *Design Verification Report (DVR)*. Each of these lower-level “standards” then mandates both a process to be followed, as well as the documentation that has to be produced. The lower-level standards were created so that the relevant software document governed by that standard would possess the quality attributes described in [26]. All of these standards together embodied an implicit safety case approach, although it was not viewed that way at the time.

The implicit safety case approach for the development of safety-critical software at Ontario Power Generation and AECL, in this case, is as follows:

1. The requirements are specified mathematically and checked for completeness and consistency. A hazards analysis is required to document risks and especially to identify sources of single point failures. These hazards have to be mitigated in the specified requirements.
2. Compliance between requirements and software design is mathematically verified. This includes a software design *review*, that evaluates how well the design exhibits mandated attributes. A prime example of such a design attribute is that of *maintainability*. Specific criteria are used to evaluate this, based on *information hiding*. This means that designers have to be able to demonstrate why they think that anticipated changes will be accomplished by changes to single design modules.
3. Compliance between the code and software design is verified through both mathematical verification and testing. Compliance between code and requirements is shown explicitly through testing. However, there is an implicit argument of compliance between code and requirements through the transitive closure of the mathematical verification - code to design, and design to requirements.

It should be noted though that the Darlington Shutdown System was a safety system that under normal circumstances did not intervene in operation of the plant. The system’s safe operation is a substantial part of its efficacy. The other

part of its efficacy is related to providing acceptable production, i.e. *not* intervening unless it is really required. It is also important to note that the implicit safety case approach actually includes more goals than does a typical safety case approach. The CANDU standard [26] was designed to demonstrate *correctness* of the implementation with respect to its requirements. In the example of the Shutdown Systems, we believe that demonstrating *correctness* is mandatory, since the complete system is a safety system. The modern movement away from trying to show correctness (primarily because we know that it cannot be achieved 100%) disturbs us. It is still not a bad goal!

5.4 Safety Case Improvements

As Bloomfield and Bishop [13] point out, there are a number of directions for the future development and improvement of safety cases. Mainly, they have identified the following directions:

- Safety Case Methodology Enhancement;
- Extension to Other Areas;
- Safety Case Structuring;
- Confidence.

Regarding methodology enhancement, we believe that the emphasis should be on prescriptive engineering methods that are domain specific, on domain and method specific analysis methods, and on well defined methods for combining analyses, i.e., the result of the combination defines some enhanced level of confidence over and above that engendered by the parts. Moreover, we need to examine how to strike an appropriate balance between analysis methods of different strengths/forms of guarantee, on the one hand, and a balance between direct product based evidence versus process based evidence, on the other hand. Another important issue is the problem of incremental certification, of which the COTS problem is just one instance. For safety case structuring, Bloomfield and Bishop [13] have identified interesting future directions. One of them is the use of diverse arguments and evidence. We believe that the diverse arguments idea is a mistake if it means that we should provide multiple arguments related to the same evidence: it confuses defence in depth within the system with different ways of arguing about the system. The latter is just confusing to the evaluator; it is demonstrating something by the bludgeon method!. If what was meant was, in reality, defence in depth, so that multiple justifications and evidence are provided to back up a single claim, then we agree wholeheartedly.

Different structuring methods for safety cases will certainly be appropriate in different domains and for differing levels of criticality. In each case, the structuring of the case must be derived from the requirements of a safety case for the combination of domain and level of criticality. This question of structure should be decided once and for all by the regulator so that there is a uniform understanding by applicants of what is required in an application for a licence.

Establishing levels of confidence goes to the very heart of the problems with safety cases as they are currently defined. The key ingredient missing, as noted

above, is the ability to make an objective, repeatable assessment of the confidence one should have in the safety of the system. The assessment procedure for determining one's confidence in a product's safety is no more and no less than a proper engineering method. This means that the main attribute we are interested in, the product's safety, must be a measurement based result, derived from directly measurable attributes by well defined functions appropriate for the purpose, i.e., the function is the result of a scientific analysis of the value to be attached to the various forms of evidence and to the functions used to combine them. This issue of determining levels of confidence in safety cases is "arguably" the most important issue for future investigation.

Early gains in improving safety cases could be made by removing the guesswork in using process based standards and actually reverse engineering the implicit safety case behind the use of the standard. Presumably, the regulators "trust" the process standard because they feel that it implicitly supports some sort of safety case; making this explicit would help all parties in standardizing requirements and expectations. The result will not be the ideal regime, but will be a big step towards it. At the very least, it removes one of the induction exercises identified in Section 5.2.

6 Conclusion

The safety case approach, and safety cases in particular, will probably play an important role in software certification. A concern is that current safety case approaches lack proper scientific and measurement principles, and it seems that many proponents of safety cases are overselling the method and its tools. One can look at our concerns and point to specific instances where the safety case will drill down to a level at which domain specific certification results are used, and quote these as an indication that safety cases do not preclude other certification processes at various steps in the safety case argument. However, if the safety case approach does not impose more prescriptive software dependent requirements for certification, we are not solving the basic software certification problem, or for that matter, the problem for embedded devices. It will still be possible, and maybe easy, for people to present apparently convincing arguments to substantiate their claims using evidence that they are free to choose. Of course, certifying authorities do not have to accept the validity of the evidence or of the argument - but that is no better, and in fact no different, from many certification regimes in existence today. Most of the arguments for less prescriptive regulation, some of which are presented in Section 5.1, seem unconvincing to us - and reminiscent of similar arguments over the years regarding software. Most of the arguments tend to favour creativity and "progress" over safety, which is exceedingly strange for safety cases. Nevertheless, an argument that seems to have some real merit is the one concerning responsibility. However, that has been dealt with adequately in other engineering jurisdictions and we see no reason why it should not be possible in software certification as well.

Moreover, the name *safety* case indicates the focus of the approach. However, in many domains it is not enough to produce a *safe* product. Other product

characteristics may compete with safety and therefore should be addressed in the certification process. For example, the FDA regulations explicitly state that medical devices must be proven to be both safe and *effective* and there seems always to be a tension between safety and efficacy. Another example comes from the nuclear industry where the nuclear power plant has to be proven *productive* in the sense that the generating station must not only be safe but it must also be cost effective in producing power. Therefore, concentrating on only one aspect certainly gets easier but it is also not realistic. If we take that thought to its logical extreme, the shutdown system for a nuclear generating station would be trivial to construct. It should always just shut down the plant. So, how do safety cases help us determine that the system is both safe and effective? We are not convinced that they can do this effectively.

Under the safety case approach, different improvements have been proposed recently. One of them is the generalization of safety cases to assurance cases where the latter can consider claims that not only relate to safety but could, for example, address effectiveness or productiveness. We believe that it is a step in the right direction, however, their scientific and measurement principles are, as in the case of safety cases, an open issue.

In conclusion, we believe that, in the future, an efficient certification process could be based on argument based evidence like safety cases or more generally assurance cases. But, similar to what is found in Civil Engineering, a “code” for building them must be defined. This code should be prescriptive, product focused, and should not only help and guide the product developers, but should also focus, reduce and clarify the audit process for the regulators.

References

1. Kornecki, A., Zalewski, J.: Certification of software for real-time safety-critical systems: state of the art. *Innovations in Systems and Software Engineering* 5, 149–161 (2009), <http://dx.doi.org/10.1007/s11334-009-0088-1>, doi:10.1007/s11334-009-0088-1
2. CBC Staff: Infusion pumps recalled in U.S. but not Canada. CBC News Online (May 2010), <http://www.cbc.ca/news/health/story/2010/05/04/con-baxter-pump.html>
3. Poulson, K.: Known software bug disrupts brain-tumor zapping. *Wired* (October 2009), <http://www.wired.com/threatlevel/2009/10/gamma>
4. Bogdanich, W.: Radiation offers new cures, and ways to do harm. *The New York Times Online* (January 2010), <http://www.nytimes.com/2010/01/24/health/24radiation.html>
5. Bogdanich, W., Rebelo, K.: A pinpoint beam strays invisibly, harming instead of healing. *The New York Times Online* (December 2010), <http://www.nytimes.com/2010/12/29/health/29radiation.html>
6. Software Certification Consortium: Software Certification Consortium Charter (Draft) (2010)
7. Jackson, D., Bloch, J., Dewalt, M., Gardner, R., Lee, P., Lipner, S.B., Perrow, C., Pincus, J., Rushby, J., Sha, L., Thomas, M., Wallsten, S., Woods, D.: *Software for Dependable Systems: Sufficient Evidence?* National Academies Press, Washington (2007)

8. Bishop, P., Bloomfield, R.: A methodology for safety case development. In: Redmill, F., Anderson, T. (eds.) *Industrial Perspectives of Safety-critical Systems: Proceedings of the Sixth Safety-critical Systems Symposium*, Birmingham, UK, pp. 194–203. Springer, Heidelberg (1998)
9. Rushby, J.: A safety-case approach for certifying adaptive systems. In: *Proceedings of AIAA Infotech@Aerospace*, Seattle, WA, American Institute of Aeronautics and Astronautics (April 2009)
10. Panesar-Walawege, R., Sabetzadeh, M., Briand, L., Coq, T.: Characterizing the chain of evidence for software safety cases: A conceptual model based on the IEC 61508 standard. In: *2010 Third International Conference on Software Testing, Verification and Validation*, pp. 335–344. IEEE, Los Alamitos (2010)
11. Fong, E., Kass, M., Rhodes, T., Boland, F.: Structured assurance case methodology for assessing software trustworthiness. In: *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement Companion (SSIRI-C)*, pp. 32–33. IEEE, Los Alamitos (2010)
12. Graydon, P.J., Knight, J.C., Strunk, E.A.: Assurance based development of critical systems. In: *DSN 2007: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 347–357. IEEE Computer Society, Washington, DC, USA (2007)
13. Bloomfield, R., Bishop, P.: Safety and assurance cases: Past, present and possible future - an adelard perspective. In: Dale, C., Anderson, T. (eds.) *Making Systems Safer, Proceedings of the Eighteenth Safety-Critical Systems Symposium*, Bristol, UK, pp. 51–67 (February 2010)
14. FDA Staff: Guidance for Industry and FDA Staff Total Product Life Cycle: Infusion Pump - Premarket Notification [510(k)] Submissions DRAFT GUIDANCE. U.S. Department Of Health and Human Services: Food and Drug Administration, Center for Devices and Radiological Health (April 2010)
15. Parnas, D.L.: Software engineering programs are not computer science programs. *IEEE Software* 16(6), 19–30 (1999)
16. Canadian Portland Cement Association Ottawa, Ontario, Canada: CSA CAN3 A23.3 M94 Concrete Design Handbook (1994)
17. IEC 61508: Functional safety of electrical/electronic/programmable electronic (E/E/EP) safety-related systems: Parts 3 and 7, International Electrotechnical Commission (IEC) (2010)
18. Vincenti, W.G.: *What Engineers Know and How They Know It: Analytical Studies from Aeronautical History*. The Johns Hopkins University Press, Baltimore (1993)
19. High, K.M., Kelly, T.P., Mcdermid, J.A.: Safety case construction and reuse using patterns. In: *16th International Conference on Computer Safety and Reliability (SAFECOMP 1997)*, pp. 55–69. Springer, Heidelberg (1997)
20. Parsons, T.: What is an argument? *The Journal of Philosophy* 93(4), 164–185 (1996)
21. *Common Criteria for Information Technology Security Evaluation: Part 1: Introduction and general model*. CSE, SCSSI, BSI, NLNCSA, CESG, NIST, NSA, Version 3.1 Revision 3 (July 2009)
22. *Common Criteria for Information Technology Security Evaluation: Evaluation methodology, Version 3.1, Revision 3* (July 2009)
23. Parnas, D.L., Asmis, G.J.K., Madey, J.: Assessment of safety-critical software in nuclear power plants. *Nuclear Safety* 32(2), 189–198 (1991)

24. Archinoff, G.H., Hohendorf, R.J., Wassying, A., Quigley, B., Borsch, M.R.: Verification of the shutdown system software at the Darlington nuclear generating station. In: International Conference on Control and Instrumentation in Nuclear Installations, Glasgow, UK, The Institution of Nuclear Engineers (May 1990)
25. Wassying, A., Lawford, M.: Lessons learned from a successful implementation of formal methods in an industrial project. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 133–153. Springer, Heidelberg (2003)
26. Joannou, P., et al.: Standard for Software Engineering of Safety Critical Software. CANDU Computer Systems Engineering Centre of Excellence Standard CE-1001-STD Rev. 1 (January 1995)

Testing Adaptive Probabilistic Software Components in Cyber Systems

Luqi¹ and Grant Jacoby²

¹ Naval Postgraduate School

² United States Military Academy

1 Introduction

This paper addresses improved principles for verification and validation to establish confidence in robustness of adaptive software systems, to include uncertainty with respect to cyber environment and dynamics of internal system configuration. It applies to component based systems with probabilistic decision making at multiple levels and bridges the gap between checking the correctness of a single component and validating systems composed of many components.

Robust adaptive software design requires substantial architectural support. Sound architectural models with adaptive probabilistic software components in cyber systems and associated quality assurance methods collectively gain the ability to replace bits of systems while maintaining system dependability [1]. We focus on system design with recorded rationale to make the testing of adaptive probabilistic software components in cyber systems possible.

This requires a shift from scenario-based testing to architecture-based quality assurance [2, 3, 4], along with a shift from code-based adaptation to architecture-based adaptation [5, 6, 7]. A good cyber system architecture would have associated dependability properties that express stable system requirements, requirements on the subsystems, and a sound software evolution model capturing the design rationale [8]. For systems that are supposed to be automatically adaptive, the adaptation is accomplished by structures explicitly visible in the software architecture. The architecture itself would provide some degree of dependability guarantees, regardless of specific configuration. Testing and analysis would be applied to a suitably specified architectural model in addition to the system implementation, as described in section 4. Realizing this vision will require a more precise and detailed description of the architecture than is commonly developed in current practice

Robust composite systems can be composed of components in a disciplined manner, according to a carefully designed software architecture. A *software architecture* consists of:

1. A set of components
2. An interconnection pattern for the components, and
3. A set of constraints on the components and connections.

The constraints typically express various kinds of requirements associated with the entire systems as well as with the components and connections of the architecture [9, 22].

An *architecture model* is an explicit and precise definition of an architecture that specifies the details of its structure and the constraints. Architecture models play a crucial role in the design and testing of flexible systems, especially for those that are supposed to be self-adapting at runtime. In this context, the constraints associated with the entire system express invariant requirements that are supposed to be met in all the configurations that can be reached via adaptation transitions. The constraints on the components and the connections express the common principles of operation shared by all of those configurations.

In a well-designed adaptive system the invariant requirements can be established based solely on the constraints associated with the components and the connections. This enables the system to operate properly regardless of which concrete components are chosen to fill each component slot in the architecture model, as long as each concrete component satisfies the constraints associated with its component slot. In this vision, each component slot has a set of compatible components, all of which satisfy the role requirements associated with a given slot in the architecture, but the individual components may differ in the specifics of their behavior within the envelope defined by the role requirements. This set may be further organized according to feature parameters that characterize the degree of component behavior variability that can be accommodated by the architecture model.

The individual components associated with a given slot in the architecture have to be certified with respect to the role requirements associated with the slot. All variants of these components must satisfy the minimum common role requirements. In cases where alternative slot-compatible components have different specialized capabilities, the role requirements may include corresponding specialized requirement clauses that are conditioned on the particular feature parameter values that call out the sub-requirements for those specialized capabilities.

2 How Testing for Adaptive Systems Differs from Traditional Testing

Traditional software testing techniques like scenario-based integration testing are commonly used for assessing dependability of today's software systems. These techniques are strongly dependent on a particular system configuration and its platform. A major drawback is that when the system configuration or its platform changes, it is necessary to reconstruct the test cases and rerun them. Plugging in a new software component will lead to a completely different system and will likely invalidate previous test results, while changes to the cyber environment may reduce the effective coverage of the test scenarios previously used. Therefore, these techniques are not effective for testing adaptive probabilistic software components in cyber systems where dynamic system configuration and frequent changes are the norm.

Cyber-physical systems are an important special case of adaptive systems [10]. *Cyber-Physical Systems* (CPS) are integrations of computation and physical processes. Embedded computers and networks monitor and control the physical processes, with feedback loops where physical processes affect computations and vice versa. The technology builds on the older (but still very young) discipline of embedded systems, computers and software embedded in devices whose principle mission is not

computation, such as cars, toys, medical devices, and scientific instruments. CPS integrate the dynamics of the physical processes with those of the software and networking, providing abstractions and modeling, design, and analysis techniques for the integrated whole. [28].

A cyber-physical system consists of software, computers, and physical components. The software in such systems is typically used to control the physical parts in a way to achieve given goals such as safety concerns, maximizing system utility, and minimizing costs. This control is typically based on sensor readings, and signals that control actuators, both of which have finite accuracy and delay. In large scale contexts, there are generally a variety of goals to be achieved, a variety of physical components to control and coordinate, multiple available sensors and actuators, environmental conditions cannot be perfectly predicted, and all physical components are subject to failure. In many contexts of interest, the cyber-physical systems are expected to operate in inhospitable and possibly hostile environments [8, 11].

A similar problem also occurs when the application has to be used in a cyber environment different from the one for which it was originally designed and tested, which is expected to be common for reusable adaptive probabilistic systems and components. This raises an important concern since flexible systems are subjected to frequent changes. In the context of automatically adaptive systems, the situation is even more severe, since adaptation is supposed to happen at runtime, rapidly, and with little or no human intervention. This does not provide enough time for traditional test and evaluation procedures, which are typically time consuming, labor intensive, and dependent on human expertise. We therefore seek techniques that can be applied in advance without detailed knowledge of the specifics of the new configuration and the new operating environment. We need to find principles and methods for quality assurance that can apply to a family of possible new configurations and a family of possible cyber environments simultaneously [12].

Traditional quality assurance approaches rely heavily on testing methods that assume system environment and system configuration are fixed and known prior to testing. These assumptions are invalid for adaptive complex probabilistic systems. Noisy data, deception, and surprise attack tactics are likely for many critical and military systems, making system environment uncertain.

System adaptation and machine learning are inconsistent with keeping system configuration fixed. We propose to model these phenomena as the choice between different configurations that share the same principles of operation and architecture, but differ in specific components that can be compatibly plugged into corresponding slots in the architecture. Probabilistic systems can be also be modeled in this framework, as nondeterministically choosing between components at runtime, where each of the alternative components realizes a different decision strategy. An adaptive system must be designed so that it can continue to operate properly despite replacement of components with other plug-compatible components that may differ in the details of their behavior and capabilities. For example, this strategy is applicable when a variety of robotic platforms are built from a designated set of plug-compatible electronic components, each of which has its own specialized purpose and has been designed to accomplish a specific aspect of a complex mission [29].

Fully realizing the dependable architecture vision thus requires new paradigms for both system synthesis and quality assurance. We propose such a paradigm here, based on the concepts of mathematical dependability contracts, interchangeable software parts, and computer-aided enforcement of dependability contracts, both during V & V and at runtime. For adaptive systems, a dependable architecture includes an explicit adaptation model, with its own requirements, structure, principles of operation, and desired dependability properties. For probabilistic systems, requirements will be expressed in terms of constraints that admit multiple, non-deterministic outcomes. Deterministic systems become a special case, in which the constraints are tight enough to admit only a single behavior in each situation. We assume that multi-component systems will be designed using adaptive architectures that define the roles of the individual components along with the protocols for the interactions among the components and parameterized goals that are to be achieved by the interactions, using a refinement of the framework described in [13].

We focus primarily on the V & V aspect, but include aspects of synthesis in cases where constrained design techniques that guarantee certain aspects of correctness by construction via special-purpose design rules, analysis and checking at design time are more effective than methods that seek to certify arbitrary unconstrained designs.

Current approaches to system development and testing are more analogous to individual craftsmanship than they are to modern concepts of mass production and interchangeable parts. Craftsmen used to build things by individually tuning mating parts until they properly fit together. In such a context, designs could be relatively informal and relatively rough. Current software testing practices work this way, particularly for integration testing. Such approaches are not feasible for adaptive systems because the number of possible configurations is much too large (exponential in the number of independently adaptable components) to feasibly test them all in advance.

In modern mass production environments, parts are built to standards with precisely specified tolerances, and it is up to the designer to determine and verify the tolerances necessary to make the design work for any combination of parts that meet the specified tolerances. For example, modern audio systems are designed this way. There exist specific standards for audio specifying how things need to fit together in order for components from different vendors to work together effectively, and components are tested based on these standards, without knowledge of the specifics of the other components they will be connected to. Determining the tolerances necessary to make this work is difficult in general, and current practice depends on highly skilled individuals to get the standards right. However, standards for audio system components can be relatively simple and manageable via relatively informal processes only because the requirements for stereo systems are very simple and insensitive to the meaning of the signals they are processing. An audio system is not concerned about whether it is playing a song or the news. This simplification does not apply to most software systems.

3 The Scientific Problem

We are seeking analogous synthesis and quality assurance techniques for systems involving software. For software systems, whose behavior is usually sensitive to the

meaning of the data, new types of standards will be needed to accomplish a similar function, along with scientific methods for designing and checking the standards to ensure the system will meet its dependability properties, as well as checking conformance of individual components to the standards. This additional degree of rigor is needed for software standards intended to meet specific requirements despite varied choices of components (system adaptation) or nondeterministic variations in the behavior of a single component (probabilistic systems) because such standards are considerably more complex than the standards supporting mass production for electronic or mechanical systems. We are studying principles, models, and methods to enable general static analysis techniques to be combined with testing methods that can support statistically significant conclusions about conformance of components to architecture standards. Targeted static analyses range from techniques involving general mathematical proofs requiring human assistance to special purpose algorithmic methods that can be completely automated for complex applications.

One context in which the proposed approach can be employed is in the design of interchangeable components for robotic platforms. Here the focus is on cooperative robotic platforms where payloads, sensors and tasks are divided into various specialized modular platforms. The modular specialized platforms can then be assembled as a team, custom tailored for the various mission requirements. Decomposing the complex task sequences required of autonomous robots into a hierarchy of increasingly sophisticated control systems provides a powerful method for escalating a robot's degree of autonomy [29]. Analogous synthesis and quality assurance techniques in designing the software for this, or different levels of autonomy at different levels of control, is needed to form a hierarchical command control structure that can deploy and orchestrate a network of autonomous robots.

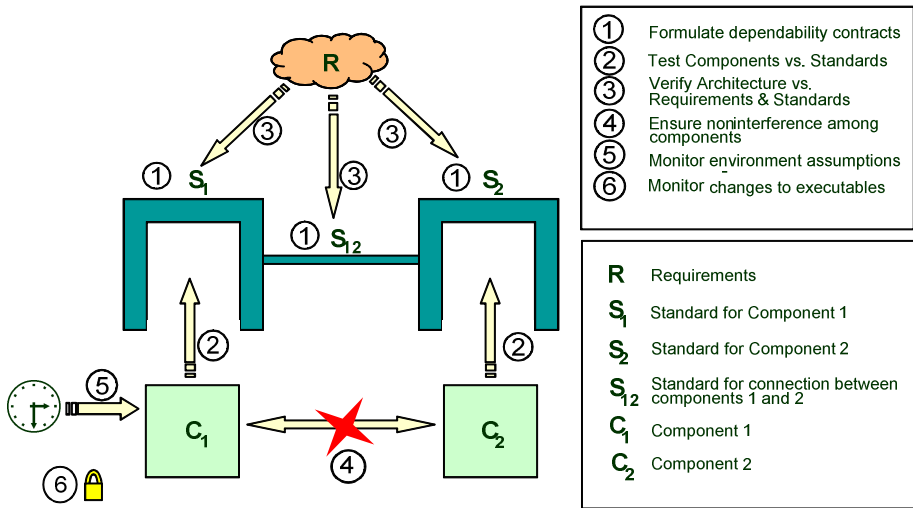
4 How to Test Adaptive Systems

The proposed approach to quality assurance for adaptive systems encompasses:

- (1) analysis of software architecture with respect to requirements on many possible configurations,
- (2) testing software components against required architecture properties driven by both validated statistical models of expected cyber environments and transformed models, and
- (3) a combination of testing and analytical methods for verifying non-interference between components.

The transformed models compensate for uncertainty about possible cyber environments by combining alternative possible models and amplifying the frequency of rare events.

A possible quality assurance approach for adaptive systems is illustrated below [14, 15, 31].



The fundamental operation of such an approach can be outlined as follows:

- 1.a. System-wide capabilities are characterized by a set of dependability properties that must hold in all acceptable system configurations. These properties comprise the dependability contract for the system as a whole. They become part of the architecture for the system, and serve as the basis for system quality assurance. Dependability contracts are primarily technical rather than legal documents, and they are intended to be checkable via software.
- 1.b. The designers of the architecture determine the common structure of the system and develop the component-level dependability contracts for the subsystems and connectors. The common structure consists of connection patterns and subsystem slots to which all configurations must conform.
2. The quality assurance team tests each component (subsystem and connector) against its dependability contract. This is envisioned to be an automated process to enable sufficient large sets of test cases for statistically significant conclusions about desirable dependability levels. The cost for this step is proportional to the number of components, and the process must be done once for each version of each atomic component. Technologies for doing this are known, and some of them are starting to be used in practice [30].
3. The quality assurance team checks the structure of the architecture and the dependability contracts for subsystems and connectors to make sure they are strong enough to guarantee the system-wide dependability properties in all possible configurations. This is a new process that uses symbolic analysis techniques. Assuring the feasibility of this step is one of our objectives.
4. The quality assurance team checks components for non-interference. This process is computer-aided. Many of the technologies for this are known and some of them are commonly used, such as data representation hiding enforced by programming language scope rules and type checking. Additional research is needed to get a complete set. This part of the process ensures that

components that work correctly in isolation will continue to do so when they are connected. Some examples of mechanisms that can cause this type of interference include mutual exclusion constraints, deadlocks, resource constraints (memory and bandwidth limits, etc), timing constraints, out of bounds memory references, etc.

5. The assumptions about the operating environment that the architecture depends on are checked by runtime monitoring. This can be done using BIT (Built-In-Test) technology that is currently in use in some DoD systems. This is recommended for all reusable components.
6. Runtime checks that the machine code actually running in the system corresponds to the source code that was subjected to quality assurance processes, and processes for restoring it to the proper state before execution if these checks fail. These processes are necessary because of the following plausible failure modes:
 - a. Memory-corrupting bugs—these include out of bounds write operations on arrays and through invalid pointers. Such bugs can cause seemingly innocuous statements to overwrite parts of the program itself at runtime, with unpredictable and potentially catastrophic results.
 - b. Deliberate cyber-attacks—compromise of system security via network or unauthorized insider access to systems can deliberately modify machine code at run-time.
 - c. Memory state corruption due to hard radiation, which is plausible in some WMD scenarios, spacecraft, nuclear power systems, scientific and medical applications, etc.

Some approaches to this problem can be found in [15].

Probabilistic methods for testing flexible modular systems to high levels of statistical confidence should be viable for testing complex probabilistic systems. This strategy focuses on using high-fidelity profile-based environment models to automatically generate test cases by sampling from probability density functions or other kinds of probabilistic models that characterize environment parameters. When combined with automated execution and output checking techniques, such models are capable of driving automated software testing, enabling affordable sample sizes large enough to support statistically significant conclusions about system behavior.

5 Challenges in Validating Statistical Models for a Wide Range of Operating Environments

Known methods for creating validated statistical models for operating environments include least-squares matching of known parametric distributions, use of the Bayesian Information Criterion or Akaike Information Criterion to choose the best of several models, and non-parametric methods such as kernel density estimation. All of these methods depend on the availability of historical data characterizing a given cyber environment, and assume that the intended cyber environment is known in advance.

These assumptions are problematic for adaptive systems, because the actual system configuration is not necessarily known in advance. In the context of unprecedented systems, the availability of historical data is also problematic.

Prior work has explored some approaches to modeling of unknown operating environments in the context of software filters for maritime tracks [15], and we are studying, refining and extending these approaches. The legacy operating environment for the previously studied software consisted of large, slow moving ships. The new environment included threats from small fast boats, which are likely to be operated by dynamic non-state organizations such as terrorist groups or smugglers. The specific types of boats involved in such cases are largely unpredictable, as are the likely navigational tactics. The solution approach in this situation was to characterize the operating characteristics of a variety of commercially available small boats, in terms of parameters such as maximum speed, maximum acceleration, maximum deceleration (separately specified because speeding up and slowing down have different mechanisms that produce asymmetric results), and maximum turning rate. These parameters defined the limits of a neighborhood of environmental conditions of reasonable concern. Kernel density estimation techniques were then used to form a composite of the different possibilities for operating conditions. Although the resulting distribution covered a wider range of conditions than each individual possible type of boat was capable of exhibiting, it provided a reasonable characterization of all conditions likely to be of concern, with tails that provided coverage of unexpected but possible conditions. Our initial study was conducted in a relatively simple two-dimensional environment. However, we see no fundamental reason why similar techniques could not be applied in more complex data domains and higher dimensional spaces.

Adaptive systems must deal with equivocality in addition to uncertainty. Here “uncertainty” refers to individually unpredictable events that follow a known probability distribution, and “equivocality” refers to situations where the probability distribution itself is uncertain, or possibly time varying according to unknown patterns. Uncertainty corresponds to the case where the expected operating environment can be characterized in terms of stable types of activities that have a known past history, while equivocality corresponds to more dynamic situations in which unexpected and possibly unprecedented new kinds of events can arise. We have modeled equivocality in the context of possible operating environments under the assumption that “small” perturbations to operating environments are more likely than “large” ones. This leads to systematic exploration of sequences of possible environments produced by perturbing transformations systematically arranged in order of increasing severity. One way to realize this in a way that captures likelihood of occurrence is to repeatedly compose the perturbation transformation to form more severe perturbations. This overall approach is compatible with a strategy that searches the most likely neighborhoods of the expected operating environment first and most thoroughly.

6 Conclusion

The usefulness of sound and systematic approaches for achieving dependable, flexible, and cost-effective software has been a focus of scientific interest for many years

[16-35]. As we move to automatically adaptive software, these types of approaches are appearing to become a necessity rather than a luxury, because manual artisanship simply cannot reach the levels of reliability and speed of adaptation that is called for.

References

1. Hinchey, M., Vassev, E.: Software Verification of Autonomic Systems Developed with ASSL. In: Monterey Workshop on Modeling, Development, and Verification of Adaptive Systems, Microsoft Research, March 31-April 2, pp. 7–15 (2010)
2. Luqi, Zhang, L., Berzins, V., Qiao, Y.: Documentation Driven Development for Complex Real-Time Systems. *IEEE Transactions on Software Engineering* 30(12), 936–952 (2004)
3. Qiao, Y., Luqi: Admission Control for Dynamic Software Reconfiguration in Systems of Embedded Systems. In: Proceedings of the 2004 International Conference on Embedded Systems and Applications (ESA 2004), Las Vegas, Nevada, USA, June 21-24, pp. 136–142 (2004)
4. Luqi: Transforming Documents to Evolve High-Confidence Systems. In: Proceedings of Workshop on Advances in Computer Science and Engineering, Berkeley, CA, May 6, pp. 71–72 (2006)
5. Oreizy, P., Gorlick, M., Taylor, R., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D., Wolf, A.: An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems and their Applications* 14(3), 54–62 (1999)
6. Garlan, S.W.C., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer* 37(10), 46–54 (2004)
7. Calinescu, R.: Reconfigurable Service-Oriented Architecture for Autonomic Computing. *International Journal on Advances in Intelligent Systems* 2(1), 38–57 (2009)
8. Rajkumar, R., Lee, I., Sha, L., Stankovic, J.: Cyber-Physical Systems: The Next Computing Revolution. In: Proceedings of the 47th Design Automation Conference, pp. 731–736 (2010)
9. Proceedings of the 4th Monterey Workshop on Specification Based Software Architectures, Monterey, California (1995)
10. Proceedings of the 16th Monterey Workshop on Modeling, Development, and Verification of Adaptive Systems, Microsoft Research, March 31-April 2 (2010)
11. Luqi, Kordon, F.: Modeling, Development, and Verification of Adaptive Systems. In: Proceedings of the Monterey Workshop on Modeling, Development, and Verification of Adaptive Systems, Microsoft Research, March 31-April 2 (2010)
12. Sztipanovits, J.: Software Verification for Adaptive Systems: is it Easier or Harder? In: Proceedings of the Monterey Workshop on Modeling, Development, and Verification of Adaptive Systems, Microsoft Research, March 31-April 2 (2010)
13. Luqi: Dependable Software Architecture Based on Quantifiable Compositional Model, AFOSR Final Report (November 2007)
14. Berzins, V., Rodriguez, M., Wessman, M.: Putting Teeth into Open Architectures: Infrastructure for Reducing the Need for Retesting. In: Proceedings of the 4th Annual Acquisition Research Symposium: Creating Synergy for Informed Change, Monterey, CA, May 16-17, pp. 285–312 (2007)
15. Dailey, P.: High-Fidelity Profile-Based Automated Testing of Open Architecture Track-Processing Software, NPS Ph.D. Dissertation (June 2010)
16. Luqi, Goguen, J., Berzins, V.: Formal Support for Software Evolution. In: Proceedings of Monterey Workshop 1994, Monterey, CA, September 7-9, pp. 10–21 (1994)

17. Luqi, Berzins, V.: Software Architecture in Computer-Aided Prototyping. In: Proceedings of 1995 Monterey Workshop on Increasing the Practical Impact of Formal Methods in Computer Aided Software Development: Software Architecture, Monterey, CA, September 12-14, pp. 44–57 (1995)
18. Luqi: Formal Models and Prototyping. In: Proceedings 1997 ARO Workshop on Requirements Targeting Software and System Engineering - Towards a Scientific Basis, Munich, Germany, October 12-14, pp. 183–194 (1997)
19. Luqi: Engineering Automation for Computer Based Systems. In: Proceedings of the 1998 ARO/ONR/NSF/DARPA Monterey Workshop on Engineering Automation for Computer Based Systems, Monterey, CA, April 1998, pp. 3–8 (1998)
20. Luqi, Nogueira, J.C.: A Risk Assessment Model for Evolutionary Software Projects. In: Proceedings of the 2000 Monterey Workshop on Modeling Software Systems Structures in a Fastly Moving Scenario, Santa Margherita Ligure, Italy, June 13-16, pp. 208–215 (2000)
21. Murrach, M., Johnson, C., Luqi: Enhancements & Extensions of Formal Models for Risk Assessment in Software Projects. In: Proceedings Monterey Workshop 2001, Engineering Automation for Software Intensive System Integration, NPS, Monterey, CA, June 18-22, pp. 120–127 (2001)
22. Luqi, Qiao, Y., Zhang, L.: Computational Model for High-confidence Embedded System Development. In: Monterey Workshop – Radical Innovations of Software and Systems Engineering in the Future, Venice, Italy, October 7-11, pp. 265–303 (2002)
23. Luqi, Zhang, L.: Documentation Driven Agile Development for Systems of Embedded Systems. In: Monterey Workshop Series: Workshop on Software Engineering for Embedded Systems: From Requirement to Implementation, Chicago, IL, September 24-26, pp. 13–25 (2003)
24. 11th Monterey Workshop: Software Engineering Tools: Compatibility and Integration, Vienna, Austria (2004)
25. Luqi, Berzins, V., Roof, W.: Nautical Predictive Routing Protocol (NPRP) for the Dynamic Ad-Hoc Nautical Network (DANN). In: Monterey Workshop 2005: Realization of Reliable Systems on Top of Unreliable Networked Platforms, Univ. of California, Irvine, USA, Laguna Beach, September 22-24, pp. 1–9 (2005)
26. Luqi, Ivanchenko, V., Rodriguez, M., Berzins, V.: Reliability and Flexibility Properties of Models for Design and Run-time Analysis. In: Monterey Workshop 2006 Composition of Embedded Systems: Scientific and Industrial Issues, Paris, October 16-18 (2006)
27. 15th Monterey Workshop: Foundations of Computer Software, Future Trends and Techniques for Development, Budapest, Hungary (2008)
28. Demir, Y.: JOGL: An OpenGL Based Graphics Domain for Ptolemy II. In: Cyberphysical Systems Education Workshop (CPSEW), Arlington, VA, August 12 (2010)
29. Jacoby, G., Chang, D.: Towards Command and Control Networking of Cooperative Autonomous Robotics for Military Applications (CARMA). In: CCECE/CCGEI Conference, Niagara Falls, Canada, May 5-7 (2008)
30. Porter, A., Memon, A., Yilmaz, C., Schmidt, D., Natarajan, B.: Skoll: A Process and Infrastructure for Distributed Continuous Quality Assurance. *IEEE Transactions on Software Engineering* 33(8), 510–525 (2007)
31. Luqi, Dailey, P.: Profile-Based Automated Testing Process for Open Architecture Track-Processing Software. Technical Report #NPS-CS-10-005 (March 2010)
32. Niebuhr, D., Rausch, A., Klein, C., Reichmann, J., Schmid, R.: Achieving Dependable Component Bindings in Dynamic Adaptive Systems - A Runtime Testing Approach. In: Proc. Third IEEE International Conference on Self-Adapting and Self-Organizing Systems, San Francisco, CA, September 14-18, pp. 186–197 (2009)

33. Prasanth, R., Boskovic, J., Mehra, R.: Computational Methods for the Verification of Adaptive Control Systems. In: Proc. SPIE 5429, 264, Orlando, FL, April 12, pp. 264–272 (2004)
34. Schaefer, I., Poetzsch-Heffter, A.: Model-Based Verification of Adaptive Embedded Systems under Environment Constraints. ACM SIGBED Review - Special Issue on the 2nd International Workshop on Adaptive and Reconfigurable Embedded Systems, APRES 2009, 6(3) (October 2009) (no page numbers used)
35. Zhang, J., Goldsby, H., Chang, B.: Modular Verification of Dynamically Adaptive Systems. In: Proc. AOSD 2009, Charlottesville VA, March 2-6, pp. 161–172 (2009)

Author Index

- Astesiano, Egidio 157
- Baarir, Souheib 103
- Bherer, Hans 206
- Butts, Kenneth 55
- Calinescu, Radu 122
- Derler, Patricia 55
- Fortes, José A.B. 77
- Grönniger, Hans 17
- Haxthausen, Anne E. 176
- Hillah, Lom-Messan 103
- Hinchey, Mike 1
- Jackson, Ethan 33
- Jacoby, Grant 228
- Kadirvel, Selvi 77
- Kang, Eunsuk 33
- Kikuchi, Shinji 122
- Kordon, Fabrice 103
- Lakos, Charles 136
- Lange, Douglas S. 193
- Lawford, Mark 206
- Leotta, Maurizio 157
- Luqi, 228
- Maibaum, Tom 206
- Naderlinger, Andreas 55
- Petrucci, Laure 136
- Pree, Wolfgang 55
- Reggio, Gianna 157
- Renault, Etienne 103
- Resmerita, Stefan 55
- Ricca, Filippo 157
- Rumpe, Bernhard 17
- Schulte, Wolfram 33
- Vassev, Emil 1
- Wassyng, Alan 206