10th Monterey Workshop

Software Engineering for Embedded Systems: From Requirements to Implementation

September 24 – 26, 2003

Hosted at Chicago Illinois

Table of Contents

Software Synthesis from Hybrid Automata

Rajeev Alur, *University of Pennsylvania*

Abstract:

Benefits of high-level modeling and analysis can be significantly enhanced if the code is generated automatically from the model such that the relationship between the two is rigorously understood. For embedded control software, hybrid systems is an appropriate modeling paradigm due to the ability to specify continuous dynamics as well as discrete switching. In this talk, we will discuss the challenges involved in code generation from hybrid models. In particular, we argue that, for portability and modularity, the traditionally separate steps of discretizing the continuous controllers, determining the sampling frequencies, and scheduling of periodic tasks, should be integrated. We will describe some of the ongoing work to achieve this goal.

Information Consistency Checking in Documentation Driven Development for Complex Embedded Systems

Valdis Berzins, Lisa Y. Qiao, Luqi

Naval Postgraduate School, Software Engineering Automation Center
Monterey, CA93943
{berzins, yqiao, luqi}@nps.navy.mil

Abstract Complex embedded systems, especially systems of embedded systems (SoES) need documentation to support their development. In our research, we are developing a documentation driven development method for SoES. In this method, keeping high confidence properties consistently identified in documentation of different development phases is an important issue since it is critical to ensure software quality of the end product. To address this issue, in this paper we investigate a method for information consistency checking in documentation driven development for SoES. We present an attributed object graph model to describe the semantics of document elements. Based on this model, we show how a set of attribute computation rules can analyze consistency between the key information such as timing properties transformed from one development phase to another.

Keywords Documentation, Systems of Embedded Systems, High Confidence, Attributes, Semantics

1. Introduction

1.1 Background

Complex embedded systems that are widely used today are usually deployed for long periods of time. They usually have mission critical requirements and demand real-time and high-confidence performance. These complex embedded systems, known as systems of embedded systems (SoES)[1], are composed of component systems that were developed by different organizations with different tools and run on different platforms. Furthermore, they must rapidly accommodate frequent changes in requirements, mission, environment, and technology. These traits make software development for systems of embedded systems face several challenges. First, key properties of embedded systems, such as high-confidence properties are hard to keep consistent during the whole development process, making software quality difficult to ensure in the end product. Second, a wide variety of stakeholders (sponsors, developers, users, maintainers, etc) are involved in the overall lifecycle of the software. Inconsistent information among different stakeholders is one of the main factors resulting in design faults. Third, complex embedded systems are difficult to evolve and maintain because of the independent development of their constituents and frequent changes in circumstances.

Previous research on embedded system development revealed that documentation plays a crucial role in coping with the above challenges throughout the software life cycle. According to the FIPS PUB 105 definition, documentation refers to all information that describes the development, operation, use, and maintenance of computer software. This information is in a form that can be reproduced, distributed, updated, and referred to when it is needed [2]. Furthermore, software documentation should provide information to support all software life cycle processes, most notably, requirements gathering, quality assurance, design, system evolution and reengineering, project management, communication among all system stakeholders and communication with software tools.

1.2 Related Work

Software Engineering aims to improve software quality and productivity by providing systematic, disciplined and quantifiable approaches to software development. Documentation has been proven to play a key role in software engineering. Many theories, methods, and techniques related to documentation have been developed in the past decades. There are different specific documents associated with different development phases. Typical phases in the software life cycle include requirements analysis and definition, architectural design, implementation, composition, deployment, maintenance and evolution.

In the requirement phase, a requirement definition, which is a kind of documentation, serves as a starting point for the whole software development process. Natural language is the most common form of requirement definition [3]. By modeling and formalizing the requirement definition, the formal documentation – the requirement specification – can be derived. In this case, the requirement specification is usually written in formal language. Typical examples include [4]，[5]，[6] and [7]. They use temporal logic to represent the formal requirement specifications that further serve as the basis for verification and validation.

The most important documentation used in the design phase is design specification. This acts as a blueprint for the actual coding by outlining the logic of individual code modules. It also assists maintenance programmers as they modify the program to add enhancements or fix errors [8]. A design specification is generally described by formal or semi-formal methods, such as hierarchy charts, logic charts, state transition diagrams, state machines, data flow diagrams, data dictionaries, object-oriented approaches, and a great number of formal languages [9]. Some typical formal and semi-formal notations used for design specification include UML [10， 11] and some kinds of architecture description language [12， 13]. Prototype system description language (PSDL) [14, 15] is another typical design specification language for real-time embedded systems. It uses operators and data streams between operators to model the embedded systems and captures timing constraints and control constraints of embedded systems. PSDL also provides a graphic interface to stakeholders. In addition, design specification also serves as the basis for formal analysis as described in [16], [17] and [18] to find design faults early in development.

Configuration is another important aspect of software development that is done based on documentation support, such as architectural specification and component specification. In complex control systems, the configuration of components must be flexible enough to allow rapid online reconfiguration and adaptation to react to environmental changes and unpredictable events at run-time. For this purpose, an open software architecture [19] has been used for integrating control technologies and resources.

Although a lot of effort has been applied toward improving documentation technology [8, 20, 21], there are still open challenges that hinder documentation from providing efficient support for complex systems of embedded systems development. First, according to the traditional concept, software documentation consists only of informal text and diagrams intended for human consumption. This kind of static information simply records some results and process steps during the software development. It cannot capture the dynamic information during the development process. Second, keeping documentation up-to-date is difficult and time consuming. The various representations of documentation increase the complexity of maintaining information consistency, increase the intellectual burden on stakeholders, and introduce the need for transformations that are tedious and error prone when carried out manually. Some formal representations with rigorous logic are conducive to machine manipulation but are difficult for human understanding. Informal representations such as natural language are comfortable for many system stakeholders but are too vague and ambiguous for direct use by computer tools. Although multiple views of the information can alleviate this problem, how to maintain consistency among information presented to both the humans and computer tools is still a challenge. In addition, to guarantee software quality in the end product, the information should be kept consistent among documents of successive development phases. Traditional documentation technologies do not solve this problem.

To attack above problems and enable documentation to provide more effective support for complex SoES development, we proposed a documentation driven development method for SoES [22]. This is a new approach for documentation that can enhance integration of computer aided software development methods, encompass the entire life cycle, support system evolution and improve communication with system stakeholders. In this method, keeping consistency of information transformed between successive development phases is an important issue. It is critical for ensuring high confidence in the end product. For this purpose, a specific method is needed to enable the key information to be consistently transferred between documentation of successive development phases. This paper presents such a specific method.

Much research has been done on attribute grammars that constitute a classic technology for compiling [23-26]. An attribute grammar is a specification of computations and dependence based on a formal calculus introduced by Knuth [27]. Since it is an efficient way to handle the semantics of context-free languages, we plan to extend and exploit it to deal with the information consistency issue identified above. In this paper, we present an attributed object graph model to represent aspects of the “meaning” of document elements and use a set of attribute computation rules to analyze and ensure the consistency of information transformed between successive development phases.

1.3 Organization of This Paper

The rest of paper is organized as follows: Section 2 addresses the core of the documentation driven development method – repository representation; Section 3 presents an attributed object graph model for document elements; Section 4 illustrates the use of attribute computation rules to help ensure consistency of documentation and section 5 presents the conclusion and future work.

2. Repository Representation

The repository representation is the core of the documentation driven development method. All the information related to development process is stored as knowledge in the documentation repository. Each development phase has its own area in the documentation repository. The information is transformed between different documentation areas that belong to successive development phases. Typical examples of the information stored in the repository are requirement specifications, abstracted models, stakeholder input (from sponsors, end users, developers, technical supporters, etc.), design rationale, project management information and the source code. The repository uses a structured central representation for this knowledge so that different stakeholders can communicate with each other based on consistent information and this knowledge can be consistently transformed between successive development phases. Figure 1 illustrates the repository representation.

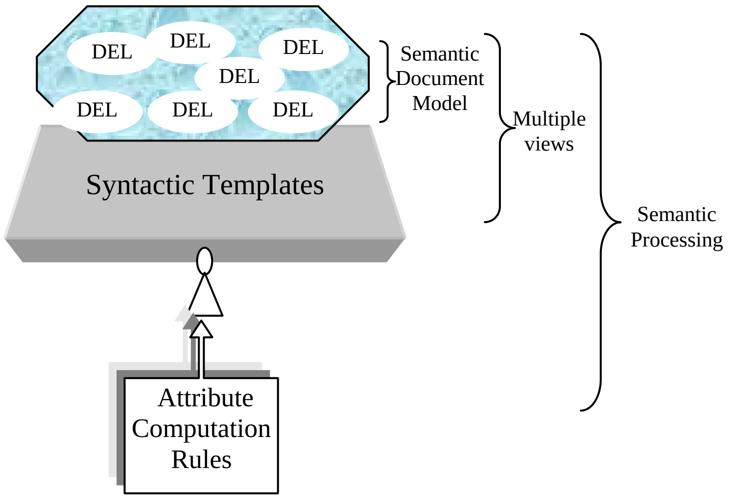


Figure 1 Repository Representation

Figure 1 shows that the repository representation includes three kinds of artifacts, i.e., document elements (DEL), a set of syntactic templates and a set of attribute computation rules. A document element is a basic building block consistent with the semantics of the information contained in the documentation. It is described by a semantic document model. This model is an object model for the information contained in the documentation whose instances form an attributed object graph. The documentation elements are the nodes of this graph. The amount of information associated with each node depends on the degree of formalization for each documentation type. Formal representations have explicit structure at a fine granularity and very simple information associated with individual documentation elements. Informal representations have only a large granularity structure and can have lengthy annotations attached to the nodes. Document elements hold the key information extracted from all the requirements, models, activities and processes related with system development. The model is strongly typed and structured according to a documentation schema. Further development of this approach will need better computer-assisted methods for resolving the ambiguities common in informal representations, transforming them into more formal, finer-grained representations, and for checking the validity of this process.

Syntactic templates are object operations with parameters. The purpose of a syntactic template is to materialize the part of a specific documentation view that corresponds to a given documentation element. The parameters represent the relevant properties of the context and the descendent nodes of the documentation element. Syntactic templates are designed together with specific sets of rules that govern the manipulation of the data stored in the document elements. The content of the document elements is treated as repository knowledge and the different templates govern how that knowledge is used and presented to the stakeholders and tools in the computer development environment. The combination of a document element and different syntactic templates forms the multiple view presentation of the same information. Combining document elements with corresponding templates can also transform the information between representations written in different description languages [22].

Attribute computation rules represent the methods for computing derived document attributes. They make the repository into an active project support system. These rules are organized in a rule base. The rule base is designed to be open in the sense that new rules can be added without changing the effect of any complete subset of the previous version of the rules. This property supports reliable incremental extension of the automation support provided by the repository and enables steady improvement of decision support processes.

In the long term, the repository will perform a variety of automated and computer aided functions such as the following:

- Materialize external representations of documents suitable for particular stakeholders or tools
- Find appropriate subsets and projections of the documents suitable for particular purposes
- Extract computed attributes of documents, such as expected completion date of the project
- Transform data among different representations as needed to support integration of development processes and tools
- Configuration management of the documents [28-30]
- Project management based on management documents such as plans and schedules [31-33]

To address the problem of consistent information transformation between documentation of successive development phases, we describe the attributed object graph model and attribute computation rules in the following sections.

3. Attributed Object Graph Model

This section explains the computational semantics of the attributed object graph model. This is an object model of knowledge in the documentation repository. It has a nested structure with potentially shared nodes, i.e., directed acyclic graph structure. This representation is a generalization of abstract syntax trees and is designed to represent and efficiently analyze constructs that appear in more than one context. This is a common pattern in software artifacts – for example, an operation is typically defined once and called from many different contexts.

In the attributed object graph model, each node represents a semantically meaningful structure, such as an individual requirement, a subsystem, an operation, or an operator within a logical expression. The nodes are the finest grain structures visible to the attribute computation rules. Each node is an instance of an abstract data type. The computed attributes of each node correspond to the operations of the data type. Invoking appropriate methods of the data type can derive the value of an attribute. Attribute computation rules are declarative definitions of these methods.

The semantics of attribute evaluation in the attributed object graph model is a generalization of the corresponding semantics in an ordinary attribute grammar. The two are the same when the graph is a tree. The difference shows up for inherited attributes of shared nodes: in an attribute grammar, each node can have at most one parent, but a shared node in an attributed object graph can have more than one parent.

We require the type of an inherited attribute to be a lattice. In implementation terms, the type must implement the lattice [T] interface with operations

$$\begin{array}{ll} bottom : T & \text{-- least element} \\ lub(T,T) : T & \text{-- least upper bound} \\ le(T,T) : bool & \text{-- approximation ordering} \end{array}$$

and these operations must satisfy the standard properties of a mathematical lattice.

The semantics of an inherited attribute *A* with a defining expression *E* is the least upper bound of the values of *E* in all contexts (i.e. the set of all parent nodes). In implementation terms, an attribute computation rule of the form $child.A = E(parent.A)$ can be realized with an initialization $node.A := bottom$ (for all nodes) and an incremental update step $child.A := lub(child.A, E(parent.A))$ which is enabled in the context of each parent node whenever the value of $parent.A$ changes in that context.

To make the above restriction on attribute types less burdensome, we propose a default extension of all types (a uniform subtype definition) that adds a new constant “bottom” representing an undefined value, another new constant “conflict_error” representing a conflict between two incompatible values inherited from different contexts, and the usual flat ordering on simple data types:

$$\begin{array}{l} le(x,y) = (x = bottom)\ or\ (x = y)\ or\ (y = conflict\_error) \\ lub(x,y)\ = if\ (x = bottom)\ or\ (x = y)\ then\ y \\ \qquad\qquad else\ if\ (y = bottom)\ then\ x \\ \qquad\qquad else\ conflict\_error \qquad \text{-- display an error diagnostic} \end{array}$$

This default can be explicitly overridden by the designer for data types where this makes sense. An example from the domain of timing constraints illustrates the idea:

TYPE DEADLINE EXTENDS INTEGER

$$\begin{array}{l} bottom = MAXIMUM\_INTEGER \\ le(x,y) = x \ge y \\ lub(x,y) = MIN(x,y) \end{array}$$

This corresponds to the idea that if a program meets a given deadline, then it also meets any later deadline. Thus, a component that inherits deadlines of 100ms, 75ms, and 120ms from three different requirement documents is subject to a design constraint to execute within 75ms (since $lub(lub(100,75),120) = 75$ for the deadline type defined above).

To ensure the high confidence of SoES, it is important to keep timing properties consistent during the whole development processes. This means the information related to timing properties needs to be consistently identified in documents belonging to different development phases. In the next section, we will use timing properties as an example to illustrate the application of the proposed attributed object graph model to the problem of maintaining document consistency.

4. Attribute Computations for Document Management

The attributed object graph model was designed to realize documentation checks and transformations that support high confidence SoES development. These computations are used to (a) calculate the attributes from the information in the documentation repository, (b) transform the information from one development phase to another, (c) analyze the consistency between the information transformed between development phases, description languages and information views, and (d) extract subsets of documents needed for particular purposes. The declarations of these computations form a set of attribute computation rules.

In the development process, the documentation generated in early development phases is taken as input for the next phases and guides the development activities in that phase to generate the output documentation. To ensure the quality of the end product, it is important to keep selected non-functional properties needed for high confidence visible and consistent during the whole development process. These high-confidence properties should be kept consistent between the documentation generated in the early phase and that generated in the next phase. Although the format of this kind of “key information” may be different between two development phases, this information of later phase should imply that of the earlier. For example, in the requirement phase, requirement documentation may include information describing a customer request for deriving the computation result within containing constraints, then in the design phase, the design documentation should include information with the same implication, such as information related to the deadline, period and maximum execution time.

In this paper, we use timing properties transformation between requirement phase and design phase as the example to illustrate the application of attribute computation rules. Suppose that the requirements specification includes a maximum response time (MRT) constraint for a given service S and that at the architectural level, S is realized by a software component C. The maximum response time appears at the requirements level because it is directly visible to the system stakeholders and is of vital concern to them, since late control signals can have catastrophic consequences.

At the design level, this constraint is transformed into lower level constraints on the period and maximum execution time (MET) of a periodic software process. If the documentation element S in the requirements document is a parent node of the documentation element C in the design document, the design rule that ensures consistency of the two documents with respect to this issue can be expressed by the following simple attribute computation rules: (MRT is an attribute of S; timing_check, period, MET and diagnostic are four attributes of C.)

$$C.timing\_check = (C.period + C.MET \le S.MRT)$$

$$C.diagnostic = Unless\ (\ C.timing\_check,\ error\_message)$$

-- Unless (*C*, *M*) displays *M* if *C* = *false* and does nothing otherwise

The rationale for this rule is that the worst case occurs when a request arrives just after the request stream has been polled. In this case, the transaction will start processing one period later, and the software can take up to the maximum execution time after the transaction starts to produce the result. This simplified example assumes that all processing is done locally, so that we do not have to account for any latency in the communications link between the machine running the component C and the machine running the consumer process waiting for the output of C.

A mature documentation repository will actively check many different generic design rules like the one illustrated in this simple example. The rule base will gradually grow as processes are improved and constraints related to high confidence attributes are gradually formalized.

5. Conclusions and Future Work

In recent years, complex embedded systems, known as systems of embedded systems (SoES), have been widely used in many fields such as flight control and avionics, industrial process control, weapon system control and nuclear plant control. The high complexity of SoES forces them to confront many software development challenges, such as difficulty ensuring software quality, difficulty supporting software evolution and difficulty supporting communication among different stakeholders. Much research on individual embedded system development has demonstrated that documentation plays an important role in development process and provides a promising way to cope with these challenges. In our research, we are developing a documentation driven development method for SoES. This is a new approach to documentation that can enhance integration of computer aided software development methods, encompass the entire life cycle, support system evolution and improve communication with system stakeholders. This effort enables documentation to provide more effective support for complex SoES development.

Furthermore, keeping information transformation consistent between successive development phases is an important issue in the proposed approach. It is critical for ensuring high confidence in the end product. In this paper, we investigate a specific method to perform information consistency checking in documentation driven development of SoES. We present an attributed object graph model to describe the semantics of document elements. Based on this model, we show how attribute computation rules can be used to analyze consistency between the key information such as timing properties transformed from one development phase to another.

However, further work still needs to be done in order to improve capability of documentation to efficiently support complex embedded system development. For example, a better language for defining attribute computations and an optimized evaluation engine that can handle the generalized attribute semantics proposed here should be designed.

References

1. M. Maier, "Architecting Principles for Systems-of-System", *Technical Report*, *http://www.infoed.com/Open/PAPERS/systems.htm*.
2. http://www.nist.gov/itl/div897/pubs/fips105.pdf
3. L. Goldin, D. Berry, "AbstFinder: A Prototype Abstraction Finder for Natural Language Text for Use in Requirement Elicitation", *Automated Software Engineering*, No.4, 1997, pp.375-412.
4. E. Clarke, E. Emerson and A. Sistla, "Automatic Verification of finite state concurrent systems using temporal logic specification", *http://citeseer.nj.nec.com/clarke93verification.html*.
5. M. Dwyer, J. Hatcliff, and G. Avrunin, "Software Model Checking for Embedded Systems", *www.cis.ksu.edu/~dwyer/projects/HCES-May-01-1.ppt.*
6. D. Garlan, "Model Checking Publish-Subscribe Software Architectures", *Presentation at ARO Kickoff Meeting*, University of Pennsylvania, Philadelphia, PA, May 24 - 25, 2001, www.cs.cmu.edu/~svc/talks/ppt/garlan.ppt
7. J. Wing, "Scenario Graph Generation and MDP-Based Analysis", *Presentation at ARO Kickoff Meeting*, University of Pennsylvania, Philadelphia, PA, May 24 - 25, 2001, http://www-2.cs.cmu.edu/~svc/talks/html/wing_files/frame.htm.
8. J. French, J. Knight and A. Powell, "Applying Hypertext Structures to Software Documentation", *www.cs.virginia.edu/~cyberia/papers/IPM97.pdf.*
9. http://www.comlab.ox.ac.uk/archive/formal-methods/
10. G. Booch, J. Rumbaugh and I. Jacobson, "The Unified Modeling Language user guide", Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, 1999.
11. Object Modeling Group, Inc., "Unified Modeling Language Specification, version 1.3", June 1999.
12. M. Kande, V. Crettaz, A. Strohmeier and S. Sendall, "Bridging the Gap between IEEE 1471, Architecture Description Languages and UML", *http://icwww.epfl.ch/publications/documents*
13. N. Mehta, N. Medvidovic, "Towards a Taxonomy of software Connectors", *Proceedings of 22th International Conference on Software Engineering*, Limerick Ireland, 2000, sunset.usc.edu/classes/cs599_2000/Conn-ICSE2000.pdf.

14. Luqi, V. Berzins and R. Yeh, “A Prototyping Language for Real-Time Software”, *IEEE Transaction on Software Engineering*, Vol.14, No.10, 1988, pp.1409-1423.
15. Luqi, R. Steigerwald, G. Hughes and V. Berzins, “CAPS as a Requirement Engineering Tool”. *Proceedings of Tri-Ada'91 International Conference*, San Jose, USA, Oct 22-25, 1991, pp. 75-83.
16. R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivancic, V. Kumar, I. Lee, *et al*., “Hierarchical Modeling and Analysis of Embedded Systems”, *Proceedings of IEEE*, Vol. 91, No 1, January, 2003, pp. 11-28.
17. R. Alur, “Model-based Design of Embedded Software”, *Presentation at Vanderbilt Workshop*, Vanderbilt University, Nashville, TN, December 13-14, 2001, www.hpcc.gov/iwg/sdp/vanderbilt/agenda_presentations/alur.pdf.
18. O. Sokolsky, A. Philippou, I. Lee and K. Christou, “Modeling and Analysis of Power-Aware Systems”, *Proceedings of 9th International Conference on Tools and Algorithms for Construction and Analysis Systems* (*TACAS03)*, Warsaw, Poland, April 7-11, 2003, pp.409-425.
19. L. Wills, S. Sander, S. Kannan, A. Kahn, J. Prasad, and D. Schrage, “An Open Control Platform for Reconfigurable, Distributed, Hierarchical Control Systems”, *Proceedings of the Digital Avionics Systems Conference*, Philadelphia, PA, October, 2000, http:// controls.ae.gatech.edu/papers/kannan_dasc_00.pdf.
20. P. Devanbu, P. Selfridge, R. Branchman and B. Ballard, “LaSSIE: a Knowledge-based Software Information System”, *Proceedings of the IEEE 12th International Conference on software Engineering*, 1990, pp.249-261.
21. C. Paris, K. Linden, “Building Knowledge Bases for the Generation of Software Documentation”, *http://acl.ldc.upenn.edu/C/C96/C96-2124.pdf.*
22. Luqi, L. Zhang, “ Documentation Driven Agile Development for Systems of Embedded Systems”, *Submitted to Monterey Workshop 2003*.
23. G. Hedin, “Reference Attributed Grammars”, *Proceedings of Second workshop on Attribute Grammars and their Applications (WAGA99)*, March 1999, pp. 158-172, http://www-rocq.inria.fr/oscar/www/fnc2/WAGA99/proceedings/hedin/hedin2.pdf
24. D. Parigot, G. Roussel, E. Duris and M. Jourdan, “Attribute Grammars: a Declarative Functional Language”, *http://www.inria.fr/rrrt/rr-2662.html*, 1995.
25. R. Herndon, V. Berzins, “ The Realizable Benefits of a Language Prototyping Language”, *IEEE Transaction on Software Engineering*, Vol. 14, No. 6, June 1988, pp. 803-809.
26. U. Kastens, “Modularity and reusability in Attribute Grammars”, *http://citeseer.nj.nec.com/kastens92modularity.html*, 1992.
27. D. Knuth, “Semantics of Context-free Language”, *Journal of Mathematical System Theory*, Vol. 2, No. 2, June, 1968, pp.127-145.
28. O. Ibrahim, “A Model and Decision Support Mechanism for Software Requirement Engineering”, *Naval Postgraduate School, Ph.D. Dissertation*, September 1996.
29. M. Harn, V. Berzins and Luqi, "A Dependency Computing Model for Software Evolution", *Proceedings of the 11th International Conference on Software Engineering and Knowledge Engineering*, June 17-19, 1999, Kaiserslautern, Germany.
30. M. Harn, V. Berzins and Luqi, "Software Evolution Process via a Relational Hypergraph Model", *Proceedings of IEEE/IEEJ/JSAI International Conference on Intelligent Transportation Systems*, Tokyo, Japan, October 5-8, 1999.
31. S. Badr, V. Berzins, "A Software Evolution Control Model", *Proceedings of Monterey Workshop 94, Monterey,* CA, September 7-9, 1994, pp. 160-171.
32. S. Badr, “ A Model and Algorithms for A Software Evolution Control System”, *Naval Postgraduate School, Ph.D. Dissertation*, December 1993.
33. M. Harn, “Computer Aided Software Evolution based on Inferred Dependencies”, *Naval Postgraduate School, Ph.D. Dissertation*, December 1999.

On Verification Modelling of Embedded Systems

Ed Brinksma and Angelika Mader [*]

Department of Computer Science, University of Twente
PO Box 217, 7500 AE Enschede, Netherlands
{brinksma,mader}@cs.utwente.nl

Abstract. Computer-aided verification of embedded systems hinges on the availability of good verification models of the systems at hand. Because of the combinatorial complexities that are inherent in any process of verification, such models generally are only abstractions of the full design model or system specification. As they must both be small enough to be effectively verifiable and preserve the properties under verification, the development of verification models usually requires the experience, intuition and creativity of an expert. We argue that there is a great need for systematic methods for the construction of verification models to move on, and leave the current stage that can be characterised as that of "model hacking". The ad-hoc construction of verification models obscures the relationship between models and the systems that they represent, and undermines the reliability and relevance of the verification results that are obtained. We propose some ingredients for a solution to this problem.

1 What is the problem?

Many embedded systems are subject to critical applications, massive replication and/or widespread distribution. The quality of their design, therefore, is an issue with major societal, industrial, and economic implications. Systematic methods to design and analyse such systems are consequently of great importance. In this paper we focus on a number of methodological aspects of the analysis of embedded systems.

In computer science formal methods research much progress is being made in tool-supported, model-based system analysis, in particular in the areas of model checking, theorem proving and testing. A prerequisite for such analysis is the availability of a formal model of the system. For pure software systems the (semi-)automated extraction of abstract models out of (a specification of) the source code is a possibility in principle. For embedded systems, however, the modelling task is intrinsically more complicated. As interaction with their (physical) environment is an essential ingredient, good models must integrate the relevant aspects of the system hardware, software, and environment.

[*] This work has been partly funded as part of the IST AMETIST project funded by the European Commission (both authors), and the MOMS project funded by the Netherlands Organisation for Scientific Research (second author).

The current modelling and verification practice for embedded systems can be characterised by the slogan: "*model hacking precedes model checking*". Constructing a model is typically based on the experience, intuition and creativity of an expert. In an initial phase the model is improved in trial and error fashion: first verification runs show errors in the model, rather than errors in the system. Once the model is considered stable and correct, the subsequent verification runs are considered analyses of the system.

Processes of model construction are mainly described in case studies, but in most cases the applied design decisions and paradigms remain implicit. As a consequence, it is difficult to compare, and assess the quality of the models. Therefore, different analysis results are also difficult to interpret. The ad-hoc construction of verification models obscures the relationship between models and the systems that they represent, and undermines the reliability and relevance of the verification results that are obtained. Moreover, most of the existing case studies are done to show that a given algorithm is faster, or that some tool is applicable to a given problem.

So far, the *method* of modelling has not been a topic of systematic research. Quoting the NASA Guidebook on formal methods [17]: "The observation [that there is a paucity of *method* in formal methods] focuses in particular on the absence of 'defined, ordered steps' and 'guidance' in applying ... methodical elements that have been identified." A first collection of relevant empirical data exists in the form of various case studies consisting of different teams applying different modelling approaches to a common problem, such as the steam boiler case study [14], the RCP-memory case study [8], the VHS batch plant case study [16], and an industrial distributed data base application [9]. What is mostly missing, however, is the extraction of the commonalities and differences of the resulting models, and their comparative evaluation against qualitative criteria such as ease of modelling, quality of analysis, tool support, adaptability, maintainability, etc.

A wealth of material exists on the topic of specification (formalisms) and their application, but this is essentially aimed at the construction of complete models (specifications) of system behaviour, as unambiguous statements of the desired functionality, where the resulting size and complexity are of secondary interest. Our interest, however, is in models for selected properties of interest where simplicity and size are of prime concern to control the combinatorial explosion that results from their analysis. Only such an approach offers hopes for tool-supported analysis.

2 What do we need?

The previous section indicated that we need methods for the construction of verfication models that combine the following properties:

- They are of *limited complexity*, meaning that they can be analysed with the help of computer-aided verification tools;

- They are *faithful*, meaning that they capture accurately the (verification) properties of interest;
- They are *traceable*, meaning that they have a clear and well-documented relation to the actual (physical) system.

It is clear that it is not easy, generally, to satisfy all of these constraints concurrently. The construction of verification models really is a design problem in its own right that may share many characteristics of the design problem of the system itself, but is not identical to it because of its different purpose and complexity constraints. Being a design problem, it generally also involves a strongly creative element that defies automization.

Below, we discuss the above points in some more detail.

2.1 Limited complexity

Modern formal methods research is tightly coupled to the development of analytical software tools such as animators, model checkers, theorem provers, test generators, simulators, Markov chain solvers, etc. Only computer-aided analysis can hope to overcome the combinatorial complexities that are inherent in the study of real systems.

Even then, one of the main obstacles to overcome is the infamous combinatorial explosion problem, causing the size of search spaces, state spaces, proof trees, etc. to grow exponentially in the number of (concurrent) system components. This makes it essential to keep the complexity of verfication models within the range that can be effectively handled by tools.

Of course, the effective limit is growing rapidly itself, due to both exponential growth of hardware speed and memory miniaturisation (Moore's law), and improvements the algorithms and data structures used in the tools. Still, it seems reasonable to assume that effective computer-aided analysis must always be based on models that simplify or abstract from the full functionality of the system at hand.

2.2 Faithfulness

The purpose of verification is to show that a system satisfies (or does not satisfy) various properties. It is obvious that the model we investigate for verification should share the properties of interest with the original system. Therefore, the abstraction steps that are applied in the verification modelling process should preserve the relevant properties. Under most circumstances this is a tall order for two reasons:

1. *We may not know the relevant formal properties*. In fact, finding the right formal properties to verify is often as much a part of the verification design problem as finding the right model. In practice, the design of the properties

and model go hand in hand. Together with the necessity to keep models small this leads to a collection of different models, each geared for a different (set of) formal property (properties) [5].

2. *We may not know whether our abstractions preserve them.* Showing that our abstractions preserve the intended properties may be as hard as our original verification problem, and the set of abstractions with known preservation properties is usually too small to suffice for practical problems.

A common way out of the second complication is to be satisfied with approximating models that may generate *false negatives* or *false positives*, i.e. report nonexistent errors, or obscure existing errors, respectively. In the first case, spurious errors may be filtered out if they cannot be traced back to the original system, e.g. by counterexample trace analysis in model checking. In the second case, one is reduced to *falsification* or *debugging*, where the presence of errors can be shown, but not their absence.

The insight that all verification is in some sense reduced to debugging, due to the untimately informal relationship between model and real system, is misleading. Although it is true in the Popperian sense of falsification, it should not be interpreted as a reason not to seek the positive identification of model properties. This minimalistic approach is not enough when a model must have certain properties to be relevant at all for the verification task at hand. For example, when deriving schedules by means of model checking [6], one needs to know positively that the relevant aspects of time are represented in the model. The modelling process should therefore guarantee in some way the presence of the properties of interest.

An interesting development in software model checking is that of (semi-)automated model abstraction from code [3, 2, 11], allowing for many approximating verification models to be analysed concurrently. This way of debugging can under circumstances achieve a good error coverage. For embedded systems, however, the modelling task is intrinsically more complicated. As interaction with a (physical) environment is an essential ingredient, good models must integrate the relevant aspects of the system hardware, software, and environment. This cannot be achieved by automated code abstraction. Libraries of succesful modelling fragments coupled to specific system domains may provide a way forward here.

2.3 Traceability

Verifying an erroneous model is both useless and time-consuming. As pointed out above already obtaining faithful models is not an easy task, and cannot be achieved by formal means only. In fact, one of the most important ways of establishing the relevance of a (verification) model is by keeping track of the design and abstraction steps that relate it to the actual system, the choices that were made, and the reasons behind them.

There is not a unique starting point for model derivation. In an a posteriori verification case one could start from a piece of embedded software together with an engineer's diagram of what the physical part of the embedded system does. It could be a standard or another informal description. In an a prioriverification we might start from a desired behaviour specification. In any case we have to get from a system description that is likely not to be a formal object to a formal model. The preservation of relevant properties is therefore (in many design steps) also not a formal notion. In such cases the only form of evidence that can be given is by insight. Insight requires a transparent representation of the design steps taken. Another advantage of a clear derivation traject is that is easier to check (by others). Transparency makes it easier to detect errors in the modelling process.

A further relevant aspect in this context is the comparability of verification approaches and results. In many research projects different groups work with different tools on the same case studies. The comparison of the results, however, is difficult due to the fact that there are no criteria available to compare the different models. Their design track records provide valuable information that can help to compare and relate them.

3 How do we get there?

Anyone attempting to obtain methods for transparent model construction for embedded system verification along the lines sketched above, is immediately confronted with two facts:

– *domain dependency*: design methods and concepts depend strongly on particularities of the application domain at hand. Moreover, different domains, specifications, system descriptions, and creative elements provide a huge range of problem settings.
– *subjectivity*: transparency is a relative and imprecise notion. What constitutes a clear documentation of a modelling step depends very much on the kind of the step, the problem domain, the level of informality and formality, and certainly also on personal taste. As clear as possible always means to restrict to the information necessary and find a good representation for it, which could be a diagram, an informal explanation, a table a formal mapping, etc. However, diagrams with ambiguous semantics can be as difficult to understand as some formalism that requires too much details.

Under these circumstances, what is needed is a *protocol* for the construction of verification models. It must adopted by the (embedded systems) verification community at large, so that it can be a point of reference for the construction, evaluation, and comparison of verification models. At the same time, it can provide the basis for the collection and organisation of succesful models and modelling patterns as ingredients for a true discipline of verification model engineering.

Even in the inherent absence of a universal approach to model construction, each modelling process should be *guided* by a number of basic considerations. The guidance is obtained by systematically considering the different aspects:

– *scope of modelling*;
– *properties of interest*;
– *modelling patterns*;
– *effective computer-aided analysis*.

We digress shortly on each of these points.

scope of modelling

It is necessary to make explicit what the model includes and what not. What part of the system, assumptions about the environment, operating conditions, etc. The system domain under consideration is also relevant here, as it usually introduces particular concepts an structuring principles that must be referred to. It has turned out te be useful to make *dictionaries* of the domain specific vocabulary used: it helps to agree, e.g., with system experts, on the interpretation of the description, to filter the relevant notions, and to have instrumentation for unambiguous explanations in later modelling steps.

properties of interest

The properties of interest should be stated as explicitly as possible. As indicated earlier, initially we may not know what will be the correct formalisation of these properties and their preservation principles. Nevertheless, their informal statement and intention should be stated as completely and unambiguously as possible. At each modelling step the best possible argumentation should be put forward on why they can be assumed to be preserved. The definitions and arguments should become more precise as formalisation progresses in the course of the model construction.

modelling patterns

Important in any more systematic approach is the identification of characterising structural elements that occur in a system under verification, and allow the sharing of modelling patterns in models that have such elements in common. The concept of solution patterns is central to many branches of engineering; an excellent elaboration of the idea of design patterns in software engineering by Micheal Jackson as *problem frames* can be found in [13].

Characterising structural elements can be representative for a particular application area (e.g. network controllers, chemical process control), but not necessarily so. Very different systems, such as e.g. a steel plant [10] and a multimedia juke box [15] have been found to share significant similarities, viz. on an abstract level both are comparable transport scheduling problems with a resource bottleneck.

Some examples of characterising structural elements that play a role in in many embedded systems are controller polling policies, scheduling mechanisms, (programming) languages, operating system and network features such as interrupt handling, timers, concurrency control, communication architecture, time granularity, etc. at various levels of abstraction depending on the properties of interest.

effective computer-aided analysis.

Modelling and analysis of realistic systems is impossible without proper tool support. Therefore, it is reasonable to focus on model classes for which efficient tool support is available. As analysis tools are a very active area of research where continuous progress is being made, it is important to be open to new developments. Tool performance is improving dramatically by such devices as better data structures, sophisticated programming techniques, and parallelisation [4] of algorithms. New, more powerful approaches such as guided [7, 1] and parametric model-checking [12, 18] are extending the applicability of tools significantly.

4 Conclusion

We have argued that the construction of verification models for embedded system is a non-trivial task for which there are no easy solutions. It is our feeling that the issues that we have raised deserve more more attention from both the community of researchers and the industrial appliers of formal methods. It is of great interest that the designs of verification models are made available to be able to evaluate, compare, improve, collect and use them in a more systematic way, and leave the current stage of model hacking. To be able to do so a generally agreed protocol for their documentation as transparent and guided design processes is much needed. We have outlined some ingredients for such a protocol.

References

1. R. Alur, S. La Torre, and G. Pappas. Optimal paths in weighted timed automata. In *Proc. of the Fourth International Workshop on Hybrid Systems: Computation and Control*, volume 2034 of *LNCS*. Springer, 2001.
2. T. Ball, S.K. RajamaniJ. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubacj, and H. Zeng. Bandera: Extracting finite-state models from java source code. In *Proceeding 22nd ICSE*, pages 439–448. IEEE Computer Society, 2000.
3. T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proceeding 29th POPL*, pages 1–3, 2002.
4. G. Behrmann, T. Hune, and F. W. Vaandrager. Distributed timed model checking - how the search order matters. In *Proceedings CAV'2000*, volume 1855 of *LNCS*. Springer, 2000.

5. E. Brinksma. Verification is experimentation! *Journal of Software Tools for Technology Transfer (STTT)*, 3(2):107–111, 2001.
6. E. Brinksma, A. Mader, and A. Fehnker. Verification and optimization of a PLC control schedule. *International Journal on Software Tools for Technology Transfer*, 4(1):21–53, 2002.
7. K. G. Larsen et al. As cheap as possible: Efficient cost-optimal reachability for priced timed automata. In *Proceedings of CAV'2001*, volume 2102 of *LNCS*. Springer, 2001.
8. M. Broy et al., editor. *Formal System Specifications - The RCP-Memory Specification Case Study*, volume 1169 of *LNCS*. Springer, 1996.
9. P. H. Hartel et al. Questions and answers about ten formal methods. In *Proceedings of the 4th Int. Workshop on Formal Methods for Industrial Critical Systems*, volume II, pages 179 – 203. ERCIM/CNR, 1999.
10. Ansgar Fehnker. Scheduling a Steel Plant with Timed Automata. In *Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*. IEEE Computer Society Press, 1999.
11. G. Holzmann and M.H. Smith. Software model checking. In *Proceeding FORTE 1999*, pages 481–497. Kluwer, 1999.
12. T. S. Hune, J. M. T. Romijn, M. I. A. Stoelinga, and F. W. Vaandrager. Linear parametric model checking of timed automata. In *Proceedings of TACAS'2001*, volume 2031 of *LNCS*, pages 189–203. Springer, 2001.
13. M. Jackson. *Problem Frames*. ACM Press, Addison-Wesley, 2001.
14. H. Langmaack, E. Boerger, and J. R. Abrial, editors. *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, volume 1165 of *LNCS*. Springer, 1996.
15. M. Lijding, P. Jansen, and S. Mullender. Scheduling in hierarchical multimedia archives. Submitted.
16. A. Mader, E. Brinksma, H. Wupper, and N. Bauer. Design of a PLC control program for a batch plant - VHS case study 1. *European Journal of Control*, 7(4):416–439, 2001.
17. NASA's Software Program. *Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems*. eis.jpl.nasa.gov/quality/Formal_methods/.
18. R. L. Spelberg, R. de Rooij, and H. Toetenel. Experiments with parametric verification of real-time systems. In *Proceedings of the 11th Euromicro Conference on Real Time Systems*, 1999.

From Natural Language Requirements to Executable Models of Software Components[*]

Barrett R. Bryant, Beum-Seuk Lee, Fei Cao, Wei Zhao, Carol C. Burt
Department of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, AL 35294-1170, U. S. A.
{bryant, leebs, caof, zhaow, cburt }@cis.uab.edu

Rajeev R. Raje, Andrew M. Olson
Department of Computer and Information Science
Indiana University-Purdue University-Indianapolis
Indianapolis, IN 46202, U. S. A.
{rraje, aolson}@cs.iupui.edu

Mikhail Auguston
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943, U. S. A.
auguston@cs.nps.navy.mil

Abstract

The UniFrame approach to component-based software development assumes that concrete components are developed from a meta-model, called the Unified Meta-component Model, according to standardized business domain models. Implicit in this development is that there is a Platform Independent Model (PIM) which is transformed into a Platform Specific Model (PSM) under the principles of Model-Driven Architecture. This paper advocates natural language as the starting point for developing the business domain models and the meta-model and shows how this natural language may be mapped through the PIM to PSM using a formal system of rules expressed in Two-Level Grammar. This allows software requirements to be progressed from business logic to implementation of components and provides sufficient automation that components may be modified at the model level, or even the natural language requirements level, as opposed to the code level.

1. Introduction

Model-driven architecture (MDA) [Fran03] is an approach whereby software components are expressed using models, typically in UML[1]. The basic approach is to define Platform Independent Models (PIMs) which express the business logic of components conforming to some domain (e.g. banking, telecommunications, etc.) and then to derive Platform Specific Models (PSMs) using a specific component technology (e.g. CORBA[2], J2EE[3], etc.). Business logic is typically expressed

[*] This material is based upon work supported by, or in part by, the U. S. Army Research Laboratory and the U. S. Army Research Office under contract/grant number DAAD19-00-1-0350, and by the U. S. Office of Naval Research under award number N00014-01-1-0746.

[1] UML – Unified Modeling Language, http://www.omg.org/uml
[2] CORBA – Common Object Request Broker Architecture, http://www.corba.org
[3] J2EE – Java 2 Enterprise Edition, http://java.sun.com/j2ee

in natural language before a model is developed. Standardization of business domains and associated components is being undertaken by the Object Management Group (OMG)[4]. To facilitate the MDA approach to be used in practice, automated tools are needed to develop the business domain specifications from their requirements in natural language as well as to enable transformation from PIMs into PSMs. Furthermore, if MDA is to be used for constructing distributed software systems, then the models must consider not only functional aspects of business logic, but also non-functional aspects, which we call Quality-of-Service (QoS). QoS attributes are not currently considered in the MDA framework.

UniFrame [Raje01] is an approach for assembling heterogeneous distributed components, developed according to MDA principles, into a distributed software system with strict QoS requirements. Components are deployed on a network with an associated requirements specification, expressed as a Unified Meta-component Model (UMM) [Raje00] in the Two-Level Grammar (TLG) specification language [Brya02a]. The UMM is integrated with generative domain models and generative rules for system assembly [Czar00] which may be automatically translated into an implementation which realizes an integration of components via generation of glue and wrapper code. Furthermore, the glue/wrapper code is instrumented to enable validation of the QoS requirements [Raje02].

This paper describes a unified method of expressing business domain models in natural language, translating these into associated business logic rules for that domain, application of the business logic rules in building MDA PIMs, and maintaining these rules through development of PSMs. The complete mapping takes place using a formal system of rules expressed in Two-Level Grammar. This allows software requirements to be progressed from business logic to implementation of components and provides sufficient automation that components may be modified at the model level, or even the natural language requirements level, as opposed to the code level. Section 2 describes our previous work with Two-Level Grammar and its use as a specification language. The application of this to Model-Driven Architecture is discussed in section 3. Finally we conclude in section 4.

2. From Natural Language Requirements to Formal Models

To achieve the conversion from requirements documents to formal models several levels of conversions are required. First the original requirements written in natural language are refined as a preprocessing of the actual conversion. This refinement task involves checking spellings, grammatical errors, consistent use of vocabularies, organizing the sentences into the appropriate sections, etc. The requirements are expected to be organized in a well-structured way, e.g. as laid out in [Wils99] or as a collection of use-cases [Jaco99], and be part of an ontological domain [Lee02b]. Since we are allowing for specification of components that will be deployed in a distributed environment, Quality-of-Service attributes are also specified [Yang02]. Next the refined requirements document is expressed in XML format. By using XML to specify the requirements, XML attributes (meta-data) can be added to the requirements to interpret the role of each group of the sentences during the conversion. The information of the domain-specific knowledge is specified in XML. The domain-specific knowledge describes the relationship between components and other constraints that are presumed in requirements documents or too implicit to be extracted directly from the original documents [Lee02a].

Then a knowledge base is built from the requirements document in XML using natural language processing (NLP) [Jura00] to parse the documentation and to store the syntax,

[4] http://www.omg.org

semantics, and pragmatics information. In this phase, the ambiguity is detected and resolved, if possible. Once the knowledge base is constructed, its content can be queried in NL. Next the knowledge base is converted, with the information of the domain specific knowledge, into Two Level Grammar by removing the contextual dependency in the knowledge base [Lee02c]. TLG is used as an intermediate representation to build a bridge between the informal knowledge base and the formal specification language representation. The name “two-level” in Two-Level Grammar comes from the fact that TLG consists of two context-free grammars interacting in a manner such that their combined computing power is equivalent to that of a Turing machine. Our work has refined this notion into a set of domain definitions and the set of function definitions operating on those domains. In order to support object-orientation, TLG domain declarations and associated functions may be structured into a class hierarchy supporting multiple inheritance.

Finally the TLG code is translated into VDM++, an object-oriented extension of the Vienna Development Method [Durr92], by data and function mappings. VDM++ is chosen as the target specification language because VDM++ has many similarities in structure to TLG and also has a good collection of tools for analysis and code generation. Once the VDM++ representation of the specification is acquired we can do prototyping of the specification using the VDM++ interpreter. Also we can convert this into a high level language such as Java or C++ or into a Rational Rose model in UML [Quat00] using the VDM++ Toolkit [IFAD00]. Using XMI[5] format, not only the class framework but also its detailed functionalities can be specified and translated into OCL (Object Constraint Language) [Warm99]. The structure of the system is shown in Figure 1.

3. Integration with Model-Driven Architecture

The method of translating requirements in natural language into UML models and/or executable code described in the previous section may be used to translate business logic into formal rules. Business domain experts from various application domains may express their specification in natural language and then our system translates this into Two-Level Grammar rules via natural language processing (NLP). These rules are encapsulated in a TLG class hierarchy defining a knowledge base with domain ontology, domain feature models (specifying the commonality and variability among the product instances in that domain), feature configuration constraints, feature interdependencies, business operational rules, temporal concerns, etc. TLG specifies the complete feature model including the structural syntax and various kinds of semantic concerns [Zhao03]. For example, assume that our application domain is banking. The business domain will then include a feature model of a bank, which includes specification of the various attributes and operations a bank will have, such as account creation and management, deposit, withdraw and balance checking operations on individual accounts, etc. In related work [Cao03a], we have investigated the construction of Generative Domain Models [Czar00] using the Generic Modeling Environment [GME01]. This tool may also be extended with a natural language processor as a front end, i.e., by applying natural language processing to the business domain model (which is represented in natural language), which can then extract feature model representation rules and then interpret those rules to generate a graphical feature diagram.

Platform Independent Models in MDA are based upon the business domains and associated logic for the given application. TLG allows these relationships to be expressed via inheritance. If a software engineer wants to design a server component to be used in bank account management systems, then he/she should write a natural language requirements specification in the form of a UMM (Unified Meta-component Model) describing the characteristics of that component. Our

[5] XMI - XML Metadata Interchange, http://www.omg.org/technology/documents/formal/xmi.htm

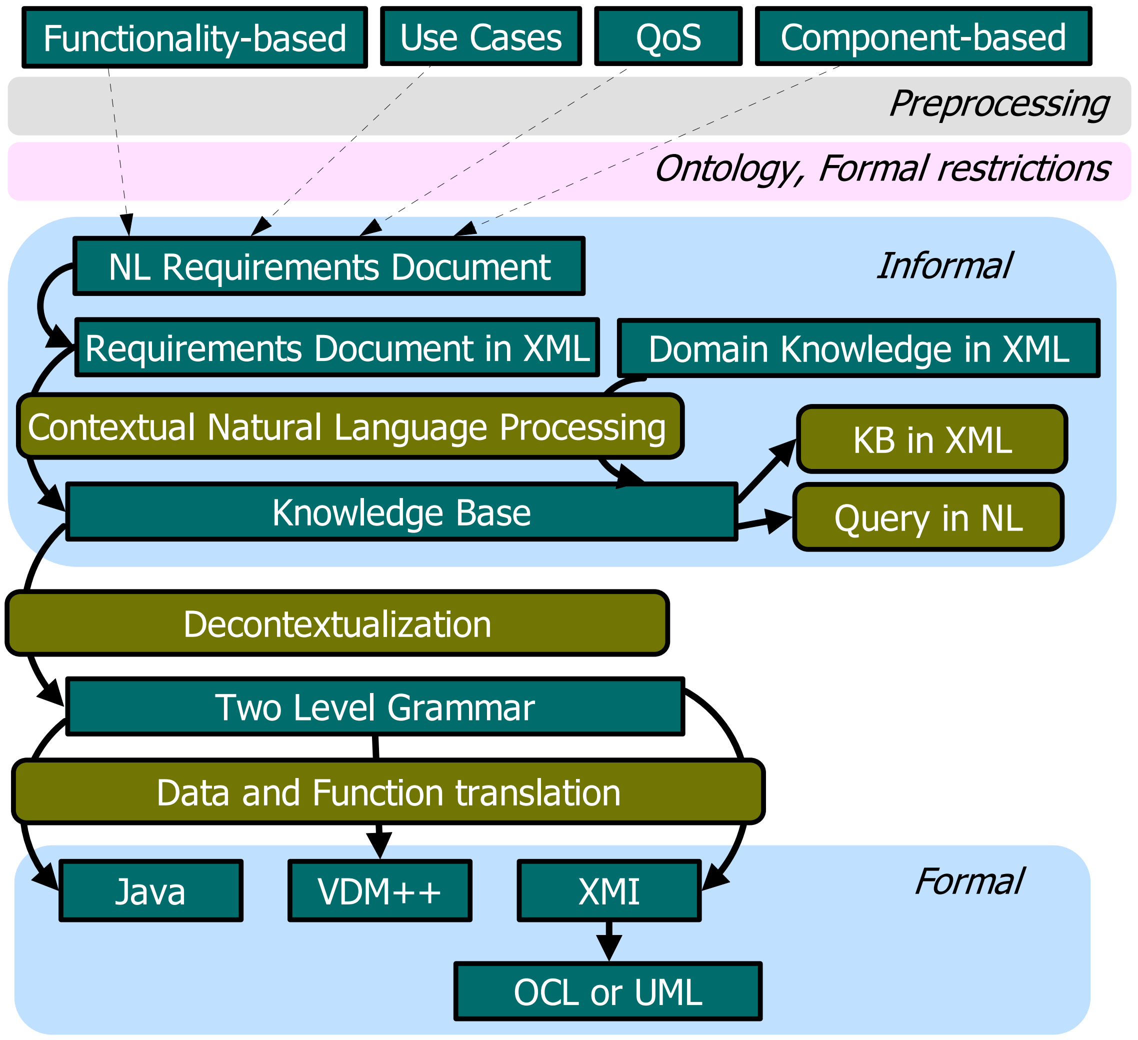
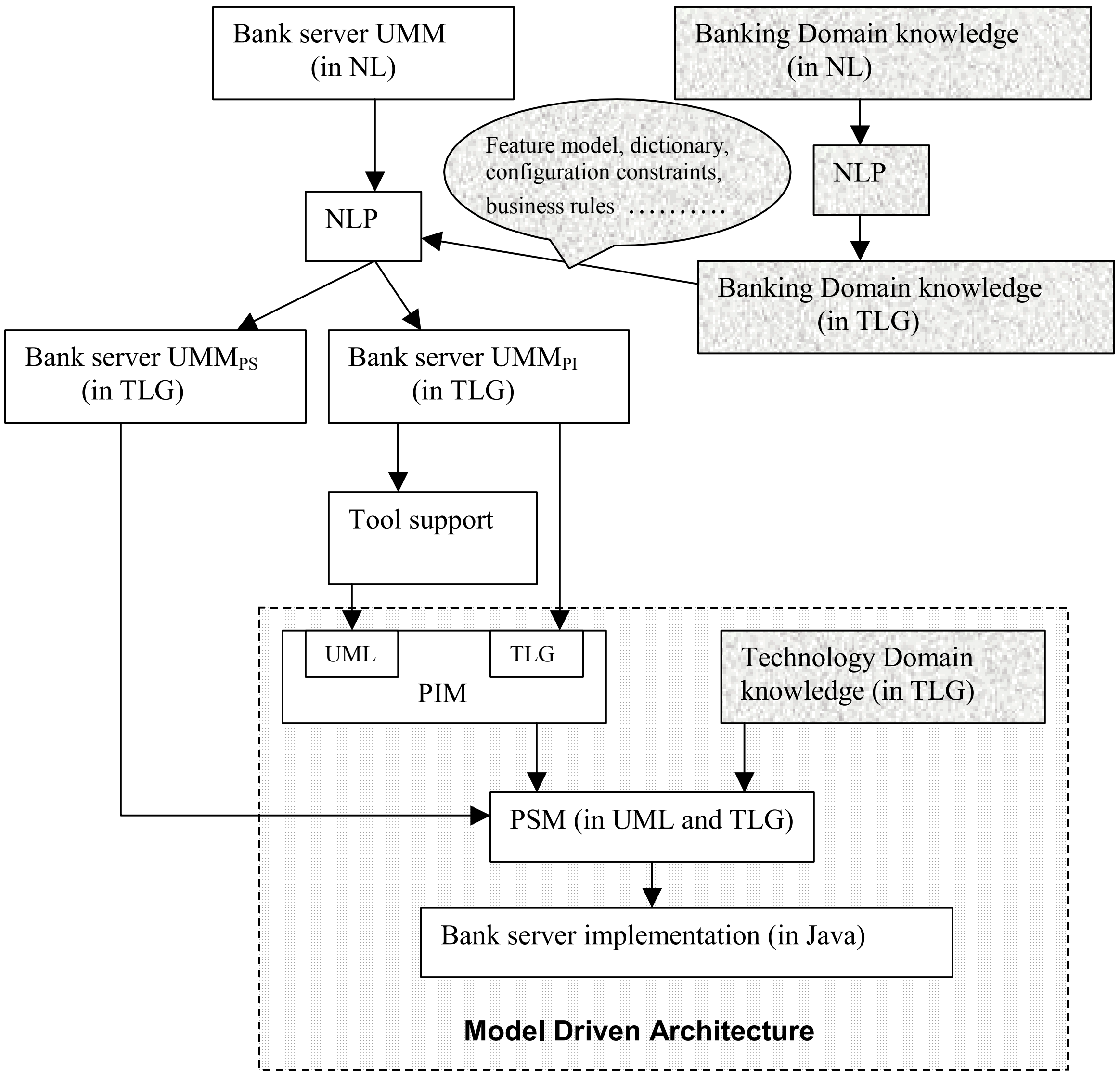


Figure 1. Natural Language Requirements Translation into Executable Models

natural language requirements processing system will use the UMM and domain knowledge base to generate platform independent and platform specific UMM specifications expressed in TLG (which we will refer to as UMM-PI and UMM-PS, respectively). UMM-PI describes the bulk of the information needed to progress to component implementation. UMM-PS merely indicates the technology of choice (e.g. CORBA). These effectively customize the component model by inheriting from the TLG classes representing the business domain with new functionality added as desired. In addition to new functionality, we also impose Quality-of-Service expectations for our components. Both the added functionality and QoS requirements are expressed in TLG so there is a unified notation for expressing all the needed information about components. The translation tool described in the previous section may be used to translate UMM-PI into a PIM represented by a combination of UML and TLG. Note that TLG is needed as an augmentation of UML to define business logic and other rules that may not be convenient to express in UML directly.

A Platform Specific Model is an integration of the PIM with technology domain specific operations (e.g. in CORBA, J2EE, etc.). These technology domain classes also are expressed in TLG. Each domain contains rules which are specific to that technology, including how to construct glue/wrapper code for components implemented with that technology and architectural considerations such as how to distinguish client code from server code. We express PSMs in TLG as an inheritance from PIM TLG classes and technology domain TLG classes. This means that PSMs will then contain not only the business-domain specific rules but also the technology-domain specific rules. The PSM will also maintain the Quality-of-Service characteristics expressed at the PIM level (a related paper [Burt02] explores the rules for this maintenance in more detail and [Burt03] explores this issue for the QoS aspect of access control in particular). Since the model is expressed in TLG, it is executable in the sense that it may be translated into executable code in a high-level language (e.g. Java). Furthermore, it supports changes at the model level, or even requirements level if the model is not refined following its derivation from the requirements, since the code generation itself is automated.



Bank server UMM (in NL)
Banking Domain knowledge (in NL)
Feature model, dictionary, configuration constraints, business rules ....
NLP
NLP
Banking Domain knowledge (in TLG)
Bank server UMM$_{PS}$ (in TLG)
Bank server UMM$_{PI}$ (in TLG)
Tool support
UML
TLG
PIM
Technology Domain knowledge (in TLG)
PSM (in UML and TLG)
Bank server implementation (in Java)
Model Driven Architecture

Figure 2. Integration of Two-Level Grammar with Model Driven Architecture

Figure 2 shows the overall view of the model-driven development from natural language requirements into executable code for the banking example we have just described.

4. Discussion

This paper has described an approach for unifying the ideas of expressing requirements in natural language, constructing Platform Independent Models for software components, and implementing the components via Platform Specific Models. The approach is specifically targeted at the construction of heterogeneous distributed software systems where interoperability is critical. This interoperability is achieved by the formalization of technology domains with rules describing how those technologies may be integrated together via the generation of glue and wrapper code. The processing of software requirements, construction of PIMs and PSMs, and specification of technology domain rules are all expressed in Two-Level Grammar, thereby achieving a unification of natural language requirements with the Model Driven Architecture approach.

For future work, we will investigate aspect-oriented technology [Kicz97] as a mechanism for specifying crosscutting relationships across components and hence improving reusability of components and reasoning about a collection of components. Such aspects of components as functional pre/post conditions and QoS properties crosscut component modules and specification of these aspects spread across component modules. Preliminary work in defining an aspect-oriented specification language is very promising [Cao03b].

We are also investigating the applicability of the UniFrame approach to real-time and embedded systems. Real-time constraints are already one of the Quality-of-Service parameters we are now validating. However, we expect that our current timing requirements will need refinement to be applicable in a true real-time setting. We are also looking at applying our modeling technology to the embedded system domain. Finally we are continuing our work in model-driven security to assure that security issues are maintained in migration from PIMs to PSMs.

5. References

[Brya02a] Bryant, B. R. and Lee, B.-S., "Two-Level Grammar as an Object-Oriented Requirements Specification Language," *Proc. HICSS-35, 35th Hawaii Int. Conf. System Sciences*, 2002, http://www.hicss.hawaii.edu/HICSS_35/HICSSpapers/PDFdocuments/STDSL01.pdf.

[Brya02b] Bryant, B. R., et al., "Formal Specification of Generative Component Assembly Using Two-Level Grammar," *Proc. SEKE 2002, 14th Int. Conf. Software Engineering Knowledge Engineering*, 2002, pp. 209-212.

[Burt02] Burt, C. C., Bryant, B. R., Raje, R. R., Olson, A. M., Auguston, M., "Quality of Service Issues Related to Transforming Platform Independent Models to Platform Specific Models," *Proc. EDOC 2002, 6th IEEE Int. Enterprise Distributed Object Computing Conf.*, 2002, pp. 212-223.

[Burt03] Burt, C. C., Bryant, B. R., Raje, R. R., Olson, A. M., Auguston, M., "Model Driven Security: Unification of Authorization Models for Fine-Grain Access Control," to appear in *Proc. EDOC 2003, 7th IEEE Int. Enterprise Distributed Object Computing Conf.*

[Cao03a] Cao, F., Bryant, B. R., Burt, C. C., Huang, Z., Raje, R. R., Olson, A. M., Auguston, M., "Automating Feature-Oriented Domain Analysis ," to appear in *Proc. SERP 2003, 2003 Int. Conf. Software Engineering Research and Practice*, 2003.

[Cao03b] Cao, F., Bryant, B. R., Raje, R. R., Auguston, M., Olson, A. M., Burt, C. C., "Assembling Components with Aspect-Oriented Modeling/Specification," to appear in *Proc. WiSME 2003, UML 2003 Workshop Software Model Engineering*.

[Czar00] Czarnecki, K., Eisenecker, U. W., *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

[Durr92] Dürr, E. H., van Katwijk, J., "VDM++ - A Formal Specification Language for Object-Oriented Designs," *Proc. TOOLS USA '92, 1992 Technology of Object-Oriented Languages and Systems USA Conf.*, 1992, pp. 263-278.

[Fran03] Frankel, D. S., *Model Driven Architecture: Applying MDA to Enterprise Computing*, Wiley Publishing, Inc., 2003.

[GME01] *GME 2000 User's Manual, Version 2.0*. ISIS, Vanderbilt University, 2001.

[IFAD00] IFAD, The VDM++ Toolbox User Manual, http://www.ifad.dk, 2000.

[Jaco99] Jacobson, I., Booch, G., Rumbaugh, J., *The Unified Software Development Process*, Addison-Wesley, 1999.

[Jura00] Jurafsky, D., Martin, J., *Speech and Language Processing*, Prentice-Hall, 2000.

[Kicz97] Kiczales, G., et al., "Aspect-Oriented Programming," *Proc. ECOOP '97, European Conf. Object-Oriented Programming*, 1997, pp. 220-242.

[Lee02a] Lee, B.-S. and Bryant, B. R., "Contextual Knowledge Representation for Requirements Documents in Natural Language," *Proc. FLAIRS 2002, 15th Int. Florida AI Research Symp.*, 2002, pp. 370-374.

[Lee02b] Lee, B.-S. and Bryant, B. R., "Contextual Processing and DAML for Understanding Software Requirements Specifications," *Proc. COLING 2002, 19th Int. Conf. Computational Linguistics*, 2002, pp. 516-522.

[Lee02c] Lee, B.-S., Bryant, B. R., "Automation of Software System Development Using Natural Language Processing and Two-Level Grammar," *Proc. 2002 Monterey Workshop Radical Innovations Software and Systems Engineering in the Future*, 2002, pp. 244-257.

[Quat00] Quatrani, T., *Visual Modeling with Rational Rose 2000 and UML*, Addison-Wesley, Reading, MA, 2000.

[Raje00] Raje, R. R., "UMM: Unified Meta-object Model for Open Distributed Systems," *Proc. ICA3PP, 4th IEEE Int. Conf. Algorithms and Architecture for Parallel Processing*, 2000, pp. 454-465.

[Raje01] Raje, R. R., Auguston, M., Bryant, B. R., Olson, A. M., and Burt, C. C., "A Unified Approach for the Integration of Distributed Heterogeneous Software Components," *Proc. 2001 Monterey Workshop Engineering Automation for Software Intensive System Integration*, 2001, pp. 109-119.

[Raje02] Raje, R. R., Auguston, M., Bryant, B. R., Olson, A. M., Burt, C. C., "A Quality of Service-based Framework for Creating Distributed Heterogeneous Software Components," *Concurrency and Computation: Practice and Experience 14*, 12 (2002), 1009-1034.

[Warm99] Warmer, J., Kleppe, A., *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1999.

[Wils99] Wilson, W. M., "Writing Effective Natural Language Requirements Specifications," Naval Research Laboratory, 1999.

[Yang02] Yang, C., Lee, B.-S., Bryant, B. R., Burt, C. C., Raje, R. R., Olson, A. M., Auguston, M., "Formal Specification of Non-Functional Aspects in Two-Level Grammar," *Proc. UML 2002 Workshop Component-Based Software Engineering and Modeling Non-Functional Aspects(SIVOES-MONA)*, 2002, http://www-verimag.imag.fr/SIVOES-MONA/uniframe.pdf.

[Zhao03] Zhao, W., Bryant, B. R., Burt, C. C., Gray, J. G., Raje, R. R., Olson, A. M., Auguston, M. "A Generative and Model Driven Framework for Automated Software Product Generation," *Proc. CBSE 6, 6th Workshop Component-Based Software Engineering*, 2003, http://www.csse.monash.edu.au/~hws/cgi-bin/CBSE6/Proceedings/papersfinal/p31.pdf.

Sidestepping verification complexity with supervisory control

Ugo Buy
Dept. of Computer Science
University of Illinois at Chicago
buy@uic.edu

Houshang Darabi
Dept. of Mechanical and Industrial Engineering
University of Illinois at Chicago
hdarabi@uic.edu

September 12, 2003

Abstract

While the goal of verification is to check whether a model of the system under consideration has a desired property, supervisory control achieves correctness by adding a so-called supervisor that prevents the occurrence of incorrect behaviors to the original system. Supervisory control methods are appealing because they can be much more tractable than the corresponding verification problems. Here we first examine two supervisory control algorithms, one for enforcing mutual exclusion properties and the other for enforcing real-time deadlines on Petri net models of the controlled system. Next, we argue that use of supervisory control methods may lead to a simpler and more effective coding style for embedded software than current practices. Finally, we highlight research issues that must be addressed in order to permit widespread application of supervisory control methods.

1 Introduction

Embedded systems often exhibit features typical of concurrent and real-time systems. The automatic verification of concurrency and timing properties has been studied extensively for over two decades; however, progress in this area has been slow. One reason for this state of affairs is the computational intractability of most verification problems. Here we suggest that supervisory control can be a more practical approach than automatic verification for a broad class of embedded systems. While verification seeks to determine whether a model of the system under consideration has a desired property, supervisory control achieves correctness by adding a so-called supervisor that inhibits incorrect behaviors to the original system.

The supervisory control methods discussed here are suitable for embedded systems that can be modeled as a discrete event system (DES). We are specifically interested in control systems for discrete manufacturing processes, although the same supervisory control techniques are generally applicable to other DES models as well. A discrete manufacturing plant consists of machines for producing, moving, and assembling parts on a shop floor. Control systems for such plants must enforce a variety of correctness properties, including traditional concurrency and timing properties.

Supervisory control methods for discrete event systems typically employ finite state automata or Petri nets to model a DES [4, 13, 14, 17]. Here we focus on Petri-net-based models for two reasons. First, Petri nets support computationally-tractable methods for supervisor synthesis. We summarize two such methods below. These methods use concepts specific to Petri nets, such as P-invariants and net unfolding. Thus, these methods are not applicable to discrete event systems modeled by finite-state automata. Existing supervisory control methods for automata models usually resort to the cross-product of finite-state automata for supervisor synthesis, which is likely to lead to state-space explosion. Second, Petri nets are used extensively for specification and analysis of discrete manufacturing systems. For instance, the language of Sequential Function Charts (SFCs) is a straightforward extension of Petri nets. SFCs are part of the IEC 61131 standard for manufacturing control languages; they are supported by popular commercial products such as Matlab and RSLogix 5000 [8].

Given a Petri net and a set of correctness properties, *supervisory control* methods can enforce the given properties by cleverly disabling net transitions that could lead to a violation of the properties. Thus, the supervisory controller of a Petri net $\mathcal{N}$ is a subnet $\mathcal{S}$ that is added to $\mathcal{N}$ in order to enforce the properties of interest. In this case, $\mathcal{N}$ is said to be the *controlled net* [6]. A supervisor is said to be *maximally permissive* if it does not disable any behavior that satisfies the property of interest while preventing the occurrence of all behaviors that violate the property.

The first method that we survey uses Petri net P-invariants to enforce a broad variety of mutual exclusion constraints of the controlled net [4, 11, 17]. An advantage of this method, which has been studied extensively in the past decade, is that its worst-case computational complexity is polynomial in the size of the controlled system. This complexity is dramatically lower than the complexity of the corresponding verification problems [16]. An additional advantage is that the method generates maximally permissive supervisors [17].

The second mehod uses the concept of *transition latency* to enforce real-time deadlines in time Petri nets [2]. This is a new method; however, it is one of few existing techniques for enforcing real-time properties in timed models. In brief, the latency of a transition t is the latest time when t can be fired while guaranteeing that the given deadline is met. Transition latencies are computed by *unfolding* the ordinary Petri net underlying the time Petri net that models the controlled system [3, 9, 15]. The complexity of this method is dominated by the unfolding, which is at worst exponential in the size of the controlled net [3]. However, we believe that the average-case complexity will be polynomial.

On the positive side, supervisory control methods not only provide a tractable alternative to intractable verification problems. Supervisory controllers can also lead to a novel programming paradigm for embedded and real-time systems. In contrast with current practices, in the new paradigm a programmer would first code an embedded system without the burden of building desired correctness properties (e.g., compliance with mutual exclusion or timing constraints) directly into the code. The programmer would then submit this code, along with a control specification, to a *supervisor generator*, which would augment the programmer's code with a supervisor capable of enforcing the properties contained in the specification.

On the negative side, several issues may adversely affect the applicability of supervisory controllers. For instance, events in the controlled system may not be *observable* and *controllable* to the extent needed for supervisor generation. Informally, an event is said to be observable if its occurrence can be detected by the supervisor. An event is controllable if its occurrence can be inhibited by the supervisor. Moreover, the integration of supervisors for different properties in order to guarantee correctness with respect to all properties considered must be explored.

This paper is organized as follows. In Section 2 we introduce some required definitions. Section 3 summarizes a method for enforcing mutual exclusion properties of generalized Petri nets. In Section 4, we present our paradigm for generating deadline-enforcing supervisors in time Petri nets. In Section 5, we discuss the potential advantages and disadvantages of supervisory control methods in software development for embedded systems.

2 Definitions

An *ordinary Petri net* is a four-tuple $N = (P, T, F, M_0)$ where P and T are the node sets and F the edges of a directed bipartite graph, and $M_0 : P \rightarrow \mathbb{N}$ is called the *initial marking* of $\mathcal{N}$, where $\mathbb{N}$ denotes the set of nonnegative integers. In general, a marking or state of N assigns a nonnegative number of tokens to each $p \in P$.

A transition is *enabled* when all its input places have at least one token. When an enabled transition t is fired, a token is removed from each input place of t and a token is added to each output place; this gives a new marking (state). Petri net $N = (P, T, F, M_0)$ is *safe* if $M_0 : P \rightarrow \{0, 1\}$, and if all markings reachable by legal sequences of transition firings from the initial marking have either 0 or 1 tokens in every place.

A *generalized* Petri net associates a positive weight *w* with each arc $f \in F$. If f goes from an input transition t_1 to an output place p, then *w* tokens are deposited in p whenever t_1 fires. If f goes from input place q to output transition t_2, then at least *w* tokens are needed in q in order for t_2 to be enabled. In this case, the firing of t_2 removes *w* tokens from q.

A *time Petri net* [1, 10] is a five-tuple (P, T, F, M_0, S) where (P, T, F, M_0) is an ordinary Petri net, and S associates a *static (firing) interval* $\mathcal{I}(t) = [a, b]$ with each transition *t*, where a and b are rationals in the range $0 \leq a \leq b \leq +\infty$, with $a \neq \infty$.

Static intervals change the behavior of a time Petri net with respect to an ordinary Petri net in the following way. If transition t with $\mathcal{I}(t) = [a, b]$ becomes enabled at time θ_0, then transition t must fire in the time interval $[\theta_0 + a, \theta_0 + b]$, unless it becomes disabled by the removal of tokens from some input place in the meantime. The *static earliest firing time* of transition t is a; the *static latest firing time* of t is b; the *dynamic earliest firing time* of t is $\theta_0 + a$; the *dynamic latest firing time* of t is $\theta_0 + b$; the *dynamic firing interval* of t is $[\theta_0 + a, \theta_0 + b]$.

The state of a time Petri net is a triple (M, Θ, I), where M is the marking of the underlying untimed Petri net, Θ is the global time, and I is a vector containing the dynamic firing interval of each transition enabled by M. The initial state of a time Petri net consists of its initial marking, time 0, and a vector containing the static firing interval of each transition enabled by this marking.

A *firing schedule* for time Petri net $\mathcal{N}$ is a finite sequence of ordered pairs (t_i, θ_i) such that transition t_1 is fireable at time θ_1 in the initial state of $\mathcal{N}$, and transition t_i is fireable at time θ_i from the state reached by starting in the initial state of $\mathcal{N}$ and firing the transitions t_j for $1 \leq j < i$ in the schedule at the given times.

3 Enforcing mutual exclusion

This supervisory control method enforces sets of linear mutual exclusion constraints on the reachable markings of the controlled net $\mathcal{N}$. For instance, if $\mathcal{N}$ has m transitions

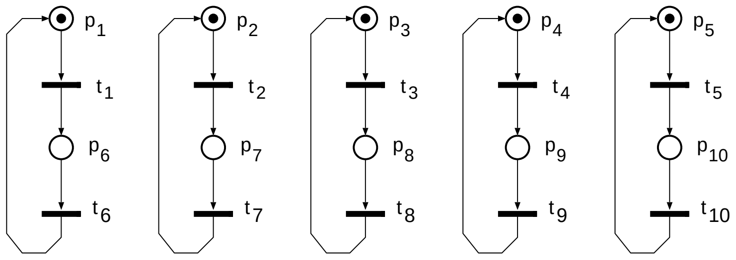


Figure 1: Example of controlled Petri net for readers and writers example.

and n places, each constraint may take the following form:

$$\sum_{i=1}^{n} l_i \cdot \mu_i \leq \beta \tag{1}$$

Variable μ_i represents the marking of place p_i, l_i is an integer coefficient, and β is an integer constant [4, 11, 17].

Given controlled net $\mathcal{N}$ and a set of linear constraints similar to inequality (1) above, this supervisory control method exploits a property of Petri net P-invariants. A P-invariant is a subset P_I of $\mathcal{N}$'s place set P such that the weighted sum of the tokens residing in places $p_i \in P_I$ remains constant in all reachable markings of $\mathcal{N}$. Inequality (1) can be transformed into a P-invariant equality by adding a slack variable μ_C:

$$\sum_{i=1}^{n} l_i \cdot \mu_i + \mu_C = \beta \tag{2}$$

Variable μ_C represents the marking of a control place p_C that enforces inequality (1). Consequently, a set of k linear inequalities can be enforced by a supervisory controller consisting of k control places and zero transitions.

The arc subset connecting P_C, the set of control places, to P, the place set in the controlled net, can be easily computed by a simple matrix multiplication. Let D be the $n \times m$ incidence matrix of a Petri net N with m transitions and n places. Entry $d_{i,j}$ is a positive (negative) integer w if N contains a weight w arc from transition t_j to place p_i to (from p_i to t_j). In general, the place invariants of N are the integer solutions to the following vector equation:

$$x^T \cdot D = 0^T \tag{3}$$

Here x^T is a transposed n-vector representing the integer coefficients of the net's place invariants and 0^T is a transposed m-vector filled with zeros.

Therefore, the P-invariants induced by a set of k inequalities (1) must satisfy the following equation:

$$[L \quad I] \cdot D = 0 \tag{4}$$

where L represents a $k \times n$ matrix containing the coefficients of inequality constraints (1), I is the unit matrix of size k and D is the incidence matrix of the net consisting of $\mathcal{N}$ and the supervisory controller. Clearly D has $n + k$ rows, where n is the number of places in $\mathcal{N}$, and m columns, one for each transition in $\mathcal{N}$:

$$D = \begin{bmatrix} D_N \\ D_C \end{bmatrix} \tag{5}$$

Here D_N is the $n \times m$ incidence matrix of $\mathcal{N}$ and D_C is the $k \times m$ incidence matrix of the supervisor net $\mathcal{S}$. From equations (4) and (5), we can find D_C as follows:

$$D_C = -L \cdot D_N \tag{6}$$

Thus, the desired supervisory controller can be found by a simple matrix multiplication involving the incidence matrix of the controlled net and the coefficients appearing in the inequality constraints to be enforced. The elements of D_C will be integers, as required, because L and D_N are integer matrices. Yamalidou et al showed that the supervisory controllers generated in this fashion are maximally permissive [17].

We illustrate the potential benefits of P-invariant-based supervisory control by applying this method to the readers and writers problem, a classical example of mutual exclusion. We consider a version with 3 readers and 2 writers. As usual, multiple readers are allowed in the buffer when no writer is in the buffer; however, each writer excludes both other writers and all readers from the buffer. Figure 1 shows a Petri net for a version in which the mutual exclusion constraints are not enforced. Places p_1, p_2 and p_3 represent the three readers in the idle state. When p_6, p_7 and p_8 have a token, the three readers are in the buffer. Likewise, places p_4 and p_5 represent the idle states of the two writers; a token in place p_9 or p_{10} means that a writer is in the buffer. To enforce the mutual exclusion constraints we write the following three inequalities:

$$\begin{aligned} p_6 + p_9 + p_{10} &\leq 1 \\ p_7 + p_9 + p_{10} &\leq 1 \\ p_8 + p_9 + p_{10} &\leq 1 \end{aligned}$$

The first constraint stipulates that at most one of the first reader and the two writers can be in the buffer simultaneously. The remaining two constraints stipulate the

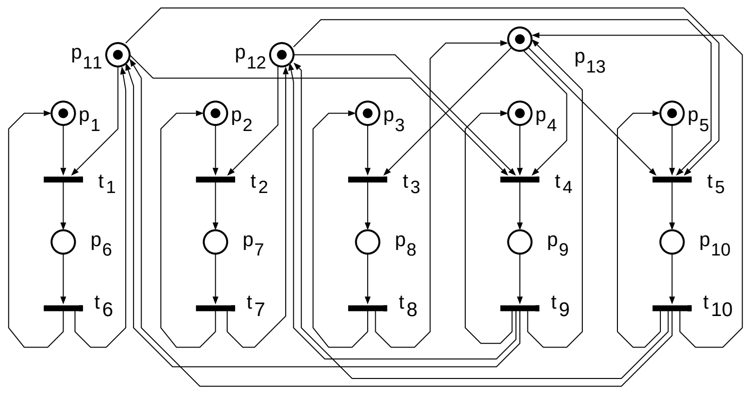


Figure 2: Petri net for readers and writers example with supervisory controller.

same condition for the second and third reader. Figure 2 shows the Petri net for the readers and writers example with the supervisory controller obtained by formula (6) above. Places p_{11}, p_{12}, p_{13}, and their incident arcs are the supervisor. In particular, place p_{11} enforces the mutual exclusion between the first reader and the two writers. Places p_{12} and p_{13} play a similar role for the second and third reader. The mutual exclusion among writers follows from each of the three constraints above.

The significance of this method is that it can enforce mutual exclusion constraints of a Petri net model in time polynomial in the size of the controlled net and the supervisor net. Therefore, the overall complexity will be polynomial whenever a mutual exclusion problem can be translated into a linear system containing a number of inequalities polynomial in the size of the controlled net. This performance is in sharp contrast with the verification of mutual exclusion properties, for which no general-purpose polynomial-time algorithm is known. When it is applicable, the approach based on supervisory control is likely to be vastly more scalable than verification. This method has also been extended to the case of Petri nets with unobservable and uncontrollable transitions [11]. Yamalidou et al defined other extensions including the case of "greater-than" constraints, constraints expressed as logical formulas, and constraints involving the firing vectors of the controlled net [17]. Finally, Iordache et al defined a method for enforcing net liveness (e.g., freedom from deadlock) in the controlled net [7].

4 Enforcing real-time deadlines

We report preliminary results on a method for enforcing real-time deadlines in time Petri nets [10]. Given a time Petri net $\mathcal{N} = (P, T, F, M_0, S)$, a net transition t_D, and a deadline λ, our method seeks to generate a supervisory controller that forces t_D to fire no more than λ time units since the latest of the previous firing of t_D and the beginning of a firing sequence. Throughout this section we assume that $\mathcal{N}$ is a safe and live time Petri net.

Our paradigm for generating deadline-enforcing supervisory controllers consists of three steps. First, we compute a so-called *transition latency* for each transition t in $\mathcal{N}$. Given a time Petri net $\mathcal{N} = (P, T, F, M_0, S)$, the latency $l(t)$ of a transition $t \in T$ is the maximum delay between any firing of t and the next firing of t_D, along firing schedules permitted by the supervisory controller for net $\mathcal{N}$. Thus, the latency of t is an upper bound on the time required for t_D to fire after t fires.

Second, we define a so-called *clock net* $\mathcal{C}$, a time Petri net whose places correspond to transition latencies identified earlier. A place in a clock net is used to disable dynamically transitions whose firing may prevent t_D from meeting λ. Of course, a transition t should be allowed to fire only when t's latency is no greater than the time left until the deadline on the firing of t_D expires.

Third, we synthesize a supervisory controller $\mathcal{S}$ based on nets $\mathcal{N}$ and $\mathcal{C}$. Controller $\mathcal{S}$ disables transitions in $\mathcal{N}$ based on the marking of places in $\mathcal{C}$. In particular, $\mathcal{S}$ dynamically disables transitions whose latency is greater than the time left until the deadline on the firing of t_D.

Consider, for instance, the time Petri net appearing in Figure 3. Suppose that target transition t_7 must be fired within 51 time units from the initial state. In order for t_7 to fire, transition t_2 must fire first. Since t_2 is in conflict with t_1, a supervisory controller must disable t_1 some time before the deadline expires. In this case, we can set the latency of t_2 to 25 time units, the sum of the static latest firing delays of t_3, t_5, and t_7. After t_1 fires, transitions t_3, t_4, t_6, t_2, t_3, t_5 and t_7 must be fired in order for the deadline to be met. The sum of their static latest firing times is 48 time units. Thus, it is safe to fire t_1 if at least 48 time units remain until t_7 must be fired.

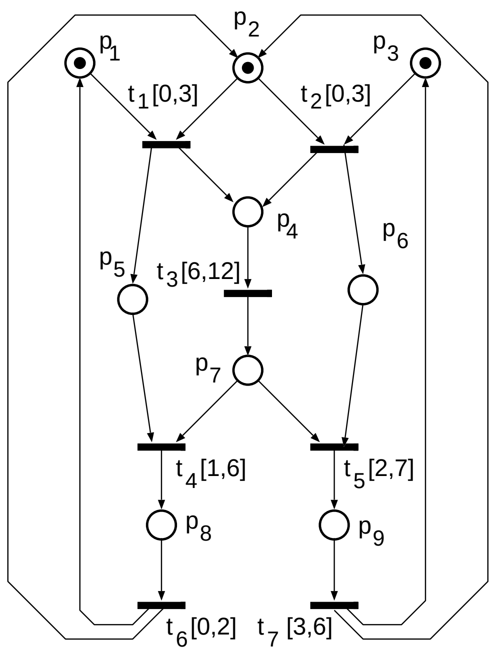


Figure 3: Example of a time Petri net.

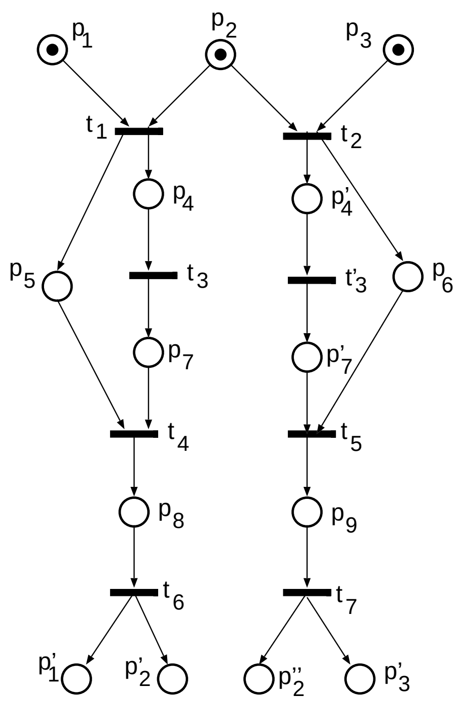


Figure 4: Unfolding of the Petri net appearing in Figure 3.

4.1 Computing transition latencies

We believe that various approaches can be followed when defining the latency of $\mathcal{N}$ transitions. Here we discuss a technique called *net unfolding* [3, 5, 9, 15]. We choose this technique for two reasons. First, net unfolding explicitly captures the causal relationship on transition firings for the Petri net under consideration. Thus, by unfolding net $\mathcal{N}$ we can define reasonably tight latency values. Second, unfolding $\mathcal{N}$ allows us to identify $\mathcal{N}$ transitions that need not be disabled in order for deadline λ to be met. In general, λ can be enforced by disabling only a small subset of transitions in the controlled net. For instance, it is sufficient to disable transition t_1 in a timely manner in order to force transition t_7 to fire in Figure 3. This fact can lead to reductions in the size of subnets $\mathcal{C}$ and $\mathcal{S}$ below.

We require the following definitions. Consider nodes x and y in an (untimed) ordinary Petri net. Node *x precedes y*, denoted by $x < y$ if there is a directed path from x to y in the Petri net. Nodes x and y are *in conflict*, denoted by $x\#y$, if the Petri net contains two distinct paths originating at the same place *p* that diverge immediately after *p* and lead to x and y. When $x\#x$ holds, node x is said to be in *self-conflict*. Nodes x and y are *concurrent* if they are not in conflict with each other and neither node precedes the other.

An *occurrence net* is an unmarked ordinary Petri net $\mathcal{O} = (P_O, T_O, F_O)$ subject to these conditions [3]:

1. $\forall p \in P_O$, p has at most one input arc.
2. $\mathcal{O}$ is acyclic.
3. Each node $x \in P_O \cup T_O$ is finitely preceded, meaning that the number of nodes $y \in P_O \cup T_O$ such that $y < x$ is finite.
4. No node $x \in P_O \cup T_O$ is in self-conflict.

Given a controlled net $\mathcal{N}$, consider $\mathcal{M}$, the ordinary Petri net underlying $\mathcal{N}$, so that $\mathcal{M} = (P, T, F, M_0)$. An *unfolding* of $\mathcal{M}$ is a marked, labeled occurrence net $\mathcal{U} = (P_U, T_U, F_U, M_{0U}, l_U)$, where l_U is a function mapping each node $x \in P_U \cup T_U$ to a node $l_U(x)$ in $\mathcal{M}$. In brief, each element of $\mathcal{U}$ is an “occurrence” of its image in $\mathcal{M}$. The formal definition of a net unfolding can be found elsewhere [3] along with algorithms for generating unfoldings of ordinary Petri nets. Here we simply report an example of a net unfolding.

Figure 4 shows an unfolding of the ordinary Petri net underlying the time Petri net in Figure 3. Places p_1, p_2, and p_3, which are initially marked, are mapped to the homonymous places in $\mathcal{N}$. Transitions t_1 and t_2 are mapped similarly. However, place p_4 in $\mathcal{N}$ is in self-conflict because it can be reached from p_2 either through transition t_1 or t_2. Thus, this place is represented by two places, p_4 and p'_4, in Figure 4. Transition t_3 and place p_7 are also split into two nodes for the same reason. Finally, places p'_1, p'_2, p''_2, and p'_3, represent the so-called *cut-off* points of the unfolding. When these places are marked, the net returns to its initial state.

We define transition latencies from the unfolding of the untimed net underlying controlled net $\mathcal{N}$. First, for each transition $t_u \in T_U$, the transition set of unfolding $\mathcal{U}$, we associate the static latest firing time of $l_U(t_u)$, the image of t_u in $\mathcal{N}$, with t_u. Second, we examine backward paths from each occurrence of t_D in $\mathcal{U}$ to the initial places of $\mathcal{U}$ and forward paths from t_D occurrences to the cut-off places of $\mathcal{U}$. We add the static latest firing times of the transitions that we find along these paths and we associate the partial sums with such transitions. Define c to be the longest backward path from a t_D occurrence in $\mathcal{U}$ to the initial places of $\mathcal{U}$. Define d to be the longest forward path from a t_D occurrence to the cut-off places of $\mathcal{U}$. Third, we consider paths from initial places to the cut-off places that do not include $\mathcal{U}$ transitions mapping into t_D. These paths may correspond to cyclic behaviors of net $\mathcal{N}$ in which t_D is not fired (e.g., if t_D is disabled along these paths). We add c to the combined delays along such paths. The resulting values yield the latencies for $\mathcal{N}$ transitions.

In the example appearing in Figure 4, we associate delays as follows: $t_1 \to 3$, $t_2 \to 3$, $t_3 \to 12$, $t_4 \to 6$, $t_5 \to 7$, $t_6 \to 2$, and $t_7 \to 6$ during the first phase of the algorithm. Next, we consider paths originating at target transition t_7. Since t_7 feeds directly into cut-off places p_2'' and p_3', we discard paths toward cut-off places. Backward paths from t_7 to initial places p_2 and p_3 yield the following latencies: $t_7 \to 0$, $t_5 \to 6$, $t_3 \to 13$, and $t_2 \to 25$. Finally, we consider the cycle involving firing sequence $\sigma = t_1, t_3, t_4, t_6$. The sum of the static latest firing delays along σ is 23. We add the latency of transition t_2 and the latest firing delay of t_2 to the delays computed on the cycle. This yields the following latency values: $t_1 \to 48$, $t_3 \to 36$, $t_4 \to 30$, $t_6 \to 28$. Transition t_3 is seemingly given two different latency values because this transition is in self-conflict. When this happens, we define the latency to be the least value.

4.2 Clock nets

We compute a clock net $\mathcal{C} = (P_C, T_C, F_C, M_{0C}, S_C)$ for a controlled net $\mathcal{N} = (P, T, F, M_0, S)$ based on the transition latencies and choice points previously defined with net unfolding. We specifically consider a subset $T_H \subseteq T$ of $\mathcal{N}$ transitions that are involved in choice points (i.e., because they share at least one input place).

First, we add to P_C one place for each distinct element in the set of latency values $L = \{v \mid \exists t \in T_H \text{ and } v = l(t)\}$. Given a latency value v, we denote the place corresponding to v by p_v. In addition, P_C contains a place p_λ corresponding to deadline λ and a place p_D for resetting the clock net after the firing of t_D. Figure 5 shows the clock net for the controlled net appearing in Figure 3. Here set T_H consists of transitions t_1 and t_2; places p_{25} and p_{48} map the latencies of these transitions in the clock subnet. Place p_{51} models deadline λ.

We define the transition set T_C, static delay intervals S_C, and flow relation F_C of $\mathcal{C}$ as follows. First, we insert a transition between pairs of clock net places with consecutive index values. The static delay of each such transition is the difference between the index values of its input and output place. In Figure 5 this yields transitions t_8 and t_9 with delays of 3 and 23. Next, we define an arc from t_D to p_D, and we add a group of $|P_C| - 1$ zero-delay transitions to T_C, one transition for each place $p_k \in P_C$, except for p_D. A token in p_D enables one of these transitions immediately after t_D fires. The transition removes the token from p_D and from one of the other places in P_C; it deposits a token in p_λ and in a suitable number of $\mathcal{S}$ control places described below. This completes the resetting of $\mathcal{C}$ and $\mathcal{S}$. In Figure 5, transitions t_{10}, t_{11}, and t_{12} reset the clock subnet and supervisor after t_7 fires. Additional details can be found elsewhere [2].

4.3 Supervisory controllers

Supervisory controllers enforce deadline λ on the firing of transition t_D in net $\mathcal{N} = (P, T, F, M_0, S)$ with clock net $\mathcal{C} = (P_C, T_C, F_C, M_{0C}, S_C)$. Let $P_V = P_C - \{p_D\}$. The supervisory control constraint is expressed by:

$$\text{Disable } t \in T_H \text{ if } p_v \in P_V \text{ marked with } l(t) = v \quad (7)$$

We note that by construction, for any marking of clock net $\mathcal{C}$, all places in P_V combined will always contain exactly one token. Unless t_D is fired, the token in P_V will always move toward places with lower index values. Constraint (7) above states that a transition $t \in \mathcal{N}$ is disabled whenever the token in P_V moves to a place p_v whose index v is equal to t's latency $l(t)$. Therefore, control constraint (7) above will disable all transitions that might delay the firing of t_D by more than v units, the index of the marked state of $\mathcal{C}$. Moreover, once t is disabled, t is not allowed to fire again until after target transition t_D has been fired. As a result, all the transitions that may result in the violation of deadline λ on the firing of t_D are disabled.

We implement constraint (7) above by defining two places, q_1 and q_2, and one transition r for each $t \in T_H$. The rules for defining arcs incident on q_1, q_2 and r are discussed elsewhere [2].

Figure 5 shows the supervisory controller for the net in Figure 3. This controller disables transitions t_1 and t_2 when transitions t_8 and t_9 are fired. For instance, when t_8 fires place c_3 becomes marked, which enables transition t_{13}. The firing of t_{13} causes the removal of the token from place c_1; this action disables transition t_1. Transition t_9

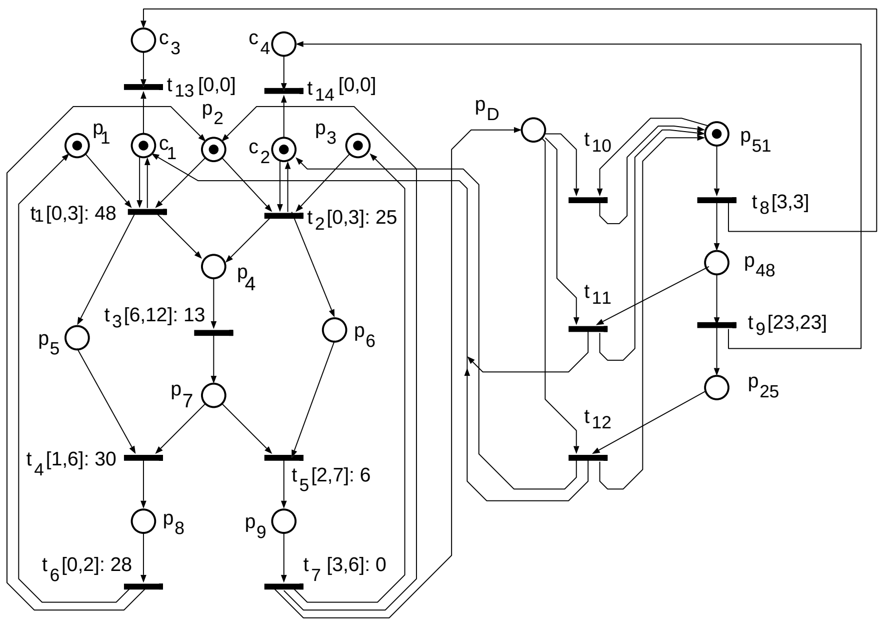


Figure 5: Supervisory controller and clock net of time Petri net appearing in Figure 3.

similarly disables transition t_2. Additional details can be found elsewhere [2].

5 Assessment

The two methods discussed earlier indicate that supervisory control may have significant benefits on the development of software for concurrent and real-time systems. The most significant advantage of supervisory control is reduced computational complexity with respect to the corresponding verification algorithms. For instance, in Section 3 we saw that a broad variety of mutual exclusion constraints can be enforced in time polynomial in the size of the system under consideration. This is in sharp contrast to the verification of mutual exclusion properties, which is computationally intractable. Although we lack empirical data on the real-time method discussed in Section 4, we believe that on average this method will also be tractable. The verification of real-time systems is generally considered even more complex than the case of untimed concurrent systems. When they are applicable, supervisory control methods may provide greater help to developers of embedded systems than existing techniques.

However, the full potential of supervisory control in software development is more far-reaching than just guaranteeing that certain correctness properties are met. The availability of supervisory control tools could free programmers from the need to build compliance with correctness properties directly into their code. The version of the readers and writers example shown in Figure 1 is a case in point. This version could be obtained by translation from software that was deliberately written without paying attention to its mutual exclusion constraints. However, a supervisory controller can subsequently enforce these and other properties that are expressed as linear equalities and inequalities on net markings and firing vectors.

Thus, the use of supervisory control methods could lead a new programming paradigm for concurrent and real-time systems. In this paradigm, a programmer would first write a version of the program without being concerned about complying with mutual exclusion and real-time properties. Next, the programmer would submit this program along with a control specification to a supervisory control tool. The tool would then translate the program into a Petri-net model and generate suitable supervisors. Finally, the tool would add code that enforces the control specification to the original code.

While supervisory control methods hold considerable promise for the development of concurrent and real-time systems, the widespread application of these methods also faces formidable obstacles. Petri net transitions may not be observable and/or controllable to the extent needed for supervisor definition. This could happen, for instance, in wireless sensor networks, special kinds of embedded systems [12]. These networks often lack the ability for a node (i.e., an embedded system equipped with sensors) to know instantaneously and control events in different nodes.

Additional obstacles may arise when attempting to integrate multiple supervisory control methods in order to enforce different properties. For instance, it is currently unclear whether the two methods that we discussed earlier can be effectively combined in an effort to guarantee simultaneously mutual exclusion *and* real-time properties.

Liveness properties, such as freedom from deadlock, pose additional challenges to the application of supervisory control methods. Although freedom from deadlock is an intractable verification problem, this is considered the "easiest" property to check through verification. The same does not hold in the world of supervisory control; the definition of supervisors for enforcing Petri net liveness is much more challenging than, say, enforcing mutual exclusion properties expressed as linear constraints [7]. A method by He and Lemmon, who use net unfoldings to enforce liveness, seems especially promising [5].

To date, several research issues must be investigated in order to answer some of the questions regarding the applicability of supervisory control to software development. First, additional supervisory control methods must be defined for enforcing different properties. In the case of the readers and writers example, it is quite conceivable to define versions in which the readers or the writers have priority or in which read and write requests should be handled in FIFO order. Supervisory control strategies for these kinds of specifications are generally not available yet. Second, we must collect empirical data on the applicability of supervisory control in software development. At the very least, we should find out how often mutual exclusion constraints can be expressed through a small number of linear constraints in the form (1). The complexity of net unfolding when applied to real-world software problems must also be assessed empirically because this techique is crucial both to liveness-enforcing and deadline-enforcing supervisors.

6 Conclusions

We briefly summarized two supervisory control methods for concurrent and real-time systems. Although these methods have not reached the level of maturity needed to permit the creation of tools for software development, they hold considerable promise because they are generally more tractable than the corresponding verification algorithms. For these reasons, we should investigate research directions that may lead to widespread applications of supervisory control in software development for concurrent and real-time systems, such as embedded systems.

References

[1] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Trans. Softw. Eng.*, 17(3):259–273, Mar. 1991.

[2] U. Buy and H. Darabi. Deadline-enforcing supervisory control for time Petri nets. In *CESA'2003 – IMACS Multiconference on Computational Engineering in Systems Applications*, Lille, France, July 2003. Available on CD-ROM.

[3] J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. *Formal Methods in System Design*, 20(3):285–310, May 2002.

[4] A. Giua, F. DiCesare, and M. Silva. Generalized mutual exclusion constraints for nets with uncontrollable transitions. In *Proceedings IEEE Int. Conf. on Systems, Man, and Cybernetics*, pages 974–979, Chicago, Illinois, Oct. 1992.

[5] K. X. He and M. D. Lemmon. Liveness-enforcing supervision of bounded ordinary Petri nets using partial order methods. *IEEE Transactions on Automatic Control*, 47(7):1042–1055, July 2002.

[6] L. E. Holloway, B. H. Krogh, and A. Giua. A survey of Petri net methods for controlled discrete event systems. *Discrete Event Dynamic Systems: Theory and Applications*, 7:151–190, Apr. 1997.

[7] M. V. Iordache, J. O. Moody, and P. J. Antsaklis. Synthesis of deadlock prevention supervisors using Petri nets. *IEEE Transactions on Robotics and Automation*, 18(1):59–68, 2002.

[8] R. W. Lewis. Programming industrial control systems using IEC 1131-3. Technical report, The Institution of Electrical Engineers, 1998.

[9] K. L. McMillan. A technique of state space search based on unfolding. *Formal Methods in System Design*, 6(1):45–65, Jan. 1995.

[10] P. M. Merlin and D. J. Farber. Recoverability of communication protocols—implications of a theoretical study. *IEEE Trans. Communications*, COM-24(9):1036–1043, Sept. 1976.

[11] J. O. Moody and P. J. Antsaklis. Petri net supervisors for DES with uncontrollable and unobservable transitions. *IEEE Transactions on Automatic Control*, 45(3):462–476, Mar. 2000.

[12] G. J. Pottie and W. J. Kaiser. Wireless integrated network sensors. *Commun. ACM*, 43(5):51–58, May 2000.

[13] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal of Control and Optimization*, 25(1):206–230, 1987.

[14] A. S. Sathaye and B. H. Krogh. Supervisor synthesis for real-time discrete event systems. *Discrete Event Dynamic Systems: Theory and Applications*, 8, 1998.

[15] A. Semenov and A. Yakovlev. Verification of asynchoronous circuits using time Petri net unfolding. In *Proceedings of the 33rd Design Automation Conference (DAC96)*, pages 59–62, Las Vegas, Nevada, June 1996.

[16] R. N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Inf.*, 19:57–84, 1983.

[17] K. Yamalidou, J. Moody, M. Lemmon, and P. Antsaklis. Feedback control of Petri nets based on place invariants. *Automatica*, 32(1):15–28, 1996.

A Vision for Integration of Embedded System Properties Via a Model-Component-Aspect System Architecture[1]

Christopher D. Gill

Department of Computer Science and Engineering
Washington University, St.Louis
cdgill@cse.wustl.edu

Abstract *– Current approaches to developing complex embedded systems, particularly those with constraints in addition to their functional requirements, suffer significant limitations when moving from system requirements to implementation. Chief among these limitations is the inability of current development approaches to achieve an appropriate match between the sets of abstractions in their programming models and the inherent structure of concerns that appears in modern embedded systems.*

While emerging techniques such as model-integrated computing, component middleware and aspect-oriented programming address parts of this problem, how they are to be combined for greatest effect is still an open research problem. This paper outlines the problem of requirements-driven integration of multiple embedded system properties within implementation software, and offers an architectural vision for system software in support of requirements-driven development of embedded systems.

Keywords *– distributed real-time and embedded systems, generative programming*

A. INTRODUCTION

Complex large-scale systems, especially those with constraints on timeliness, footprint, or other properties outside the *functional* semantics of the application, are posing an increasing challenge to current approaches to software and system engineering. Specifically, developing *correct* distributed real-time and embedded (DRE) systems requires programmers to address constraints in two distinct semantic dimensions:

- *Functional* constraints – algorithmic correctness, type safety, computability and similar concerns related to computations and their results.
- *Para-functional* [1] constraints – end-to-end timeliness, recovery from faults, memory footprint, security, concurrency, and similar concerns that fall outside the functional semantics of the system, but are nonetheless essential to its correct operation.

Not only do these dimensions invite different styles of programming for configuring their semantics correctly, but they often interact in ways that can cause them *interfere* with each other. This paper considers the fundamental problem of interference in systems programming, examines several existing approaches to address that problem, and proposes a solution architecture that both synthesizes and extends current approaches.

This paper is structured as follows. Section B presents the problem of interference, which is the motivation for this work. Section C examines the challenge of supporting appropriate abstractions in the face of an overall constraint structure that resists decomposition along a single dimension of abstraction. Section D examines current approaches to addressing interference between system aspects, and notes key related research problems that remain open today. To address the open problems described in Section D, Section E presents an integrated architectural vision that combines and augments the existing approaches of model-integrated computing, component middleware, and middleware aspect frameworks. Finally, Section F presents conclusions and describes future work.

B. MOTIVATION: INTERFERENCE

This section describes the problem of interference between system aspects, and gives examples of interference within the context of concurrent distributed component middleware. Section B.a first describes a simple motivating example in which such interference can occur. Section B.b explains in detail how functional and para-functional system aspects can interfere in the example given in Section B.a. Section B.c then discusses the more general subject of interference, and the need for further research in that area.

[1] This work was supported in part by the DARPA PCES program, under contract F33615-00-C-3048 to Boeing and contract F33615-03-C-4110 to Washington University.

a. A Motivating Example

Consider the class of systems consisting of interacting application *components* distributed across multiple embedded endsystems. In this paper the term "component" is used in its formal technical sense, *i.e.*, consisting of objects implementing specified component interfaces, *e.g.*, as defined by the CORBA Component Model (CCM) [2] or J2EE [3] standards.

For these kinds of embedded applications, standards-based middleware such as CCM or J2EE allows application-specific components to be plugged into more general middleware frameworks, allowing application-specific configuration and re-use of independently developed software components. Examples of applications that can benefit from this approach include industrial control, avionics mission computing, and automotive information systems.

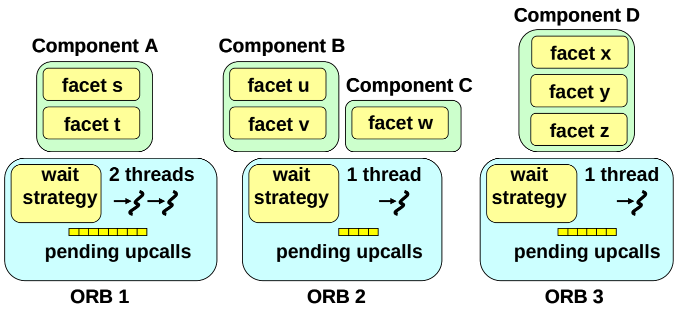


Figure 1: Deployment of Application Components

Figure 1 illustrates many key features of component-oriented embedded systems, including:

- *Components* – compose interfaces with objects that implement them;
- *Component Interfaces* – formally specify interactions between components:
    - *Facets* – advertise method invocation entry points for a component,
    - *Receptacles* – advertise points where invocations are made into other components,
    - *Event sinks* – advertise event push entry points for a component, and
    - *Event sources* – advertise event pushes into other components;
- *Supporting Infrastructure* – enables interactions between components, including:
    - Distributed communication – *e.g.*, between components on different object request brokers (ORBs), and
    - Local concurrency – *e.g.*, the number of threads available and the upcall concurrency policies in an ORB.

Use of component middleware technologies can significantly reduce the complexity of packaging, assembling, and deploying application components, but unfortunately are mainly focused on the *functional* properties of the components, *i.e.*, their implementation and interfaces but not how quality of service (QoS) requirements are communicated to or enforced in the underlying middleware. For embedded systems with additional *para-functional* QoS constraints such as timeliness and distribution, QoS-aware component models that are aware of and provide programming and configuration mechanisms to address these constraints are needed [4].

However, even with the support of QoS-aware middleware, the process of programming and configuring both functional and para-functional properties remains tedious and error-prone. The fundamental problem is that functional and para-functional properties can *interfere* in subtle and complex ways that are often insufficiently represented, checked, or corrected in the overall system programming model. We now consider an example of interference between the inherent *functional* properties of the example application shown in Figure 1, and induced *para-functional* properties resulting from its deployment.

b. Interference Leading to Deadlock

For the example application shown in Figure 1, *functional* properties of interest are captured by a graph of invocations of facets between components. Independent of how the components are deployed on the available endsystems, the invocation graph encodes a causal ordering of the component entry points. The *para-functional* properties of interest in this example are captured by the grouping of components onto ORBs, the number of threads on each ORB, and the strategy used by each thread to wait on connections.

As we have examined in other work, using a non-blocking wait-on-connection strategy has implications for feasible schedulability of invocations, while using a blocking strategy runs a risk of deadlock [5]. We have also examined the ability of adapting rates of invocation at different points in the component packaging, assembly, and deployment lifecycle, to support resource utilization optimizations, *i.e.*, to achieve feasibility or to reduce pessimism [6].

To illustrate the more general problem of interference, we focus here on the problem of deadlock, without reference to rates, priorities, or schedulability. Specifically, we assume that an ORB thread blocks on a connection when it invokes a facet on a component on another ORB. Note that when a facet is invoked on another component in the same ORB, or even a facet within the same component, the thread of execution moves from the invoking method to the invoked method, crossing the component interface. In contrast, when a method invokes a facet on a component on another ORB, two relevant concurrency events occur:

- The thread in the method making the invocation *blocks* until the invocation completes.
- A thread in the ORB hosting the invoked facet is *bound* to the invocation.

We note that for a synchronous two-way invocation, completion does not occur until the reply message from the invoked method has been received; for an asynchronous one-way or AMI invocation, completion of the invocation occurs after the request message has been sent and any callback handlers or other post-processing mechanisms have been registered locally. With two way invocations, a case of *interference* between functional and para-functional properties emerges from this example:

- To process invocations originating from another ORB, a thread from the available ORB threads must be bound to that invocation upcall.
- If a two-way invocation of a facet on another ORB is made within the original upcall, or transitively within another method invoked within the scope of that upcall, the bound thread blocks until the remote invocation completes.
- If all threads in an ORB are blocked, no further upcalls can occur until at least one of the threads completes its original upcall and is then unbound.
- Therefore, any path in the invocation graph that *crosses into* a particular ORB more times than the number of threads in that ORB, will lead to deadlock.

Figure 2 illustrates a scenario that explores cases where deadlock does or does not occur, in a hypothetical invocation graph based on the example in Figure 1:

1. An invocation of facet s arrives at ORB 1, and binds a thread.
2. The method implementing facet s makes a blocking invocation of facet u, which binds the thread in ORB 2.
3. The method implementing facet u invokes facet v in the same thread.
4. The method implementing facet u makes a blocking invocation of facet t, which binds the second thread of ORB 1, but then returns without making further invocations which in turn unbinds the second thread in ORB 1 and unblocks the thread in ORB 2. The method implementing facet u then also returns without making further invocations, which unbinds the thread in ORB 2 and unblocks the first thread in ORB 1.
5. The method implementing facet s makes a blocking invocation of facet x, which binds the thread in ORB 3.
6. The method implementing facet x makes a blocking invocation of facet w, which binds a thread in ORB 2 – note that the same thread was bound earlier to the invocation of facet u, but was unbound upon return of the method implementing facet u.
7. The method implementing facet w makes a blocking invocation of facet y, but the only thread in ORB 3 is already blocked on the invocation from the method implementing facet x, to facet w. At this point, ORB 3 is thus deadlocked, as is the entire invocation chain through facets s, x, and w.

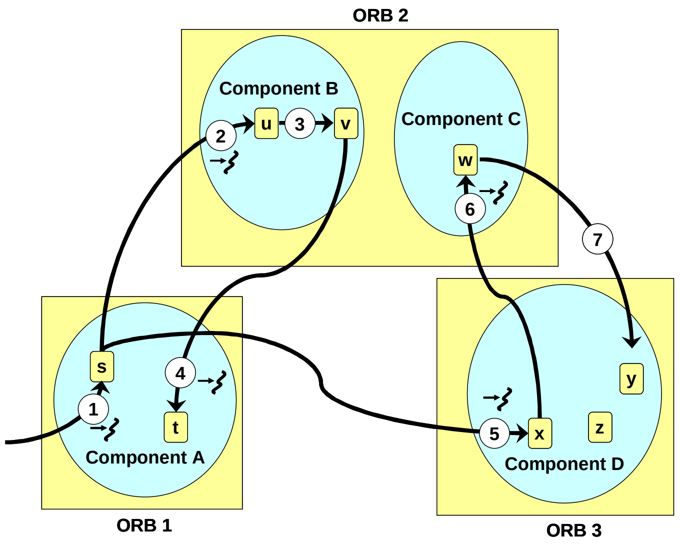


Figure 2: Interference Resulting in Deadlock

c. *Toward more Complete Models of Interference*

Numerous hazards other than deadlock must be avoided in complex embedded systems, such as failures of end-to-end timeliness, hardware or software faults, or even adversarial intrusion and compromise of the system. For component-based real-time systems, temporal patterns of facet invocations and execution times of implementing methods must be modeled. The behavior of the supporting middleware must again be considered, *e.g.*, preemption policies for access to resources and the overhead from context switches and other "hidden" behaviors of the underlying middleware and operating system.

Interference due to fault-tolerance mechanisms and policies must be modeled, and measures to mitigate or repair faults must be modeled and their benefits must be balanced against their impact on other constraints, such as timeliness. For example, replication of data and invocation messages consumes computation and communication resources, but leads to more timely recovery in the event a fault occurs. A similar kind of interference between security measures and other properties is illustrated by the practice of adding watermarks to application data: computation and storage resources are consumed to increase confidence in the authenticity of information managed by the system.

C. The Problem of Appropriate Abstraction

A fundamental problem in modeling constraints in multiple dimensions such as functionality, timeliness, fault-tolerance, and security, is to provide appropriately scaled abstractions within the programming models used to develop and configure the system. Problems can arise both from excessive abstraction and from insufficient abstraction.

Excessive abstraction can result in mismatches with crucial constraints of an application. For example, if the number of threads provided by an ORB is not configurable in the programming model, hazards such as deadlock and deadline failure may result. The scenario examined in Section B.b illustrates this problem, where an application whose functional properties result in a invocation chain that crosses into an ORB more times than the number of threads it has available to perform upcalls will result in deadlock.

Changing the policy for waiting on connections to be non-blocking can avoid deadlock, but may have implications for deadline feasibility due to increased blocking times [5], and may also increase overhead by making more calls to operating system functions such as `select()`. Simply increasing the default number of threads in each ORB would alleviate the deadlock problem for some applications, but one can envision that some applications might still exhaust a given fixed limit on the upcalls a single ORB may perform due to cyclic or recursive structure in their invocation graphs. Furthermore, increasing the number of threads may increase operating system and middleware overheads, resulting in significant performance degradation.

Insufficient abstraction leads to another kind of problem, in which the details needed to configure a system are exposed in the programming model, but the space of configurations is unmanageably large, and the job of producing correct systems is thus made tedious and error-prone. This kind of *accidental complexity* has been addressed by frameworks like the ADAPTIVE Communication Environment (ACE) [7], which capture key higher level abstractions that are obfuscated by lower-level interfaces like POSIX [8].

The heterogeneity of concerns in real-world systems makes it implausible that any single configuration of properties in the supporting middleware and operating systems, when composed with the functional semantics of each application, will meet the para-functional constraints of every system. A key part of this dilemma has been called "the tyranny of the *dominant decomposition*" [9], in which a particular style of abstraction is appropriate for most of a system of constraints, but does not address others completely.

For example, ACE encodes an object-oriented decomposition of the configuration space inherent in POSIX and similar operating system interfaces such as Win32. ACE selectively uncovers certain low-level details such as file handles, which need to be shared among objects or passed to operating system interfaces, avoiding problems of either too much or too little encapsulation.

Even so, limitations of a purely object-oriented style of abstraction begin to appear when composing multiple ACE objects to create higher-level middleware subsystems such as dynamic scheduling event dispatchers [10]. In particular, information needed by different scheduling strategies may vary significantly, and using inheritance polymorphism to associate combinations of scheduling parameters with the events to be dispatched would lead to an explosion in the number of classes needed and result in an *increase* in the complexity of programming the infrastructure. Instead, applying a *generic programming* [11] style of abstraction to complement the predominantly object-oriented structure of ACE allows arbitrary scheduling parameter types to be composed with events, while preserving type safety of QoS parameters with respect to the scheduling heuristics used in the dispatcher.

Objects, distributed objects, components, and generics all treat a single type and its interfaces as the fundamental unit of encapsulation, achieving a better fit between system requirements and the abstractions available to satisfy those requirements. In real-world systems, however, key concerns still cross-cut such single-type encapsulation boundaries leading to additional complexity in meeting their associated requirements.

The Aspect-Oriented Programming (AOP) [12] paradigm encapsulates sets of related points that cross-cut other abstraction boundaries, and thus serves to complete an appropriate set of abstractions when used in conjunction with other programming paradigms. The combination of multiple styles of abstraction avoids the problem of a single dominant decomposition, and leads to the solution approach advocated by this paper: a synthesis of model-integrated computing tools, component middleware, and lower-level aspect frameworks to organize the application and supporting infrastructure.

D. Towards Generative System Programming

This section surveys existing approaches to addressing interference between system aspects, and identifies the benefits and limitations of the current state of the art. Section D.a first describes several existing approaches to addressing interference between system aspects. Section D.b then details open research questions that these approaches do not address.

a. Survey of Current Approaches

We first survey existing approaches to managing interference between system aspects, which fall into three main categories:

- System Aspect Frameworks – system infrastructure frameworks that directly encode abstractions for managing interference.
- Model-Driven Toolsets – integrated combinations of modeling abstractions and infrastructure generation or configuration abstractions, often focused on a particular problem domain.
- Model Engineering Tools – more general abstractions for model creation, manipulation, and checking, which can be applied variously to infrastructure generation, configuration, modeling, and checking for different domains.

System Aspect Frameworks: BBN technologies *Qoskets* [13] are high-level aspect-oriented middleware abstractions for QoS management. Qoskets cross-cut distribution and layer boundaries, and thus offer end-to-end configurability of large-scale system infrastructures.

The CIAO [4] project at Washington University and Vanderbilt University extends the standard CORBA Component Model for configurability of para-functional as well as functional properties. CIAO extends the standard CCM XML-based component packaging, assembly, and deployment capabilities to include configuration of para-functional system aspects within the components, their containers, and the supporting system infrastructure. We are working with researchers at BBN to integrate their Qoskets approach with CIAO.

Researchers at the University of British Columbia have developed an AOP tool called AspectC, for C systems programming environments [14]. Complementing the AspectJ tool for Java, AspectC offers the ability to refactor many examples of open-source systems software along aspect-oriented paths. The extension of AOP tools to multiple languages is very promising and availability of such tools for C++, including the ability to weave aspects into template code, would allow new and powerful combinations of generic, object-oriented, and aspect-oriented techniques to be applied together.

Absent the availability of mature AOP tools for C++, we have tended to combine generic and object-oriented techniques with logic-driven composition approaches in the vein of work at the University of Utah on Task/Scheduler Logic [15]. In particular, we are in the process of re-factoring the Kokyu [10] dispatching framework to use both generic and object-oriented techniques in conjunction with composition logics to ensure safety of configurations with respect to para-functional properties.

The Time Weaver [1] framework developed at Carnegie-Mellon University provides system aspect configuration abstractions called *couplers* that are similar to the BBN Qoskets approach. The CMU approach explicitly promotes recursive composition of couplers for hierarchical encapsulation of abstractions, and notes several design dimensions along which para-functional aspects are composed in existing DRE systems. Of particular note in the CMU approach is the discussion of *projections* of aspects between different design dimensions, and the need to reconcile constraints, *i.e.*, to address *interference* across as well as along those design dimensions.

The CoSMIC [16] project at Vanderbilt University has similar aims to the Time Weaver project at CMU, but pursues model-driven configuration of DRE systems within the context of existing middleware frameworks, notably CIAO and QuO. By adopting both the component-oriented programming model in CIAO and the aspect-oriented programming model in the QuO

Qoskets approach, CoSMIC can leverage the appropriate style of abstraction for each of a broader set of concerns.

Model-Driven Toolsets: The Cadena [17] project at Kansas State University applies model-based techniques to configuration of component-based systems, and in particular to behavioral aspects of component-based applications and their supporting infrastructure. Cadena thus covers a wide domain of component-based applications, while allowing selective customization of the aspects relevant to each particular application. We are in the process of exploring integration of Cadena with CIAO, leveraging CIAO's ability to configure QoS properties directly within Cadena.

The VEST [18] toolkit developed at the University of Virginia is another model-driven toolset for DRE systems. Where Cadena focuses on configuring a particular kind of middleware, VEST takes a more vertically crosscutting approach, modeling, configuring and checking abstractions at the application, middleware, operating system, and even hardware levels. Both the Cadena and VEST implementations are focused on particular domains, but each is extensible to cover additional abstractions beyond those in its current implementation.

Model Engineering Tools: Whereas Cadena is focused on a particular domain, the Bogor [19] framework that is also being developed at Kansas State University provides flexible and general capabilities for developing a variety of model-checking tools. The generic modeling environment (GME) [20] tool developed at Vanderbilt University is a similarly general meta-modeling environment, and in fact was used in the VEST tool implementation.

b. Open Research Challenges for Generative System Programming

Each of the approaches surveyed above covers an important segment of the space of model-driven computing, but it is apparent that no one of those approaches can completely cover the configuration space of functional and para-functional properties in DRE systems. Therefore we argue that a synthesis of techniques is needed, to allow DRE system developers to draw on multiple tools and infrastructure frameworks and apply each to its best use, with reasonable assurance of (1) the fidelity of the abstract representations to the composed system, and (2) the correctness of the system's properties in each of its multiple design dimensions.

The following challenges must be addressed to achieve such a synthesis:

- **Implicit, ad hoc structure** of implementation frameworks must be re-factored to avoid unnecessary forms of interference, and must provide explicit reflective information for use in their composition and configuration.
- **Alternative dominant decompositions** must be supported fully in the programming model, to reduce complexity of design dimensions.
- **Higher levels of abstraction** should be used in projecting model abstractions into the implementation frameworks used to compose the model-based system.

E. AN ARCHITECTURAL VISION

This section describes an architectural vision that seeks to align the different kinds of system component infrastructures, system aspect frameworks, model-driven toolsets, and model engineering tools described in Section D.a, to address the challenges outlined in Section D.b. A key theme of this vision is that both (1) top-down modeling of application characteristics and requirements, and (2) bottom-up modeling of infrastructure aspects are necessary and appropriate. Section E.a first gives an overview of the proposed architecture. Section E.b then illustrates how segments of that architecture could serve to resolve the deadlock problem discussed in Section B.b.

a. Architecture Overview

Figure 3 illustrates the proposed architecture, which integrates *model-driven, component-oriented, and aspect-oriented* approaches.

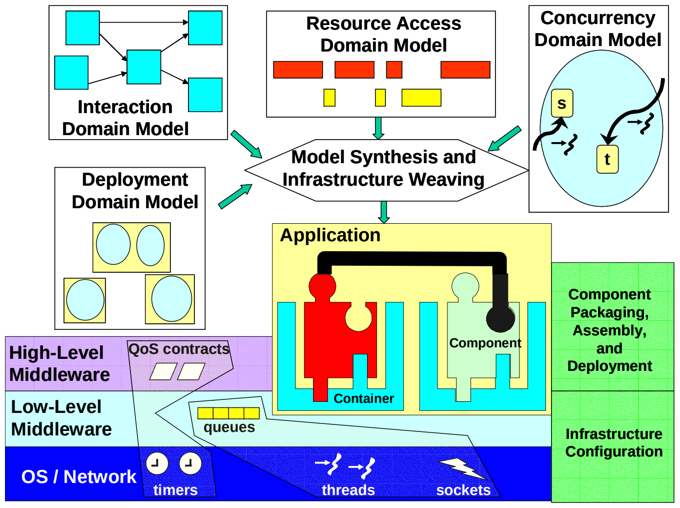


Figure 3: Model-Component-Aspect System Architecture

The following list describes the major segments of the envisioned *model-component-aspect* architecture, and the respective contribution of each segment:

- **Domain-Specific Modeling Tools** – increase scalability by encapsulating representation and analysis of individual domain models, increasing separation of modeling and checking concerns.
- **Model Synthesis Tools** – allow synthesis and checking of multiple separate domain models, a necessary adjunct to the separation of models for distinct domains.
- **Component middleware** – offers a common and standardized implementation context within which reflective information for different domain models can be applied directly to weaving and configuration of application components and supporting system infrastructure.
- **High-level system aspect frameworks** – support *integration and configuration* of existing middleware infrastructure and application implementations using their *external* interfaces.
- **Low-level system aspect frameworks** – allow *synthesis and customization* of infrastructure and components by weaving *into* those abstractions.
- **Implementation Weavers** – perform model-driven system generation, through synthesis, integration and configuration of application components and system infrastructure.

The relationships between these architectural segments are as important as the segments themselves. Application components must be able to provide model information about their para-functional constraints such as memory usage and thread safety, as well as their functional constraints, to modeling tools. Middleware infrastructure implementations must similarly provide model information about their configuration options and para-functional properties. High-level system aspect frameworks can glue together integrated configurations of existing middleware implementations [13]. Low-level system aspect frameworks are promising complements to model engineering tools, so that a domain-specific model, *e.g.*, for real-time analysis, can be co-designed with model information about specific system framework abstractions, *e.g.*, threads, timers, and queues in a dispatching subsystem [10], from which its model implementations will be generated. Component models' packaging, assembly, and deployment capabilities can be used, possibly in conjunction with domain-specific infrastructure configurators, as implementation weavers.

b. Example Resolved

We now offer a brief illustration of how various segments of a *model-component-aspect* architecture could be applied to resolve the problems illustrated in Section B.b. In the deployment scenario shown in Figure 2, the following steps could help avoid deadlock while still meeting other functional and para-functional constraints:

1. Component middleware would provide reflective information about the packaging, assembly, and deployment of components, including the invocation graph and placement of components onto ORBs.
2. Low-level system aspect frameworks would provide reflective information about default concurrency and reply-wait strategies and other properties, and offer points of configurability of those properties.
3. High-level system aspect frameworks would insert instrumentation points for collecting run-time information, such as exhaustion of a thread pool or deadline misses. Adaptation mechanisms may also be added to allow run-time manipulation of system properties, in cases where some hazards are uncommon and recovery from them is possible if encountered.
4. Domain-specific modeling tools would capture models for the system's interaction, resource access, deployment, and concurrency domains, using the reflective information supplied by the components and the aspect frameworks.
5. Model synthesis tools would integrate those models, and perform crosscutting analysis to determine ways in which the deadlock could be resolved, *e.g.*, by migrating Component C from ORB 2 to ORB 3, or by increasing the number of threads configured in ORB 3. If multiple alternatives were feasible, additional analysis would be performed to determine which would be more desirable, and under what operating conditions.
6. Finally, implementation weavers would be used to configure or even synthesize properties of the implementation – in cases where multiple configurations are possible and among them different ones would best suit different operating conditions, adaptation mechanisms would be configured to detect those conditions and adapt among the alternative configurations accordingly.

F. CONCLUSIONS AND FUTURE WORK

This paper has presented the position that interference between system aspects, in both the functional and para-functional semantics of an application, is a fundamental issue in real-world systems programming. Furthermore, it is apparent that no single dominant decomposition can address the problem of entanglement. Instead, appropriate abstractions in each of several domains must be composed and woven together in both the modeling and system generation segments of a larger integrated architecture. Model-component-aspect architectures are proposed to achieve the composition of abstractions from multiple domains, and generate DRE systems that are correct in both their functional and para-functional properties.

The challenges outlined in Section D.b offer a roadmap for future work. Of particular interest is the co-design of model engineering tools with abstraction frameworks, such as low-level aspect frameworks, that can reduce the complexity of model specification and checking, and allow more rapid development of additional domain-specific modeling tools. A further area of interest is the representations that would arise from or perhaps even precede the co-design of general model engineering tools and their abstraction libraries. In general the goal is to increase the level of abstraction within the models, and correspondingly within the implementation frameworks used to realize those models. Algebras and logics for interference-aware composition of system aspects is of particular interest, both for co-design of aspect frameworks and model engineering tools, and for customization of domain-specific models.

References

[1] R. Rajkumar and D. de Niz, “Model-Based Embedded Real-Time Software Development”, RTAS 2003 Workshop on Model Driven Embedded Systems, Washington, D.C., USA, May 2003.

[2] Object Management Group, “CORBA Components”, OMG document formal/2002-06-65, June 2002

[3] D. Alur, J. Crupi, and D. Malks, “Core J2EE Patterns: Best Practices and Design Strategies”, Prentice Hall, 2001.

[4] N. Wang, D. C. Schmidt, A. Gokhale, C. D. Gill, B. Natarajan, C. Rodrigues, J. P. Loyall and R. E. Schantz, "Total Quality of Service Provisioning in Middleware and Applications," *Microprocessors and Microsystems, special issue on Middleware Solutions for QoS-enabled Multimedia Provisioning over the Internet*, vol. 27, no. 2, pp. 45-54, March 2003.

[5] V. Subramonian and C. Gill, “A Generative Programming Framework for Adaptive Middleware”, HICSS 2004, Hilo, HI, January 2004 (to appear).

[6] N. Wang and C. Gill, “Improving Real-Time System Configuration via a QoS-aware CORBA Component Model”, HICSS 2004, Hilo, HI, January 2004 (to appear).

[7] D. C. Schmidt, “The ADAPTIVE Communication Environment (ACE), www.cs.wustl.edu/~schmidt/ACE.html, 1997

[8] IEEE, POSIX1003.1c, “Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) [C Language]”, 1995

[9] Tarr, Harrison, Ossher, Finkelstein, Nuseibeh, and Perry, “Workshop on Multi-Dimensional Separation of Concerns in Software Engineering”, ICSE 2000, Limerick, Ireland

[10] C. Gill, D. Schmidt, and R. Cytron, “Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing”, IEEE Proceedings 91(1), Jan 2003.

[11] Matthew H. Austern, “Generic Programming and the STL: Using and Extending the C++ Standard Template Library”, Addison-Wesley, Reading, Massachusetts, 1998

[12] Kiczales, Lamping, Mendhekar, Maeda, Lopes, Loingtier, and Irwin, “Aspect-Oriented Programming”, Proceedings of the 11th European Conference on Object-Oriented Programming, June, 1997

[13] Schantz, Loyall, Atighetchi, and Pal, “Packaging Quality of Service Control Behaviors for Reuse”, Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-time distributed Computing, Washington, DC, April, 2002.

[14] Coady, Kiczales, Feeley, and Smolyn, “Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code”, Joint ESEC / ACM SIGSOFT FSE-9 Conference, September 2001.

[15] Alastair Reid and John Regehr, “Task/Scheduler Logic: Reasoning about Concurrency in Component-Based Systems Software”, 2002, citeseer.nj.nec.com/reid02taskscheduler.html

[16] Gokhale, Natarajan, Schmidt, Nechypurenko, Gray, Wang, Neema, Bapty, and Parsons, “CoSMIC: An MDA Generative Tool for Distributed Real-time and Embdedded Component Middleware and Applications”, ACM OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture, Seattle, WA, November, 2002

[17] Hatcliff, Deng, Dwyer, Jung, and Prasad, “Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems”, Proceedings of the International Conference on Software Engineering, Portland, OR, May 2003.

[18] Stankovic, Zhu, Poornalingam, Lu, Yu, Humphrey, and Ellis, “VEST: An Aspect-based Composition Tool for Real-time Systems”, 9th IEEE Real-time Applications Symposium, Washington, DC, May 2003.

[19] Robby, M. B. Dwyer, and J. Hatcliff, “Bogor: An Extensible and Highly-Modular Model Checking Framework”, Proceedings of the Fourth Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003), to appear.

[20] Ledeczi, Bakay, Maroti, Volgysei, Nordstrom, Sprinkle, Karsai, “Composing Domain-Specific Design Environments”, IEEE Computer, November, 2001.

From Natural Language to Linear Temporal Logic: Difficulties of Capturing Natural Language Specifications in Formal Languages for Automatic Analysis

Elsa L. Gunter*
New Jersey Institute of Technology

Abstract

To be able to apply tools to verify properties of any system it is necessary to have a formal model of that system, and a formal expression of the properties to be verified. Linear Temporal Logic [5] is a common formalism for expressing properties of systems of concurrent communicating processes, as found for example in embedded systems. To verify these formulae (or attempt to derive test suites from them) for a system, the relevant processes are modeled as automata, and algorithms are applied to the pair of the formulae and the automata to find paths in the automata that satisfy certain properties pertaining to the given specification. However, specifications begin as informal natural language concepts (and maybe documents). It is generally these informal specifications that express the true and fullest system requirements. Therefore, it is necessary to recognize the gap between the informal and the formal specifications. In this paper we concentrate on the gaps that exist between informal specification and formal specifications given in Linear Temporal Logic, and some possibilities for controlling that gap. We also discuss the impact these methods of handling the gap have on the system model.

1 Introduction

In the ideal world, a software programming project would begin with a user's requirements document, which would be expressed in terms that were meaningful to the user and which would form the agreement between the user (or consumer) of the product and the producer [6]. System specifications would then be created by refining the requirements to a description of just the behavior of the system, using (and hopefully documenting) knowledge of the environment in which the system would be deployed to factor out the aspects that rely on phenomena that are not directly input to or output by the system. From here the specification would be decomposed into functional components to provide a framework for the design of the software. Each functional component would be refined to an operational model, possibly such as a finite state machine, and from here coding could begin. After the software has been developed, the specification again would be needed, this time to guide system testing.

Requirements and specifications begin their existence as ideas expressed in natural language. Before automated tools may be applied to the specification, it must be captured in some formal language processable by computers. If an operational model is to be demonstrated to satisfy the specification, then the language capturing the specification must also have a rigorous semantics. However, natural language is highly expressive, and highly ambiguous. Formal languages having a rigorous semantics and admitting automated checking of properties are of limited expressive power. Therefore, just as there is a gap between the implementation and the operational model,

*This research was supported in part by ARO DAAD19-01-1-0473.

and between the operational model and the specification, there is a gap between the informal, natural language specification and the formal specification.

Since the specification is the foundation of the software construction, it needs to satisfy various sanity criteria by itself. In addition, it must be possible to confirm that the composition of the operational models for the system components together with the necessary domain knowledge satisfy the specification, and that the individual code units implement the corresponding operational models. At each step of this confirmation process there are concerns that must be taken into consideration. Efforts to derive code automatically from operational models attempts to address the problem of confirming that individual code units satisfy the operational models. Program and temporal logics, and model checking are a very active area of research for demonstrating that the operational models satisfy formal specifications. In this paper we wish to draw attention to the step of capturing the specifications in a formal language suited to automatic analysis and use in the subsequent stages of development and validation.

2 Examining Natural Language Specifications

As mentioned above, specifications begin there existence in human thought, which is typically not expressed as logical formulae, automata, or mathematical expressions, but rather in natural language. Natural language is at the same time on the one hand imprecise and ambiguous, and on the other, highly expressive. In rendering a natural language specification in a formal language, both aspects are hurdles to be overcome.

Ambiguities are removed by a careful reading of the natural language document by a reader who is prepared to to challenge the precision of every statement, both individually and a component of the whole document. The reader is greatly assisted in this process, if to aid their critical reading, they are attempting to formalize the document as literally as possible in a language with a clear and precise semantics. Eliminating ambiguities is done in an iterative process of recognizing the ambiguity, seeking clarification from the author or authority, attempting to formalize the clarified version, then uncovering new ambiguities which need clarification. This process is inherently a human one; on one end the basic understanding of the meaning of the specification resides in the minds of its authors, and on the other, the recognition of the ambiguity is done by the human performing the translation. With time, the act of recognizing ambiguities may become increasingly assisted by machine translators. However, by making use of a translator, in effect, at the point we submit the "natural language" specification to the translator, the language becomes a machine language and the question of whether the natural language expressions have the same meaning as the meaning given by the translator still remains.

Because of the human factor, both the act of seeking clarification and rendering clarification are subject to non-determinism. On the side of seeking clarification, the reader of the specification may read a statement for which only one meaning occurs to them. However, there may have been a different meaning, possibly the one intended by the author. On the other side, the author, when asked for a clarification, may not understand the differences in the possible interpretations, and may choose a possibility different from the one they truly meant.

As an example of the need to recognize and eliminate ambiguities, let us consider a sample statement that might be found in a user requirements (or system specification) document of a system that must monitor and regulate a device that includes a pressure reading:

> Once the pump has been turned been on, an initial pressure reading will be taken and displayed, and thereafter the displayed value for the pressure will be updated every 2 seconds.

At first reading, this may seem like quite a precise statement. Let's look at it a bit. What does it mean for the displayed value to be *updated*? The first answer is that the displayed value is made to be the same as the actual pressure value. But since we are talking about time, are we insisting that the displayed value is made to be the same as the actual value at exactly the same time? Are we insisting that there is *no time* allowed for the data to be read and the display to be changed? If the underlying notion of time is course enough, and our sensors and processors are fast enough, then this may be reasonable. However, this is really just sweeping the issue of time for computation under the rug and assuming that the machine is fast enough to meet whatever the real unwritten requirements are. Thus "updated" really means something more like that the display value is the same as the actual pressure value of not more than some allowed amount of time ago. As the reader has probably also noticed, we can't realistically require the display value to be "the same" as the actual pressure value. We can't measure the actual pressure value, we can only measure an approximate value for it. Another question is what does it mean for the display to be updated *every 2 seconds*? Does it mean that the reading upon which the display is based should be taken exactly every 2 seconds, or that the new display value would be generated exactly every 2 seconds (or perhaps they really wanted to know that the displayed valued was never more than 2 seconds old, which isn't really the same as updating the display every 2 seconds). If the time to read the pressure value and update the display is constant, then the first two options above amount to the same thing. However, if for example, the time to read the pressure is variable, then they are not. And lastly (for now), what should happen if the system fails to be able to update the displayed value at the appropriate time? Saying that this should be impossible means claiming there will never be any faults in the environment or with the system interface to the environment.

The points made above about the given example are relevant regardless of the choice of logic in which to model the specification. However, attempting to formalize this statement in various frameworks would tend to show up at least some of these issues. For example, if we tried to capture the specification with an extended finite state machine, we would likely end up with a guarded transition labeled by something like:

$$\begin{array}{l}\mathsf{seconds}(\mathsf{current\_time}\ - \mathsf{last\_update\_time}) \geq 2 \Rightarrow \\ \mathsf{display\_value} := \mathsf{pressure\_value}\end{array}$$

If the model is interpreted in a setting where, for each transition, in addition to the actions committed by the machine, the environment is allowed to arbitrarily change all values it supplies (e.g. [7]), then, with each transition, pressure_value potentially will change its value. Therefore, we can not guarantee that there is ever a time when display_value = pressure_value. All we can guarantee is that the current value of display_value is the same as some previous value of pressure_value.

3 From Natural Language to LTL

As discussed above, natural language specifications are subject to considerable ambiguity. However, even if they are rendered with the utmost care and precision, before they can become the grist for the mill of automatic analysis, they must be faithfully translated into a machine-processable formal language, with a rigorous semantics. Moreover, that formal language must admit the kinds of automated analysis desired.

Linear Temporal Logic (LTL) is a common language used for specifying properties that are to hold of concurrent systems. One advantage of LTL is that it has a fairly simple and intuitive syntax with a simple and well-defined semantics. However, the driving motivation behind the use of LTL is that it is possible to effectively check whether a given LTL formula holds of a given system expressed as an automaton. However, these factors are at odds with LTL being a highly expressive language. LTL has a limited ability to express a set of states within a sequence where a property is to hold. Perhaps an even bigger impediment in practice, at least by anecdotal evidence, is the inability in LTL to express desired relations between state components at different points in time.

In LTL, time is not explicit, and thus, if it is necessary to talk about time, this must be done by making time an explicit component of the states tested by the base sub-formulae in the LTL formula. Then time is also added explicitly to any model of the system being checked. (As an example, see [1].) As an example, consider the requirement given earlier concerning the monitoring of the pump pressure. To be able to rendered the clause "updated every 2 seconds" in LTL, requires a variable to monitor the passage of time so that we know when two seconds has passed. However, there are properties of time to which any system model must conform. For example, time can not decrease. (If it could, most implementations of the pump monitor would not provably satisfy the update requirement.) These properties cannot in general be expressed in LTL, so checking that time has been correctly modeled is generally outside the scope of what can be verified by model checking LTL formulae. Just as it is important to document what domain knowledge is used in refining user requirements into a system specification, it is also important to document these extra-logical facts that are required to hold, and upon which the correctness of the system may be depended.

In addition to concepts such as time, which are possibly implicit in the system or inherited from the system environment, but which may need to be made explicit in the model, it is also often necessary to add "history variables" to our model to enable us to express requirements pertaining to past values of components of the state. LTL is a logic for expressing properties of sequences of states. However, the base formulae are only over a single state. There is no ability to express relations between the values of the state at one point in time and another. We can say that the value of a given variable is always 5, but we cannot say the value of a given variable is constant, or monotonically increasing. To be able to capture the idea that the value of x is monotonically increasing, we would introduce an auxiliary variable, say $\mathsf{previous\_x}$ and require that

$$\mathsf{Always}\ \mathsf{x} \geq \mathsf{previous\_x}.$$

This only captures the notion that x is monotonically growing if we have the additional knowledge that every transition updates $\mathsf{previous\_x}$ to the previous value of x. This is a fact that can not be expressed in LTL, so it must be verified in the augmented system model by some means other than LTL model checking.

In concurrent and embedded systems, there is an additional difficulty with the use of the temporal operator Next. To be able to prove properties of a system, it is generally necessary to decompose the system into modules and abstract out those components that do not contribute to the property currently being verified. If specifications are given with the Next operator, they may be checked as true in the subcomponent deemed relevant, but the property may be false in the larger system where all the components are composed. Let us consider the LTL specification

$$\begin{aligned}&\mathsf{Always}\ (\mathsf{seconds}(\mathsf{current\_time} - \mathsf{last\_pressure\_reading\_time}) \geq 2 \Rightarrow\\ &\qquad \mathsf{Next}\ (\mathsf{display\_value} = \mathsf{previous\_pressure\_value})).\end{aligned}$$

In natural language, this tells us that anytime the system sees that it's been at least 2 seconds since the last time the display was updated, then the system must update the display to the current pressure value (what will be the previous pressure value, once we complete the transition doing the update). This specification may very well hold for the subsystem composed only of the system (or environment) clock, the pressure monitor, and the display, but if the system is more complex with other values to be monitored and perhaps controlled, when the other components are put in parallel with the particular subsystem (with an interleaving semantics), it is not automatic that the specification still holds, and indeed, unless there are some kinds of locks preventing other parts of the system from progressing, this specification is likely to be false. At the first point at which $\mathsf{seconds}(\mathsf{current\_time} - \mathsf{last\_pressure\_reading\_time}) \geq 2$, it is possible for some other part of the system to be busy doing something. The problem with Next, as well as other constructs such as Always and Until, is that they are not compositional.

Unfortunately, the most obvious alternative to the use of Next is the use of $\mathsf{Eventually}$. This is generally much weaker than is intended by a natural language specification where Next seems appropriate. If one can identify a property φ that holds of states of the subsystem prior to the transition being taken and that fail to holds as soon as it is taken, then it might be possible to use a formula of the form φ Until ψ in place of Next psi. What we would like to use for φ is a statement that the state of the subsystem doesn't change. However, being able to express this directly requires being able to compare values for variables from one state with those from another state (the "successor" state), and as we have said above, this is outside the expressive power of LTL. A non-solution is to add a variable to our state for each transition (or each state) in our subsystem. For each transition, we add the action of setting the corresponding variable to true, and setting all of the variables representing the other transitions (or out-states) to false. Then one can attempt to capture that the subsystem hasn't changed by enumerating all the transitions (or states) and saying that each variable has an explicit fixed value until the property that is to hold next holds. Something like this might be workable in the setting of system models using only plain finite state machines. But this is generally not a solution because it does not allow us to detect when change has occurred in the subsystem through taking a transition that is a loop. In extended finite state machines, where there is a notion of "program states" given by variable values, in addition to the states of the finite state machine, one may enter the same finite state machine state many times having a different program state each time. Thus the very values that we are interested in not changing may change anyway.

Let us return to the example of the display update. Below is a "clarification" of the original specification given.

> Once the pump has been turned been on, an initial pressure reading will be taken and, if the value read is not an error, it is displayed. Thereafter, provided there are no errors in pressure readings, the display will never be more than 2 seconds old.

We can capture much of the intent of this specification with the following LTL formula:

```
Pump_on ∧¬error(pressure_reading) ⇒
Eventually
(display_value = previous_pressure_reading ∧
 last_display_update_time = previous_current_time ∧
 (error(pressure_reading)
  Releases
  (seconds(current_time − last_display_update_time) ≤ 2
   Until
   (display_value = previous_pressure_reading ∧
    last_display_update_time = previous_current_time))))
```

Some of the assumptions that are made for this to correctly express the desired specification include that we can define error from just the value of pressure_reading, and that last_display_update_time always contains the time when the display was last updated. These requirements are extra-logical and need verifying by means other than LTL model checking.

The difficulties we have mentioned here are not specific to Linear Temporal Logic, but extent to similar, more expressive logics such as CTL* (see [2]) and the μ-Calculus (see [4]). Allowing branching time constructs (along some path, or along all possible paths) does not address the issues raised here. The additional expressiveness of the μ-Calculus does allow greater ability to describe a set of states for which a proposition is to hold. However, in all these systems, there is still a lack of ability to relate values from different points in time, and a lack of compositionality.

4 Reflecting Assumptions in the System Model

In the previous section we discussed methodology for how we can use LTL formulae to indirectly capture the meaning of natural language specifications. These methods have limitations and come at an expense of imposing restrictions on the system models to be checked. These restrictions often involve the addition of new variables and alterations to the set of transitions to reflect the needed behavior of these variables. For example, when adding history variables, there is the extra-logical (i.e., not expressible in Linear Temporal Logic) requirement that the history values record the appropriate values at the appropriate times, and never take on any irrelevant values. Not only must these extra variables appear in the LTL specifications, but they must also occur throughout the system model to be verified with respect to the LTL specification, in such a way as to guarantee the extra-logical requirements. Cluttering the system model by modifying it everywhere needed with this additional information that is not explicitly present in the actual implementation reduces the trustworthiness of the verification, and the increases the gap between the system model and the implementation.

We propose that, where possible, the system model be left intact, and the model that is verified be an automatically generated product of the original system model together with a separate model capturing the added variables and transitions. Augmenting the model with history variables, for example seem a prime candidate for such automation. For some notions of automata and products, deriving the model needed for verification from the initial system model via product may not be an appropriate thing to do. However, for extended finite state machines (EFSM) with labeled transitions, this is possible. Briefly, an extended finite state machine with labeled transitions is a finite state machine where the edges are labeled from a given alphabet, but where we extend with a function mapping letters of the alphabet to guarded actions over a fixed set of variables. In this setting, if the original system specification is deterministic to the extent that no edges from the same start have the same label, then if we take the product of the system EFSM with an EFSM with the same labeled finite state machine, but extended by different guarded actions, the result will yield an extended finite state machine that has the same underlying finite state machine, but where the guarded actions are now the pairwise composition of the guarded actions from the two machines.

Let us see what this means in the example with the pressure display. Possible (pseudo-) code for the pressure display is as follows:

```
if Pump_on then if not(pressure_value = error_value)
then {display_value := pressure_value;
      last_display_time_update := current_time;}
else pressure_alarm := true;
while (pressure_alarm = false)  do
{if not(pressure_value = error_value)
 then (if seconds (current_time - last_display_time_update) >= 1.8
       then {display_value := pressure_value;
             last_display_time_update := current_time;})
 else pressure_alarm := true;}
```

For reasons of space, we omit a detailed description of the EFSM that captures this process. Briefly, the code can be compiled into a control flow graph [3] and then automatically translated into an EFSM with ten states. Each state corresponds to either a conditional or an assignment. All transitions are labeled distinctly. Each conditional state has two out transitions, each with a guard but no real action, and each assignment state has one out transition with no guard, but an action corresponding to the assignment. A more compact EFSM can undoubtedly be created, but there is a simple algorithm to automatically generate this one. To add the information about the previous_ variables, we create a new EFSM with exactly the same states, edges, and transition labels, but now every transition has the same unguarded action associated with it:

$$\begin{array}{l}\mathsf{previous\_pressure\_reading} := \mathsf{pressure\_reading};\\ \mathsf{previous\_current\_time} := \mathsf{current\_time}\end{array}$$

If we had more previous_ variables to be kept track of we would add them in. The product of this EFSM with the system EFSM yields an EFSM with the same states, edges, and labels, but where the action component of the guarded action associated with each transition is augmented by the additional assignments. By decomposing the problem this way, we separate out the task of of verifying the "extra-logical" properties, so that they only need to be shown for the EFSMs that were constructed solely to make them true. For the example of the history variables, by this factorization, it is immediate that for each transition the value of each previous_ variable is the same after the transition as the value of corresponding original variable was before the transition. Moreover, by this factorization, we don't have to worry about having accidentally changed the system model in some unpredictable way since the system model makes no mention of the previous_ variable and the history EFSM effects only them. At present, what is proposed here is only methodology. However, it seems clear that at least in some limited but common cases, this process of factoring and constructing auxiliary EFSMs, such as the history EFSMs, could be automated.

5 Conclusions

In this paper we have discussed the transition from an informal natural language specification to a formal specification in Linear Temporal Logic and the construction of a model suited to the specification, but still capturing the system design so that the system design can be verified to have the desired properties. To formalize informal specifications, there is an iterative process of clarification and rendering the specification more precise. Following, or interleaved with this, is the process of trying to express the informal specification in a formal language. The more expressive the language, the closer the formal specification can be to expressing exactly the intent of the informal specification. However, the more expressive the language, the less likely there is to be effective fully automated support for proving that the properties hold of the system model. Since Linear Temporal Logic is a dominant language for formal specification, owing to the ability to effectively model check LTL properties of finite models, we focus on translating specifications into LTL. We discuss some of the ways LTL lacks expressive power, and we discuss techniques for circumventing some of the difficulties resulting from this deficiency. A method for capturing aspects of the environment, such as time, and information about past state involves augmenting the state with additional variables. The properties to which these auxiliary variables must adhere are typically not expressible in LTL. These auxiliary variables have no existence in the system design (or only an existence for purpose of being read, if they represent input to the system from the environment). To capture the additional needed restrictions on these variables it is necessary for it to be added to the system model. We propose a methodology for this addition that involves creating separate models capturing the needed behavior of just these auxiliary variables, and then automatically composing these models with the existing system model to generate the model used for model checking. This discipline allows us to isolate the aspects of the model expressing the auxiliary information so that it can be verified by other methods (such as informal proof through inspection). It also allows us to retain the system model intact so that we can have confidence that the results of the model checking applied to the composite model imply the intended meaning for the original system specification.

6 Hopes for the Future

In this paper we have identified three problems with transitioaning from natural language specifcations to formal language specifications:

- Natural language specfcations are generally more ambiguous than they first appear. Therefore, there is a need to identify ambiguiutes and disambiguate. The process of rendering the specifcations in a formal language can serve as a useful tool in identifying ambiguities.

- To date, formal langagues suitable for such automation as model checking are not adequately expressive enough to directly capture the natural langague specifications. We have proposed some methodologies for indirect capture through a combination of auxiliary variables and extra-logical requirements. This methodolgy handle common cases moderately well, but at heart is only a collection of heuristics.
- Adding auxiliary notions to the system specification requires adding them to the system model as well. Rather than clutter the system model with this auxiliary information not explicit in the implemention, we propose that the auxiliary information be captured in a auxiliary model constructed solely for this purpose, and then composed with the system model to obtain the model used for verification.

The problem that we have indicated the best solution for is that of the augmenting the system model. For certain methods of augmenting, it is clear that it is posible for this to be performed autmatically. So do it. And apply it. See how well we can handle real world problems this way.

We have discussed some heuristics/methodologies for for capturing natural language concepts in a language such as LTL. However, clearly we are contorting ourselves to fit the language. More work needs to be done on constructing langagues with the expressive power to capture the ideas needed in specifcations. Simply making the language more expressive, in the mathematical sense, isn't the answer. I do believe that a language such as Zermelo-Frankel set theory, or higher-order logic, which are expressive enough to capture basic mathematics, are expressive enough to capture system specifcations, and even user requirements. However, such languages cannot admit the decision procedures necessary for system verification. It is not clear, however, that for "real" system specification, the full expressive power of such langagues is required. One area of research that is needed is to find languages, probably similar to LTL, but with expressivity extended in directions that are those needed for capturing most commonplace specifcation constructs. An example of this might be adding the notion of the next-state value for a variable, in addition to the current value. Such constructs need to be chosen with consideration for both the extent to which they will facilitate rendering specifications, and the abilty to automatically check that they hold of system models.

Another possible partial answer is to translate the natural language into a highly expressive formal language, and then derive approximaitons to these requirements in a less expressive language suitable for automation such as model checking. In this scenario, at first derived formulae that were stronger than the original would be checked. If they succeeded, then the original requirement would be known to hold. If they failed to check, then counter-examples to the derived formulae could be generated and if these

With experience of what kinds of auxiliary information can be a

References

[1] Nikolaj Bjrner, Anca Browne, Eddie Chang, Michael Coln, Arjun Kapur, Zohar Manna, Henny B. Sipma, and Toms E. Uribe. Step: Deductive-algorithmic verification of reactive and real-time systems. In *International Conference on Computer Aided Verification*, volume 1102 of *LNCS*, pages 415–418, 1996.

[2] E. A. Emersom and J. Y. Hapern. 'Sometimes' and 'Not Never' revisited: On branching versus linear time temporal logic. *JACM*, 33(1):151–178, Jan. 1986.

[3] Elsa Gunter and Doron Peled. Path exploration tool. In Rance Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99*, volume 1579 of *Lecture Notes in Computer Science*, pages 405–419, Amsterdam, The Netherlands, 1999. Springer.

[4] Dexter Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, pages 333–354, Dec. 1983.

[5] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[6] C. Mazza, J. Fairclough, B. Melton, D. de Pablo, A. Scheffer, and R. Stevens. *Software Engineering Standards*. Prentice Hall, 1994.

[7] M. A. Ozols, K. A. Eastaughffe, A. Cant, and S. Collignon. DOVE: A tool for design modelling and verification in safety critical systems. In *Proceedings of the 16th International System Safety Conference*, Seattle, USA, September 1998.

The Fixed Logical Execution Time Assumption

Thomas A. Henzinger, *University of California, Berkeley*

Abstract:

A central challenge in real-time programming is the definition of a programming model at a level of abstraction that supports both implementability and verifiability. If a programming model is too close to an abstract specification, then it is difficult to generate efficient code. On the other hand, if a programming model is too close to the execution platform, then the gap between specificationand program is difficult to bridge. Many traditional real-time programming models are based on priorities. These models are arguably not sufficiently abstract, and the resulting code is often unpredictable with respect to both timing (jitter) and function (data races). Some newer programming models are based on the synchrony assumption, which postulates that computation is infinitely faster than the physical environment. These programming models are often too abstract and difficult to compile onto resource-constrained and distributed platforms. We present a novel real-time programming model, based on the FLET (fixed logical execution time) assumption, which is less abstract than synchronous models but more abstract than priority-based models. We demonstrate that FLET-based programming leads to predictable, composable, portable, and efficient real-time code.

Refining middleware functions for verification purpose

Jérôme Hugues, Laurent Pautet
{hugues, pautet}@enst.fr
École Nationale Supérieure
des Télécommunications
CS & Networks Department
46, rue Barrault
F-75634 Paris CEDEX 13, France

Fabrice Kordon
Fabrice.Kordon@lip6.fr
Laboratoire d'Informatique de Paris 6/SRC
Université Pierre & Marie Curie
4, place Jussieu
F-75252 Paris CEDEX 05, France

Abstract

The development of real-time, dependable or scalable distributed applications requires specific middleware that enables the formal verification of domain-specific properties. So far, typical middleware implementations do not directly address these issues. They focus on patterns and frameworks to meet application-specific requirements.

Patterns propose a high-level methodology adapted to the description of software components. However, their semantics does not clearly address verification of static or run-time properties. Such issues can be addressed by other formalisms, at the cost of a more refined description.

In this paper, we present our current effort to combine both patterns and Petri Nets to refine and then to verify middleware. Our contribution details steps to build Petri Net models from the Broker architectural pattern. This provides a model of middleware and is a first step towards formal middleware verification.

1 Issues in middleware development

Distribution middleware provides description methods, services and guidelines to ease the development of distributed applications. Middleware specifications describe the semantics and runtime supports for distribution.

Successful implementations of solutions such as CORBA, Java Message Service (JMS) or SOAP demonstrate that distributed applications require very different distribution models: Remote Procedure Call (RPC), Distributed Objects Computing (DOC), Message Passing (MP) or Distributed Shared Memory (DSM).

Besides, there is a rising demand for a wider range of runtime and platform support: embedded, mobile, real-time, multimedia, etc. These new criteria increase complexity in both middleware development and use. Middleware implementations should be versatile enough to handle different (and potentially antagonist) platform requirements; application must abide to complex middleware semantics.

Current middleware implementations rely on *patterns* to enable configurability and then to meet user requirements for one specific distribution model. Architectural and design patterns are introduced to describe specific solution to recurrent design problems (request demultiplexing, buffers allocations, concurrent execution, etc). Middleware is described by means of a language pattern that weaves together a set of related patterns. This approach proved its efficiency in various industrial projects [1]. Hence, the combination of patterns provides a high-level description of middleware. Yet, weak pattern descriptions may lead to slightly different implementations or implementations that interleave different patterns concerns. This impedes implementations verification.

Moreover, patterns are only descriptive. They do not provide any verification guidelines. Thus, implementations rely only on simple verification methods to verify behavioral-only properties: the use of some middleware functions and the execution of predefined test cases. But this approach lacks generality: it can only test a restricted subset of the infrastructure properties.

As middleware use evolves toward real-time and dependable applications, there is a strong need for formal verification of middleware with respect to explicitly defined properties. Yet, verification process is a complex task. The choice of a verification mechanisms is thus significant.

In the remainder of this paper, we detail our current effort on middleware verification. We focus on remote invocation models (RPC, DOC or MP) and exclude DSM. We present a middleware typical architecture, built around the *Broker* architectural pattern and show limits that prevent verification. Then we introduce our work to fill this gap and detail the modeling notations and process we use to verify the *Broker*. We also detail our specific middleware architecture, implemented by PolyORB[1] and demonstrate how separation of concerns eases verification. Then we explain how we refine the *Broker* and detail how to formally verify it.

2 Broker: a key middleware pattern

Basically, middleware provides mechanisms to enable transparent interaction between application nodes using messages. Reception of a message triggers: *1)* resources allocation, *2)* specific processing for the distribution model and then *3)* execution of application-specific code. A response may be sent back to the message initiator, following the same path. The architectural pattern *Broker* provides a view of the components involved in this process.

We first present this pattern; we show its importance for middleware specifications, performance and runtime; then we discuss limitations when coming to verification.

2.1 Overview of the Broker

The *Broker* architectural pattern provides a synthetic description of the role of middleware [2]: *" [..] (to) structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions."*.

The *Broker* pattern prescribes the use of many different objects: proxy, client and server, repository, bridge. These objects cooperate in the following way: *Servers* register themselves with the broker through the *Repository*, and make their services available to *Clients* through method interfaces; *Clients* access servers by sending requests via the *Broker*. *Broker* exchanges requests between nodes by locating the appropriate server, forwarding the request to it and transmitting results back to the client. *Proxy* and *Bridge* handle communication mechanisms and enable data exchange across heterogeneous platform.

We can note that the *Broker* specification presented above provides a complete view of an architecture to achieve remote service invocation. It covers all functions involved in middleware execution: protocol stack, data representation for transmission through network, resource allocation, etc. Hence, its precise definition and study will provide a first analysis of a middleware architecture.

2.2 Broker within middleware architectures

The *Broker* pattern has a key role in middleware architecture and implementations. Moreover, similar patterns [2] propose simpler views of the *Broker* adapted to specific cases. Variations of this pattern are used by middleware norms or specifications: e.g. for CORBA or Microsoft .NET specifications. Some implementations detail variations that support different distribution models or enable precise tuning of middleware performance:

- The Advanced Communication Toolkit (ACT) [3] provides a flexible implementation of the *Broker* pattern. It allows a precise description of resource allocation policies for multi-threading or data marshalling. ACT shows the *Broker* may serve as a basis to implement various distributions models. It supports CORBA (DOC) and cBus (Message Oriented Middleware, MOM).

- The ACE ORB (TAO) [4] demonstrates how the *Broker* pattern can be extended and then adapted to several concurrent executions policies. TAO proposes multiple patterns to control concurrent execution of *Broker* instances. A performance analysis revealed these different patterns enable great flexibility in configuration and good performance.

Hence, the *Broker* pattern is used under multiple forms at the core of most middleware architectures. Its precise definition and analysis would provide significant information about resource use, execution flow, middleware faults and performance analysis; but also to detect incorrect design.

So far, middleware implementations rely on slightly different behavioral descriptions of the *Broker*. This pattern definition is not sufficiently detailed and may lead to many fine variations introduced by implementation choices. Moreover, this pattern interleaves multiple functions: protocol, resource allocation, etc. Such a description impedes verification: it covers many complex functions. Thus, the *Broker* architectural pattern provides a specification of middleware architecture not suitable for formal verification.

However, patterns provide an elegant way to describe a component. We now present how we extract a precise specification of middleware components from the *Broker* architectural pattern that is suitable for verification.

3 Analysis guidelines

A complete analysis of the *Broker* pattern requires first a clear description of the component interface, related semantics and expected properties. Ultimately, this description is expressed using a notation that enables formal verification. In between, several transformations may be required to go from high-level unformal specification to strongly formalized description of a component.

This raises the question of the most adequate modeling notation (or set of notations) to achieve this process. We

[1]`http://libre.act-europe.fr/polyorb`

contemplate using one or more models among automata, UML diagrams (and derived stereotypes), Petri nets (colored, stochastic, etc) and architecture description languages (ADL). These are the most used notations for modeling software components.

We can note that none of these notation is supported by a complete specification and verification cycle. Each notation only covers a restricted part of the software life cycle: UML diagrams focus on system specifications and modeling; Petri nets on formal verification of controlled systems; ADL on the description of system architectures.

Thus, one has to use two or more notation to cover both specification and verification. Current research activities focus on the combination of different description models to provide a complete description of a component :

- For instance one can derive Petri Nets from UML state machines and diagrams [5] to achieve verification, yet there is no fully automated tool to complete this task.

- Another possibility is to rely on domain specific notations such as the Avionics Architectural Description Language (AADL) [6] or LfP [7]. They detail how new notations can be defined by extending and combining multiple models. Yet, they are still at an early definition stage and not fully supported by tools.

These studies provide guidelines for the formal specification and verification of components. They follow a top/down approach from high level specifications to formal one, enabling verification.

We propose to follow a similar approach, adapted to a very specific problem: verifying the *Broker* pattern. We present the different steps in the following sections. First we propose a middleware architecture that eases verification; then we refine the *Broker* pattern and define it with respect to our middleware architecture. Finally, we present the formal model of the *Broker* we produced using Petri Nets.

4 Middleware architecture for verifi cation

In this section, we present how specific middleware architectures enable formal verification. We introduce our proposal, the *schizophrenic* architecture, and our implementation: PolyORB.

4.1 Rationale

We stated in section 2.2 that the interleaving of many high-level functions is a major limitation for the verification of middleware architectures based on the *Broker* pattern. To solve this problem we have to propose a comprehensive definition and then separation of middleware functions.

Generic middleware proposes such a separation: they assert that middleware implementations have similar design. Hence, distribution models may be built from a set of generic elements using a functionality-oriented approach. Then, these elements are instantiated to conform to a specific distribution model.

Several projects demonstrate how middleware functionalities can be described by a set of generic services, independent from any distribution model. They propose a set of abstract interfaces. Distribution models are implemented by combining the concrete modules that implement these interfaces and provide access to generic middleware services.

- Quarterware [8] is generic middleware from which CORBA, RMI and MPI instances have been produced. These models have been implemented using a restricted set of components that can be extended to implement a specific model; or specialized for optimization and high-performance.

- Jonathan [9] architecture emphasizes on instances as adaptations of the core system Jonathan. Jonathan is a framework of configurable components and abstract interfaces. Dedicated instantiations provides CORBA (David), Java RMI (Jeremie) distribution models, or specialized ones for multimedia.

These different projects provide incomplete solutions for verification. They enable the implementation of distribution models as instances or *personalities* of a generic set of components. But personalities implementation interleaves instantiated components: this impedes verification.

Thus, we have to clearly separate middleware functions at both the definition and implementation levels. We now present our solution: *schizophrenic middleware*.

4.2 Schizophrenic middleware

Schizophrenic middleware refines the definition and role of personalities to increase separation of concerns. It introduces application level, protocol level personalities and a Neutral Core Middleware. The latter allows for interaction between personalities. Figure 1 presents interaction capabilities between personalities available in our implementation of schizophrenic middleware: PolyORB.

Application personalities constitute the adaptation layer between application components and middleware through a dedicated API or code generator. They register application components with the core middleware; and they interact with it to enable the exchange of requests between entities at the application-level.

Protocol personalities handle the mapping of personality neutral requests (representing interactions between application entities) onto messages exchanged through a chosen communication network and protocol. Requests can be received either from application entities (through an application personality and the neutral core) or from another node of the distributed application. They can also be received from another protocol personality: in this case the application node acts as a proxy performing protocol translation between third-party nodes.

The *Neutral Core Middleware* acts as an adaptation layer between application and protocol personalities. It manages execution resources and provides the necessary abstractions to transparently pass requests between protocol and application personalities in a neutral way. It is completely independent from both application and protocol personalities: this enables the selection and interaction of any combination of application and protocol personalities.

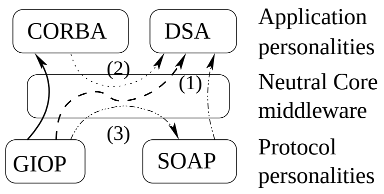


Figure 1. PolyORB's interacting personalities

Personalities implement a specific aspect of a distribution model. The Neutral Core Middleware enables the presence and interaction of multiple application and protocol personalities within the same middleware instance, leading to its "schizophrenic" nature (see [10] for more details).

Hence, this architecture separates three main components of middleware: protocol-side, application-side and internals. This reduces components interleaving. We now detail their interactions.

4.3 Separating middleware functions

Personalities implement middleware functions with respect to a specific semantics. Yet, most of these functions are notionally similar and can be defined as instances of some generic services.

Hence, schizophrenic middleware define generic services that express key middleware functionalities based on an analysis of multiple implementations. These services are focused on the completion of interactions between two nodes of a distributed application. One can combine particular services instances to implement the *neutral core middleware* and *application* or *protocol* personalities.

- **Addressing** Each entity is given a unique identifier within the entire distributed application.

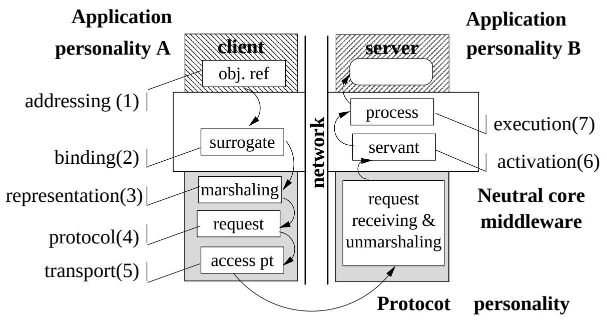


Figure 2. PolyORB's services

- **Binding** Middleware establishes and maintains associations between interacting objects and resources allowing this interaction (e.g. a socket, a protocol stack).

- **Representation** Request must be translated into a representation suitable for transmission over network.

- **Protocol** Middleware implements a protocol for the transmission of requests amongst nodes.

- **Transport** A communication channel is established between a node and an object to transmit messages.

- **Activation** Middleware ensures a concrete entity implementing objects is available to execute the request.

- **Execution** Middleware assigns execution resource to process every incoming request.

Figure 2 illustrates how PolyORB's services cooperate to transmit one request from one application personality to another, on two separate nodes using a common protocol.

The client (application personality 'A') gets a reference on the object from the *"addressing"* service (1); the core middleware creates a binding object (2, *"binding"* service): a dynamic gateway to the remote object through which the client communicates; a message is built from the request (3 and 4, resp. *"representation"*, *"protocol"* services). and sent to the remote node (5, *"transport"* service). Upon reception, remote node middleware ensures that a concrete entity implementing the object is available to execute the request (6, *"activation"* service) and assigns execution resources (7, *"dispatching"*) leading to the execution of application code by application personality 'B'.

So far, we demonstrated in [11] how composing and reusing services enable the rapid prototyping of RPC (distributed system annex of Ada), DOC (CORBA) and MOM (based on Sun's JMS API) middleware. PolyORB allows implementors to design multiple distribution model from a common set of services. Code reuse ratio reaches 70%; it is less than 30 % for generic middleware. We show how the instanciation of PolyORB services to build personnalities preserve separation of middleware functions. Finally, benchmarks show performance are correct for such a middleware, when compared to generic middleware.

Per construction, each service encompasses a restricted, well-defined set of functionalities. This separation of concerns enables separate analysis and verification of each middleware component. It is a first step towards the verification of the whole middleware. Thus, schizophrenic middleware architecture provides foundations for formal verification.

5 Refi ning the Broker pattern

The *Broker* architectural pattern interleaves functions that define middleware architecture. We now detail how we refine this pattern specification to separate its functionalities and thus facilitate verification.

5.1 From architectural to design pattern

The *Broker* pattern role is to coordinate distributed applications and handle request exchange between nodes. Its initial definition as an *architectural* pattern (section 2.1) encompasses protocol services, resource allocation, request execution, etc. It gathers many components that should be delegated to separate components.

We propose to refine this architectural pattern and to define a *Broker* design pattern, i.e. a component that interacts with other services to provide the same functionalities than initial architectural pattern. This definition provides greater separation of components involved in middleware.

PolyORB's architecture enables function delegation to a specific service. Thus, middleware are combination of these services. In this respect, we define the *Broker* design pattern whose unique role is to coordinate communications between nodes and to transmit requests.

This design pattern cooperates with other PolyORB's services. It sends requests from a node to another using both *addressing* and *binding* services. It receives incoming requests from remote nodes through *transport* service; *activation* service ensures request completion.

Hence, the *Broker* design pattern along with PolyORB's services is notionally equivalent to the *Broker* architectural pattern. Yet, it clearly separates functions into separate elements. We now define each of them.

5.2 Elements of the Broker design pattern

We focus on elementary abstractions to express the *Broker* design pattern interfaces. These abstractions interact with other PolyORB's services to form the complete middleware. The elements to describe this pattern are:

- **Asynchronous Event Sources:** enable waiting on external event sources, e.g. incoming data on TCP sockets. These sources are input points used by remote nodes to interact with this *Broker* instance. An API to manipulate and check sources is provided.

- **Job Queue:** stores all incoming *jobs* the *Broker* will process. Elementary *Broker* actions are defined as *jobs* to be processed by tasks running *Broker*; e.g. monitoring an event source, binding, processing incoming data, executing a request.

- **Broker:** enables services to access *Broker* functionalities either to process *jobs*, or to receive incoming data from remote nodes through *asynchronous event sources*. We distinguish the *Broker* main loop procedure that monitors sources or process jobs until a given exit condition is met; from the *Broker* API that allows for interaction with event sources (protocol level) and the job queue (application level).

- **Scheduling Policy:** allocates existing tasks to run the *Broker* main loop. Allocation is done upon the notification of event occurrences within middleware.

Figure 3 details communication between these blocks.

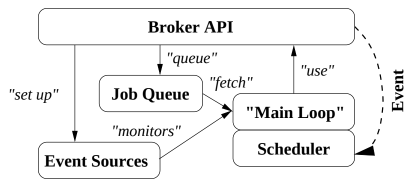


Figure 3. Interactions in *Broker* design pattern

This specification, and the unformal descriptions above provide a first overview of the *Broker* design pattern. It implies interactions among its sub-components. We now briefly present them in order to determine coupling between them.

- **Broker:** multiple tasks may concurrently execute *Broker*'s main loop, or other functions from the *Broker* API. This requires a precise locking policy.

- **Job Queue:** jobs can be queued by the *Broker* main loop upon the notification of an event on a source; or at the request of user code. This implies the definition of a specific queuing policy.

  Jobs are fetched in the *Broker* loop for processing.

- **Asynchronous event sources:** event sources list can be modified at any time; a traffic model defines how incoming data are received.

- **Scheduler:** Tasks executing the *Broker* main loop may be scheduled to achieve specific tasks. Figure 4 details the automaton used for scheduling. Current *Broker*'s global state determines the next action a task running the *Broker*'s main loop will perform: direct leaving if exit condition is met, processing jobs, blocking on event sources or going idle.

  The scheduler may be triggered at any time to ensure a task will process events: e.g. sources modification, new job queued, …

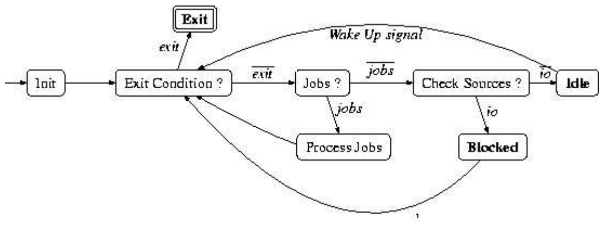


Figure 4. Task state automaton

These different elements show that this system is highly concurrent and driven by the occurrence of specific events. Thus we must choose carefully an appropriate modeling technique to model each *Broker*'s functions and their combination. We now detail how we move forward formal specification and then verification using Petri Nets.

6 Petri Net modeling of Broker

Petri Nets proved to be a rigorous formalism to verify concurrent event driven systems. Existing tools enable automatic verification of structural properties or model checking. In this section, we detail how we mapped previous specifications to Petri Nets.

We first briefly present well formed Petri nets. Then we discuss the modeling process of the *Broker*, and finally introduce the Petri net model of the *Broker*.

6.1 Unformal presentation of Petri nets

A well formed colored Petri net [12, 13] is a 5-uple $<P, T, Pre, Post, Types, M_0>$ where P is a set of places (depicted by circles), T is a set of transitions (depicted by rectangles), $Pre[t]$ is the precondition function for transition t, $Post[t]$ is the postcondition function for transition t, $Types$ is the set of basic types (a basic type is a finite set) and M_0 is the initial marking.

To each place p, a domain $Dom(p)$ is associated: $Dom(p)$ is the cartesian product of some basic types. $Dom(p)$ corresponds to the set of token color that place p can possibly contain. In figure 5, basic classes is C. The domain of place **a** is the cartesian product of C with itself, the one of **b** and **c** is C.

A marking $M(p)$ is associated to each place p: $M(p)$ is a multi-set over $Dom(p)$. Therefore, a marking M is the function that associates a marking to each place p of P. An element of a marking in a place is called a token. In figure 5, the initial marking associates one token having value <1> in place **b** and two tokens having values <1> and <2> in place **c** (function $<C.all>$ generates one token for any value of class C).

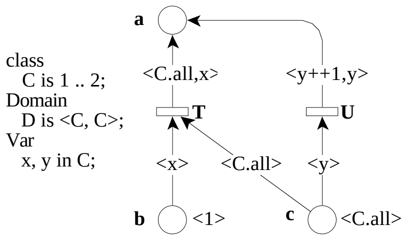


Figure 5. A small Petri net example.

Pre and *Post* functions describe how a marking is modified when an action is performed. Since actions are associated to transitions, instead of "an action is performed" we say: "a transition is fired". To each transition, a set of variables $Var(t)$ is associated. Each variable is defined over a basic type. In figure 5, $Var(\mathbf{T}) = <x>$ and $Var(\mathbf{U}) = <y>$. The binding of a transition is the association of an actual value to each parameter. When a transition fires, the corresponding tokens are generated in its output places. Based on this evolution, the state space can be generated. The figure 6 provides the one of our example and explicitly presents all possible bindings (the double circle represents the initial state and black circles represent deadlock states).

Computing the state space allows behavioral analysis such as detection of specific states (e.g. "is a given situation possible") or causal relation between two states (e.g. "if I reach state S_1, will I eventually reach state S_2?").

Petri nets also support structural analysis: properties such as invariants (e.g. "the number of tokens remains constant on a subset of places") are computed on the graph structure and thus do not require to compute the state space too. Thus, infinite systems may be verified [13].

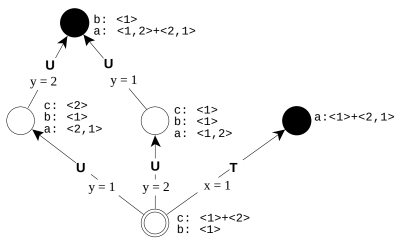


Figure 6. The corresponding state space.

6.2 The modeling process

Having unformally defined *Broker* subcomponents and their interactions, we now detail how to formally specify it.

We first transcribe the different components and interactions we presented into Petri Nets modules. We associate a specific action to each transition of the Petri Net; places represent states. Interaction between components is specified as common places between different modules. Hierarchy can also be used to provide partial views and helps in refining an initial Petri net model. However, this hierarchy should be flatten to use formal verification tools.

Then, Petri net modules are merged to produce a complete model, suitable for verification. To do so, we use the CPN-AMI[2] CASE environment that provides modeling facilities as well as model checking and structural analysis tools. Hence, the Petri Nets model for the *Broker* pattern is the aggregate of several Petri Net modules, each of which specifies one function of the *Broker*.

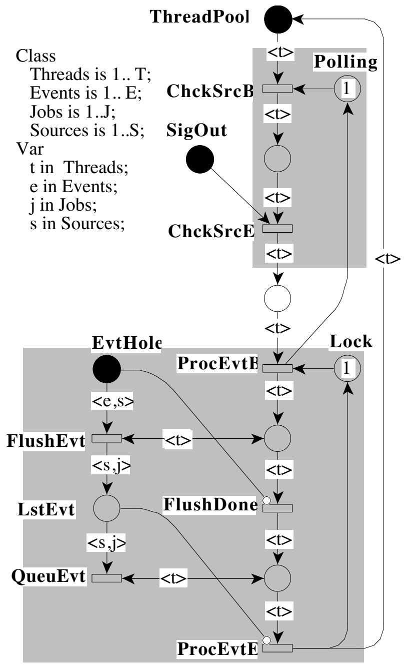


Figure 7. Petri Net for one *Broker* module

6.3 Petri Net of the Broker

By lack of space, we only present the Petri net module of one core function processed by the *Broker* main loop (see Figure 7). This procedure is triggered when a task is blocked, waiting for events. It consists of two phases: *1)* polling on event sources and *2)* processing events.

[2]`http://www-src.lip6.fr/logiciels/mars/cpn-ami`

- **Polling on event sources.** Place *ThreadPool* contains all threads scheduled to check event sources (see section 5.2). Only one task can actually check sources. When transition *ChckSrcB* fires, one available threads is selected to check sources. Transition *ChckSrcE* fires upon the notification of the presence of an event in the sources (place *SigOut*). Place *Polling* ensures only one thread can check sources.

- **Event processing.** Since we read from event sources (place *EvtHole*), this step has to be performed under critical section (ensured by place *Lock*). Transition *FlushEvt* is fired as long as there are events coming from any of the sources. Transition *FlushDone* fires when all events are consumed, which is enforced by the inhibitor arc from *EvtHole* to *FlushDone*. Events are stored in the Job Queue for further processing by others threads. The related functions are defined in other Petri net modules. When all events are queued, transition *ProcEvtE* fires to release the lock and restore the thread in the pool.

Places outlined in black have a special status in the Petri net module. They support communication with other Petri net modules. Their marking is generated by these modules and ensure an appropriate connection between the modules. They represent either other *Broker* functions or middleware modules interacting with the broker (i.e. behavior of PolyORB services). This typical composition technique is called *channel place* [14].

This Petri net module allows a stand-alone assessment of the modeled function. This can be done by changing the initial marking: each initial marking corresponds to a set of potential scenarios. This provides useful behavioral information on the correctness of the modeled function. All functions can be separately tested and then combined to form the complete Petri net model. Besides, module substitution allows us to define different scenarios that emulate specific conditions (e.g. queuing, locking or scheduling policies).

Then, these different models of the *Broker* can be tested, providing information on resource consumption/ We may first test for any deadlock or livelocks situations. Then, we may compute resources needed to fullfill a specific scenario; we may look for stable states or compute shorter or longuest processing path. The main advantage of this technique is to enable verification targeted on the way the middleware is used. Thus, optimizations can be formally verified according to specific execution conditions.

7 Conclusion and future works

This paper presents the first steps towards middleware verification. We have detailed how to extract from an unformal specification components that can be verified; and proposed the use of Petri Net to verify these components.

Most efficient middleware architectures rely on design patterns as a language to express and then implement user requirements. Test cases are defined to validate this architecture. Yet, this approach lacks generality: it only tests the use of a restricted set of functions.

Then, we presented the *Broker* architectural pattern. It provides a complete and precise definition of all components involved in remote service invocation. It is of common use in middleware architecture. We noted its analysis would provide analysis of middleware implementations. But this pattern interleaves many high-level functions. This impedes verification. We thus looked for precise separation of all middleware functions to ease the verification process.

We detailed existing modeling notations. We showed at least two different formalisms are required to enable the complete definition and then verification of software components. We chose to rely on design patterns notations and Petri Nets to model the *Broker*.

We first presented the *schizophrenic* middleware architecture and its implementation PolyORB. We showed how its architecture clearly decouples middleware functions at both definition and implementation level. We refined *Broker* architectural pattern and define a *Broker* design pattern. It interacts with other PolyORB services to fulfill the same functions. Moreover, the *Broker* design pattern embeds less functionalities. This also eases verification.

Finally, we explained how we produced Petri net models of the *Broker*. We presented how to build specific scenarios to verify properties with respect to application needs. We contemplate verifying our implementation PolyORB.

We defined specific conditions to be tested. This will require future work to be completed. We expect it will provide more information on middleware behavior with respect to specific scenarios and lead to the formal validation of properties of our model, and then of our implementation.

References

[1] D. Schmidt and F. Buschmann, “Patterns frameworks and middleware: Their synergistic relationships,” in *Proceedings of the 25th International Conference on Software Engineering*, 2003.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal., *Pattern-Oriented Software Architecture: A System Of Patterns*. John Wiley and Sons Ltd., 1996.

[3] C. Francu and I. Marsic, “An Advanced Communication Toolkit for Implementing the Broker Pattern,” in *Proceedings of ICDCS’99*. IEEE, June 1999.

[4] D. C. Schmidt, D. L. Levine, and S. Mungee, “The design of the TAO real-time object request broker,” *Computer Communications*, vol. 21, no. 4, pp. 294–324, Apr. 1998.

[5] J. Merseguer, J. Campos, S. Bernardi, and S. Donatelli, “A compositional semantics for UML state machines aimed at performance evaluation,” in *Proceedings of the Sixth International Workshop on Discrete Event Systems*, october 2002.

[6] H. Feiler, B. Lewis, and S. Vestal, “The SAE Avionics Architecture Description Language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering,” in *RTAS 2003 Workshop on Model-Driven Embedded Systems*, May 2003.

[7] D. Regep and F. Kordon, “**L***f***P**: a specification language for rapid prototyping of concurrent systems,” in *12th IEEE International Workshop on Rapid System Prototyping*, June 2001.

[8] A. Singhai, A. Sane, and R. Campbell, “Quarterware for Middleware,” in *Proceedings of ICDCS’98*. IEEE, May 1998.

[9] F. D. Tran and J.-B. Stéfani, “Towards an extensible and modular ORB framework,” in *Workshop of ECOOP’97*, Jyvaskyla, Finlande, Apr. 1997, http://sirac.inrialpes.fr/~bellissa/wecoop97/dangtran.ps.gz.

[10] T. Quinot, F. Kordon, and L. Pautet, “From functional to architectural analysis of a middleware supporting interoperability across heterogeneous distribution models,” in *Proceedings of the 3rd Int’l Symposium on Distributed Objects and Applications (DOA’01)*, Sept. 2001.

[11] J. Hugues, L. Pautet, and F. Kordon, “Contributions to middleware architectures to prototype distribution infrastructures,” in *Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping (RSP’03)*, San Diego, CA, USA, June 2003.

[12] G. Chiola, C. Dutheillet, G. Franceschini, and S. Haddad, “On Well-Formed Coloured Nets and their Symbolic Reachability Graph,” *High-Level Petri Nets. Theory and Application, LNCS*, 1991.

[13] C. Girault and R. Valk, *Petri Nets for System Engineering*. Springer Verlag, Sept. 2002.

[14] Y. Soussy, “Compositions of Nets via a communication medium,” in *10th International Conference on Application and theory of Petri Nets*, Bonn, germany, June 1989.

Eliciting a Formal Model From Informal Requirements Specified In a Natural Language -- Some Issues and a Particular Approach

Aravind Joshi, *University of Pennsylvania*

Abstract:

In this talk we will discuss some general issues concerning the extraction of a formal model from a requirement specification document in natural language. These issues pertain to the degree of feasibility, interface issues, and finally, problems in specifying evaluation criteria. In addition we will also describe briefly our approach to this elicitation problem. This work has just begun and the work is preliminary at this stage.

Model-driven Development
From Object-Oriented Design to Actor-Oriented Design

Edward A. Lee, *UC Berkeley*

Abstract:

Most current software engineering is deeply rooted in procedural abstractions. Objects in object-oriented design present interfaces consisting principally of methods with type signatures. A method represents a transfer of the locus of control. Much of the talk of "models" in software engineering is about the static structure of object-oriented designs. However, essential properties of real-time systems, embedded systems, and distributed systems-of-systems are poorly defined by such interfaces and by static structure. These say little about concurrency, temporal properties, and assumptions and guarantees in the face of dynamic invocation.

Actor-oriented design contrasts with (and complements) object-oriented design by emphasizing concurrency and communication between components. Components called actors execute and communicate with other actors. While interfaces in object-oriented design (methods, principally) mediate transfer of the locus of control, interfaces in actor-oriented design (which we call ports) mediate communication. But the communication is not assumed to involve a transfer of control.

By focusing on the actor-oriented architecture of systems, we can leverage structure that is poorly described and expressed in procedural abstractions. Managing concurrency, for instance, is notoriously difficult using threads, mutexes and semaphores, and yet even these are extensions of procedural abstractions. In actor-oriented abstractions, these low-level mechanisms do not even rise to consciousness, forming instead the "assembly-level" mechanisms used to deliver much more sophisticated models of computation. In this talk, I will outline the models of computation for actor-oriented design that look the most promising for embedded systems.

Documentation Driven Agile Development for Systems of Embedded Systems

Luqi, Lynn Zhang

Software Engineering Automation Center
US Naval Postgraduate School
{luqi; lzhang} @ nps.navy.mil

Abstract: This paper presents the framework of documentation-driven agile development (DDAD) methodology for high confidence systems of embedded systems. DDAD mainly includes two parts: a documentation management system (DMS) and a process measurement system (PMS). DMS will create, organize, monitor, analyze and transform all documentation associated with the software development process. The information will be stored in an abstract and active form that will support a variety of formal and informal documents for different stakeholders and can interact with software tools. PMS will monitor the frequent changes in system requirements and assess the effort and success possibility of the project with a measurement model based on a set of quantitative metrics that can be automatically collected in requirements phase and stored and organized in DMS. PMS will also measure the properties of the software system that must be realized with high confidence (safety in this paper) based on quantitative metrics. DDAD will provide a mechanism to monitor and quickly respond to changes in requirements and provide a friendly communication and collaboration environment to enable different stakeholders to be easily involved in development processes and therefore significantly improve the agility of software development of SoES. DDAD will also support automated software generation based on a computational model and some relevant techniques. Several potential application domains are proposed in the paper.

Keywords: Software Development; Documentation; Agility; Knowledge Representation; Systems of Embedded Systems.

1. Introduction

Design of real-time embedded systems involves a multi-disciplinary team of systems, software and hardware engineers. They have different concerns, use different tools, and work somewhat independently of one another. For a high confidence system of embedded systems, development is much more complex than development of monolithic embedded systems. Non-essential software complexity of a system of systems can have a greater negative impact on system behavior than for a single system. In general, systems of embedded systems are usually deployed for long periods of time, are used globally, and have mission critical requirements. They demand real-time performance and high confidence. Attributes like system effectiveness, availability, reliability, safety, security, and clarity of design are all essential. Most importantly, the SoES must rapidly accommodate frequent changes in requirements, mission, environment, and technology. Consequently they are often structured as a coalition of separate components to form systems of embedded systems with dynamic configurations. In addition, SoES are usually composed of component systems that were developed by different organizations with different tools and run on different platforms. A wide variety of stakeholders (sponsors, developers, users, maintainers, etc.) are involved in the overall lifecycle of the software [1, 20].

A large amount of research has been conducted on real-time systems. Progress has been made, but mostly on "point solutions" that address sub-areas of complex system development. Integrated systematic methods that collectively provide an end-to-end solution, are easy to use, and are amenable to computer aid are needed to meet these challenges.

Software development agility is drawing more and more attention in the software engineering community. Agile software development is presented as the solution to deal with the frequent changes of requirements [11]. This approach focuses on individuals and interactions over processes and tools; working software over comprehensive documentation; customer collaboration over contract negotiation; responding to change over following a plan [37]. Thus, compared to other methods heavily depending on the traditional documentation, many current agile software development methods try to provide better communications with the user, reduce the comprehensive documentation and be capable to adapt to requirements changes. Some typical agile development methods are extreme programming (XP); dynamic software development method (DSDM); adaptive software development; feature-driven development; lean development; rapid application development etc.

Extreme Programming (XP) was created in response to problem domains whose requirements change [8, 38]. The XP practices are also intended to mitigate the risk and increase the likelihood of success. XP requires an extended development team. The XP team includes not only the developers, but also the managers and customers, all working together elbow to elbow. Asking questions, negotiating scope and schedules, and creating functional tests require more than just the developers be involved in producing the software. However, XP is only suitable for small groups of programmers, between 2 and 12. XP was not designed for a project with a huge staff or a large number of different stakeholder roles.

DSDM uses an iterative process based on prototyping and involves the users throughout the project life cycle [9]. DSDM achieves delivery with tight timescales through shortening communication lines between users and developers, between analysts and designers, between and across team members, and between differing levels of management. The mechanisms by which these communication lines are shortened differ from one application to another. DSDM defines a strategy for defining what the necessary documentation set will be for a given project. Much of the documentation that is traditionally produced is for the transfer of ideas from one developer to another or from developers to users. DSDM provides guidance on how to decide what sort of documentation is necessary and why. There are key criteria that a project should satisfy for DSDM to be applied easily. The project should be able to identify all the classes of users who will use the end result so that knowledgeable representatives can participate throughout the lifecycle of the project and provide coverage of the views of all the user classes.

These agile methods' attitude to documentation is to reduce the amount of traditional informal documents as much as possible by increasing direct communications between users and developers. The problem with these approaches is that the users are required to be knowledgeable and well versed in the software domain skills to be able to participate in the development process. Following some of the agile principles runs a high risk when the motivated individuals don't have the requisite domain skills [39]. Moreover, software development automation is reduced when direct communications between users and developers are over emphasized. It's well known that the automation of development can significantly improve productivity and minimize errors in software products. A good tradeoff between software development automation and agility is needed to develop systems that require high confidence on a large scale with frequent changes.

Making suitable use of documentation in the development process can reduce the requirements for participants to have specific knowledge. Moreover, by generalizing and abstracting the essence of documentation and exploiting the capability for computer-aided documentation, documentation can be used to significantly improve the agility of SoES software development while sacrificing automation to a minimum extent.

According to traditional concept and current common practice, software documentation consists only of informal text and diagrams intended for human consumption. This kind of static information in documentation cannot provide effective support for the development process, especially for systems of embedded systems. In our opinion, this traditional concept should be extended so that all the information needed to carry out the development process is considered documentation. The requirements for both high confidence and frequent changes in systems of embedded systems can only be realized by development processes that provide effective computer aid. Effective documentation should support humans to the extent the relevant development processes are carried out by humans, and should support software tools to the extent development processes are carried out by tools. In the common case where an aspect of the development process is carried out by a collaboration of both humans and software tools, the documentation should provide two views, one for the humans and one for the tools. For such aspects, consistency and accurate correspondence between the two views are of most importance, and computer aid is needed to effectively realize these properties.

In this approach, models and simulations are included as documentation. Some typical models include computational models and design models. They serve as the basis to support development activities such as requirements analysis, architecture design, validation and verification. Simulation and prototyping are examples of computer aided processes used to check the correctness of the requirements for the system under development. With this extension, documentation can provide more effective support for whole development process. This paper proposes a documentation driven methodology with respect to the features of systems of embedded systems. This methodology will significantly improve the agility of software development to accommodate frequent changes in requirements of SoES and support partial automation of software development as well.

2. Overview of Documentation Driven Agile Development

Agile development emphasizes the relationship and cooperation of different stakeholders. It requires that the development group, comprised of system designers, hardware developers, software developers and customer representatives, should be well-informed, competent and authorized to consider possible adjustment needs emerging during the development process life-cycle [26]. Our idea to improve agility on a large scale by taking advantage of a good documentation system is depicted in Figure 1. It's named the Documentation Driven Agile Development (DDAD) methodology. Three typical development processes are shown to illustrate the methodology.

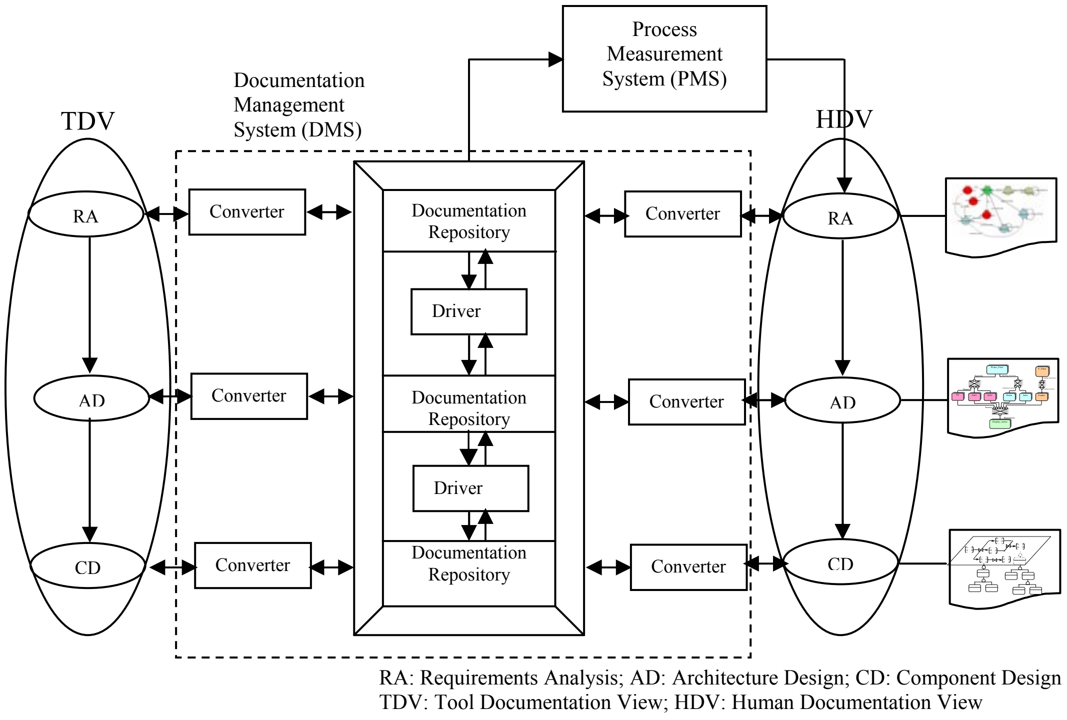

Process Measurement System (PMS)
Documentation Management System (DMS)
TDV
HDV
RA
AD
CD
Converter
Documentation Repository
Driver
RA: Requirements Analysis; AD: Architecture Design; CD: Component Design
TDV: Tool Documentation View; HDV: Human Documentation View

Figure 1. Documentation Driven Agile Software Development

The main idea behind DDAD is to build and use a Document Management System (DMS) and a Process Measurement System (PMS). The key to DDSD is that information from any activity involved throughout the software development process as well as the entire software life cycle will be recorded, managed and transformed by the DMS. The information will be stored in a form that will support a variety of formal and informal documents for different stakeholders and can be manipulated by a set of software tools. Eventually, the DMS will monitor and drive the overall development process and be applied throughout the entire software life cycle. DMS makes the development processes transparent and traceable, enables documentation to be updated quickly and facilitates communications and collaboration between stakeholders to promptly respond to changes in requirements. Process Measurement System (PMS) is used to track and analyze changes in requirements to verify the feasibility of the requirements, assess effort and risk of development, provide clues to modify the requirements, and measure the required high confidence properties. PMS is based on a set of quantitative metrics, most of which can be automatically collected in requirements phase. These metrics are stored and organized in the documentation management system. PMS and DMS working together will help the development of SoES rapidly accommodate frequent changes in requirements.

3. Documentation Management System (DMS)

DMS will create, organize, monitor, analyze and manipulate all documentation associated with the software development process. It will record all information from the development process such as requirement specifications, abstracted models, stakeholder input, design rationale, project management information and the source code. It will also extract important information from all development activities such as requirements analysis, prototyping, architectural design, software composition, system verification and validation, and system deployment. A documentation repository will be used to store the information in a structured, well-organized format. Information from the repository will support knowledge transfer between processes and generate the various presentations of this information for the different stakeholders and tools. The information stored in the repository drives both the Tool Documentation View (TDV) and Human Documentation View (HDV). By doing this, the development processes can be automated and the communications between stakeholders can be easier.

Tool Documentation View (TDV) representations are based on formal representations of the knowledge stored in the documentation repository and transformed into a format appropriate for use by the computer environment (software tools). They are usually in the form of mathematical formulas like temporal logic or process algebra, formal languages like PSDL or ADL, and programming languages. Typical TDVs include system models, requirements/design specifications, ontologies, source codes, test cases etc. They can also include application data such as geographic databases, results of measurements, medical records, financial databases, tables of properties of physical materials, and any other reference information relevant to system design.

Human Documentation View (HDV) representations are typically graphical in nature and in a form easily understood by humans. They are used by the stakeholders to communicate and interact with each other (sponsors, end users, developers (system, hardware and software engineers), technical supporters, etc.). Additional forms include text annotations written in natural language, decision tables and spread sheets. They can easily be expanded to include modern communication techniques such as video and audio clips. The latter can be useful for recording raw data about application process and content, to capture implicit requirements information that system stakeholders can demonstrate but cannot describe. The information in the HDV can include computed attributes that are not explicit in the information entered into the DMS. We envision this type of information to be useful for engineering and project management decision support. Examples include results of design rule checks, values of performance and reliability metrics, projections of project completion date and cost, and project risk metrics.

DMS contains a set of tools (e.g. converters and drivers) that will automatically convert the stored information from one representation to another to support different stakeholders and integrate the development processes by driving the knowledge transfer between them.

3.1 Documentation Repository

Keeping documentation up to date is difficult because of the various representations of information used in various stages of the development. The various representations of the same documentation information increase the complexity of maintaining information consistency and also hinder unaided communications between human and machine. Although multiple views of the information can solve this problem, how to maintain consistency among information presented to both the human and computer tools is still a challenge. This paper presents a documentation repository in which a common internal representation, template-based knowledge representation, is used to represent all information contained in the documentation.

Template-based knowledge representation is the kernel part of the documentation repository. It includes the following artifacts:

- Document Elements that are described by a semantic document model. It is an object model for the information contained in the documentation whose instances form an attributed object graph.
- A set of syntactic templates. The specifying elements together with syntactic templates can translate representations from one form to another or transform the information from one view to another.
- Attribute computation rules. This artifact represents the methods for computing derived document attributes.

Document Element

A document element is a basic building block consistent with the semantics of the information contained in the documentation. We use a semantic model named Attributed Object Graph Model (AOGM) to describe the semantics of each document element [16]. This is an object model of knowledge in the documentation repository. It has a nested structure with potentially shared nodes, i.e., directed acyclic graph structure. This representation is a generalization of abstract syntax trees that was developed in our previous research to represent constructs that appear in more than one context. This is a common pattern in software artifacts – for example, an operation can be defined once and called from many different contexts. In this model, each node represents a semantically meaningful structure, such as an individual requirement, a subsystem, an operation, or an operator within a logical expression. The nodes are the finest grain structures visible to the attribute computation rules. Furthermore, each node is an instance of an abstract data type. The computed attributes of each node correspond to the operations of the data type. Thus, invoking appropriate methods of the data type can derive the value of the corresponding attributes.

Syntactic Template

To improve the communication between the human and machine during the development process, computed multiple views of the same information for different people and different computer tools involved in the development provide a way to avoid inconsistencies between different representations of the same information due to incomplete manual updates. We are developing corresponding templates to support multiple views of the information. These views include the Human Document View (HDV) and the Tool Document View (TDV). In this case, the templates serve to transform the information from one view to another.

Syntactic templates are object operations with parameters. They provide a context for the resident document elements that will appear in different kinds of specifications. The combination of a document element and its syntactic context forms the multiple view presentation for the same information. Combining document elements with corresponding templates can also transform the information between representations written in different description languages.

We use tokens in an initial prototype representation of templates. Special tokens such as blank-filling tokens and action-interpreting tokens support computation of concrete document views. The blank-filling tokens indicate the blanks to be filled out, the actions to be interpreted and the information to be correlated etc. Action-interpreting

Template Items	Formalized Identification	Operational Semantic
Key-word	≪! key !≫	Key word to be matched
Token-Blank	≪@type@≫	Type to be replaced with the value of a document element
Token-In / Token-Out	≦ … … ≧	Enclosed by Token-in and Token-out will be contributed as properties of preceding Token
Routine Action	≪&action&≫ ≪&NL&≫ ≪&HL&≫ … …	Action to be performed New line is output Hyper Link is followed … …
Appearance of N≧0	*⌈ … ⌋	Items that appear 0 to n times
Appearance of 0 or 1	ο⌈ … ⌋	Items that appear once or none
Selective Appearance	⌈ <condition1> -> <item1> ⁝ <condition2> -> <item2> ⁝ … ⌋	Select one of values from list
Semantic symbol	≪, ≪@, ≪&, *⌈, ο⌈ , ⁝ ≫, @≫, &≫, ⌋, ≦, ≧	Enumerated characters have special meanings for software tools
Real Appearance	Typed characters	Any character appearing in the template only represents itself
Template Comment	//	Omitted

Table 1. List of Semantic Tokens

tokens are used to indicate actions to be conducted by software tools. Some possible tokens are listed in Table 1.

Attribute Computation Rules

We are studying methods for computing derived attributes and developing a set of schemata used to (a) calculate the attributes from the information in the documentation repository, (b) transform the information from one stage to another, (c) analyze the consistency between the information transformed between stages, and information views, and (d) extract subsets of documents needed for particular purposes.

Based on the Attributed Object Graph Model (AOGM), we developed a set of attribute rules to check whether significant aspects of the meaning are preserved during the information transformed from one development phase to another phase. These attribute rules can ensure that there is no information lost in transformation. We used timing properties transformation between requirement phase and design phase as the example to describe corresponding attribute computation rules [16].

3.2 Representation Converter

The representation converter presents the repository documentation to different stakeholders in a traceable, consistent and understandable way. These presentations include graphical depiction, formal description, logic formulation, audio and video media and so on. This tool will present the knowledge embodied by specifying elements and syntactic templates in a form the stakeholders can understand. The converter is based upon the combination of the knowledge-centric templates and the collection of specifying elements. It will "combine" the content of the document elements and the syntactic templates together to create and present desired documents for different stakeholders. Based on a specific template design, the tool generates presentation output for different stakeholders. A template selector is used to determine what kinds of documents will be produced. Also, based on the specific template design, the converter guides information to a collecting specifying element. This is similar to drag and drop with dialogue resources supported in a Windows application.

We have conducted research on a successful example that supports multiple document presentations based upon syntactic knowledge, such as the Computer Aided Prototyping System (CAPS) [17, 18, 40]. CAPS is the computed-aided prototyping system, whose computational model can be described in both PSDL specification and graphical depiction. Different stakeholders can share this information. Although a designer will use both the formal and graphical documents, a customer might use just a graphical document, and software tools use just the formal documents.

3.3 Transition Driver

A transition driver serves as a process transition tool based on the combination of knowledge-centric templates and a collection of document elements. Its function is to analyze the key information held by the templates and the document elements and to promote the transition of repository knowledge from one development process to the next. A transitional driver has the ability to act in both a forward and reverse direction. It can drive the transition of knowledge from one process to a succeeding one (forward) or from one process to a preceding one (reverse). In the first mode, the transitional driver promotes forward engineering of software products. The transition driver analyzes the preceding knowledge (knowledge used as an input), guides user's intervention, and then generates succeeding knowledge (process output). In the second mode, the driver promotes reverse engineering of legacy software systems if necessary. In this case, the driver serves as an extractor. It performs analysis and extracts useful information from what is normally considered the output information from a phase and generates what should have been the input information for that phase. A challenge in this area is how to best manage designer and user interaction to extract specification and design information the way it should have been built, rather than capturing the way it actually was built, including all of the errors and faults. A first step is to support annotations that identify such faults with links to explanations of why they constitute faults.

4. Process Measurement System (PMS)

The function of the process measurement system is to monitor the frequent changes in system requirements, assess the effort and success possibility of the project, and measure the high confidence properties of the system. The PMS obtains necessary information from the documentation repository. The analysis results will be presented to the developers and users as feedback. This quick communication is a key factor to make development of SoES agile: feedback is most useful when it can be delivered while the relevant aspect of the system is still in the process of being created, rather than after it has been completed and other system decisions have been made based on a faulty version of that aspect.

The process measurement system includes two parts: (1) a measurement model for effort and risk of a project; (2) a measurement model for high confidence. We have introduced a set of metrics to measure the effort and the risk in an evolutionary software project [22]. These metrics can be automatically obtained early in the requirements phase. They accommodate changes in requirements, process, technology, and resources of a project. Based on the set of metrics, a measurement model has been proposed [22]. The result is a statistical model that is used to estimate development effort and risk of failure of the project. The high confidence measurement model in this paper is only focused on software safety, because safety is the most critical factor for many DoD software systems and the state of the art in software engineering lacks a formal method and metric for measuring safety. We developed an Instantiated Activity Model (IAM) that supports a formal approach for safety analysis by providing precise metrics [30].

4.1 The Measurement Model for Effort and Risk of a Software Project

Current state of the art techniques for risk assessment rely on checklists and human expertise. This constitutes a weak approach because different people could arrive at different conclusions from the same scenario. The measurement model we developed for effort and risk is a statistical model based on a set of quantitative metrics. The metrics include requirements volatility, organization efficiency, product complexity, and technology maturity. This model will enable different program managers to derive the same projections on the same software project.

Metrics for Requirements Volatility

Requirement changing is the most significant characteristic for a system of embedded systems. Requirements volatility clearly influences the possibility of project success. From the point of view of the metrics, a change in a requirement can be viewed as a death of the old version and a birth of the new one. The requirements volatility can be obtained from birth-rate and death-rate. Birth-rate is defined as the percentage of new requirements incorporated in each cycle of the evolution process. Death-rate is defined as the percentage of requirements that are dropped by the customer in each cycle of the evolution process. The requirements volatility (RV) is defined as:

$$RV = BR + DR,$$

where, $BR = (NR / TR) * 100\ \%$, $DR = (DelR / TR) * 100\ \%$, NR = number of new requirements; DelR = number of requirements deleted; TR = total number of requirements.

Metrics for Organization Efficiency

The efficiency of the organization can be measured by observing the fitness between people and their roles in the software process. The skill match between the person and the job is required to estimate the speed in processing information and the rate of exceptions, which in turn affect efficiency. Efficiency also depends on many factors like team structure, experience, and tools. Simulations have shown that there exists an easier way to estimate team efficiency by observing the ratio between direct working time and idle time. The team efficiency metric (EF) is defined as:

$$EF= Dwork\% / Idle\%+Dwork\%$$

where Dwork% is the percentage of direct working time; Idle% is the percentage of idle time.

Metrics for Product Complexity

Product complexity is in general a function of the relationships among the components of the product. Hence, it is important to measure the complexity as a predictor. Product complexity is also directly related to the effort needed to develop a product.

Some requirements are difficult for the user to provide and are difficult for the analysts to determine. It's notably the case for real-time systems. The best way to discover these hidden requirements is via prototyping. CAPS is a CASE tool specially suited for this task, which uses the Prototype System Description Language (PSDL) [17-19]. Specifications written in PSDL can be analyzed to compute the complexity. Metrics for complexity can be defined by using a hybrid complexity measure that properly accounts for data flow and the properties associated with each operator and data stream in PSDL. A complex metric FC is defined as follows:

$$\text{FC} = \sum_{i=1}^{n} w(o_i)[dsi(o_i) * dso(o_i)]$$

where, $w(o_i) = 1 + \sum_{k=1}^{m} pw_k * c_{ik}$ is the total property weight of operator o_i. pw_k is the property weight of the kth property, with $0 \le pw_k \le 1$ and $\sum_{k=1}^{m} pw_k = 1$. c_{ik} is the property occurrence coefficient, with $c_{ik} = 1$ if operator o_i has property p_k and $c_{ik} = 0$ otherwise. m is the numbers of property types in PSDL. $dsi(o_i)$ is one plus the number of data streams flowing into operator o_i; $dso(o_i)$ one plus the number of data streams flowing out of operator o_i; *n* is the total number of operators.

Metrics for Technology Maturity

The software industry is characterized by frequent technology changes. A system of embedded systems is usually deployed for long periods of time and is used globally. In the process of evolutionary development of a SoES, the related technologies will change significantly during the period the system is deployed. Generally, the newer the technology is, the more quickly the technology changes. The impact of technology maturity on success of a project, especially for a SoES, is important.

Technology mainly consists of two parts. One is the software technologies that are selected to implement the project. The other is the domain technologies involved in the project. The choice of implementing technologies should be subordinated to the project domain technologies and requirements.

A new technology becomes mature in the process of transition from a scientific discovery to routine engineering practice in product development. Technology transition is referred to as diffusion in the literature. Diffusion is the process by which an innovation is communicated through certain channels over time among the members of a social system. Based on information theory, communication theory, and statistical mechanics, we developed a metric, named 'technology temperature *T*', to measure the maturity of a technology [23].

According to information theory, the quantity of information in an ensemble of possible messages is measured by entropy. A message is made up of sets of terms. In this context, the relevant information is the knowledge about a technology. Following reasoning similar to that used in statistical and condensed particle physics and recalling the standard definition from the thermodynamics, the temperature *T* for technology transition can be defined as follows:

$$\frac{1}{T} = \frac{\Delta S_H}{\Delta n}$$

where, Δn is the change in the number of terms of a message alphabet $\varXi$. ΔS_H is the change in entropy. The entropy is defined as follows: for the message alphabet $\varXi$ with the given probability mass function

$p(x) = \Pr\{X = x\}, x \in \Xi$, X is a discrete random variable, the definition of information entropy is $S_H(X) = -\sum_{x \in \Xi} p(x)\log_2 p(x)$.

The temperature is measured in "degrees" in a physical system, however, in the context of information degrees can be expressed in information units (bits). The value of T represents the maturity of a technology. It's a function with respect to time step [23].

Measurement Model

A Weibull distribution can be used to build the measurement model. The Weibull distribution was originally used to model strength of Bofors's steel, fiber strength of Indian cotton, length of syrtoideas, fatigue life of steel, statures of adult males, and breadth of beans. Many authors have advocated the use of this distribution in reliability and quality control [21, 25]. Others used it to model software life cycles [15]. The three parameter Weibull distribution is defined as follows.

A random variable x is said to have a Weibull distribution with parameters α , β and γ ($\alpha > 0, \beta > 0$) if the probability distribution function (pdf) and cumulative distribution function (cdf) of x are respectively:

$$\text{pdf: } f(x) = \begin{cases} 0 & x < \gamma \\ (\alpha / \beta^{\alpha})(x-\gamma)^{\alpha-1}\exp(-((x-\gamma)/\beta)^{\alpha}) & x \geq \gamma \end{cases}$$

$$\text{cdf: } \mathrm{F}(x) = \begin{cases} 0 & x < \gamma \\ 1-\exp(-((x-\gamma)/\beta)^{\alpha}) & x \geq \gamma \end{cases}$$

where,

- x is the random variable under study. In our context, x can be interpreted as development time.
- α is a shape parameter. It affects the skew of the function. When α = 1, the function reduces to the exponential distribution. The combined effect of α and β controls the variability of the pdf.
- β is a scale parameter that stretches or compresses the graph in the x direction.
- γ is a location parameter that determines the mean of the pdf.

We have conducted a large number of empirical experiments to determine the relationship between the parameters in the above model and the quantitative metrics above [22]. When the metrics are input then development effort and success possibility of the project can be estimated by the model. The outputs of the model are important supporting information to help the sponsors and developers to make decisions about the next process.

4.2 The Measurement Model for Safety Analyses

Safety is a critical to many high confidence systems of embedded systems, especially for DoD systems. Software safety focuses on the failures of the system as they relate to hazardous events. A system is considered as "safe" if the probability of a hazardous failure has been reduced to some defined acceptable level. Safety is not a Boolean value of purely safe or unsafe, but a variable that ranges from completely unsafe towards safe [31, 32]. We developed a formal Instantiated Activity Model (IAM) and a metric to measure the probability that a hazardous event will occur and the severity of that hazardous event [30].

Instantiated Activity Model (IAM)

The IAM is a typical Input-Process-Output (IPO) block schema dealing with a set of related activities such as, input, process, output, failure, malfunction, etc. Figure 2 gives an example of an IAM. This is a typical IPO block with possible failure attached to the activities. For instance, Input I_1 with potential failure F_1, through successive activities Process P_1 with potential failure F_2 and Output O_1 with potential failure

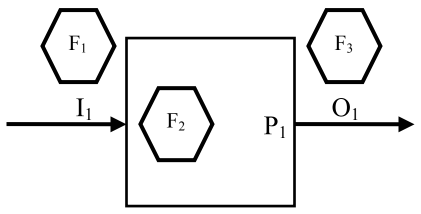


Figure 2. An Instantiated Activity Model

F_3 would result in a failure leading to a malfunction. The IAM reveals the relationship between essential IPO activities, the potential failures, and a hazardous situation or malfunction so we can establish a metric base for the safety analysis and risk assessment.

Hazard Probability of the IAM

The IAM is the key that supports formal approach for system safety analysis and risk assessment. This is based on the probability that a hazardous event will occur and the severity of that hazardous event (i.e., the consequences). Through the combination of these two elements, we can derive the hazard probability for the system as follows:

$$P_H(g) = \sum_i P_f(F_i, g) * P_e(A_i) * P_e(A_i\{DA_i\})$$

where $P_f(F_i, g)$ stands for probability of activity failure at degree *g*, *g* is the failure severity degree, $P_e(A_i)$ stands for probability of activity execution, $P_e(A_i\{DA_i\})$ stands for the probability of execution of A_i and $\{DA_i\}$, $\{DA_i\}$ stands for the dependent activities caused by activity A_i, A_i is the ith element of *A*, $A = I * O \cup R$, $I = \{I_1, I_2, I_3, \cdots \mid \text{all possible input activities}\}$, $O = \{O_1, O_2, O_3, \cdots \mid \text{all possible output activities}\}$, $S = \{R_1, R_2, R_3, \cdots \mid \text{all possible process activities}\}$.

The goal of making the IAM measurable on probability of failure is to identify potential hazards before the start of development, balancing development against effect. This method is especially effective for systems of systems. We can assume that each component system may have a myriad of different process flows that ultimately may result in a malfunction. We determine single failure probabilities using appropriate methods, as well as the determination of applicable process execution and related execution probabilities. It is possible to derive the probability that the whole system with execute a malfunction.

The risk exposure is the hazard probability times the cost of hazard occurrence.

5. Automated Software Generation based on Computational Models

DDAD integrates key processes in the software life cycle by the documentation management system (DMS). Models, activities, prototypes, simulations involved in these processes will be stored and manipulated in DMS. Supported by DMS, automated program generation can be realized based on a well-defined computational model and series of relevant techniques. A computational model was developed to describe the emergent properties, the interactions between component systems, and constraints associated with both functional and non-functional properties of a SoES [20]. A SoES ζ is modeled as follows:

$$\zeta = (S, E, C, D, F_1, F_2)$$

S is the component system set, $S = \{s_i \mid i \in [1, n]\}$, s_i denotes the component system constituting SoES (n is the number of component systems in the whole SoES); $E = \{e_{jk} \mid j, k \in [1, n]\}$ denotes the interaction sets between component systems, e_{jk} denotes the set of interactions from component system s_j to component system s_k; $C = \{c_i \mid i \in [1, n]\}$ denotes constraint sets on how the component systems are used in the given environment. c_i is a set of constraints on s_i. $D = \{d_{jk} \mid j, k \in [1, n]\}$ denotes constraint sets on interactions between component systems, d_{jk} is a set of constraints applied to interactions in e_{jk}.

Constraint sets *C* and *D* include the constraints for the design phase. They are refined from emergent properties *G* and high confidence constraints *H* of a SoES,

$$C = F_1(G, H); \quad D = F_2(G, H),$$

where F_1 and F_2 are two maps that map emergent properties and high confidence measures into local constraint sets on component systems and local constraint sets on interactions between component systems respectively. The mappings specify what must be assessed to ensure that the SoES satisfies its requirement with high confidence, if it has already been certified that the individual s_i meet their requirements with high confidence. The constraint sets also represent a design for the systems integration, which will be realized by wrappers around the s_i.

Based on this model, a prototype system can be established to validate the requirements for a SoES. Well-formulated prototyping documentation can be used to promote system transition by extracting compositional architecture and evolving components. We found a way to build an explicit architecture for a prototyping system so that the product system can evolve through a transitional procedure [29]. The compatible composition model allows both explicit architecting and componential evolving by incorporating computer-aided prototyping techniques into a transitional process. Additionally, we introduced an object-oriented model for interoperability via wrapper-based translation [28]. This model performs transition from a computational phase, through a compositional phase, to a componential phase. During the transitional process, documentation passes throughout the development process. These results support automated software generation.

6. Development Knowledge Sharing Based on Ontologies

Collaboration capability between stakeholders is another important feature of DDAD. Effective sharing of information and interoperation of development artifacts are vital to collaborative software development, e.g. development of SoES. Ontology is now widely used for realizing knowledge sharing between organizations and/or individuals who have different culture backgrounds. Ontology is the term used to refer to the shared understanding of some domain of interest that may be used as a unifying framework to solve problems in that domain [24]. An ontology is a set of definitions of content-specific knowledge representation primitives: classes, relations, functions, and object constants. We have studied how to establish the software development tool ontology to improve interoperability in heterogeneous software development [13]. The methodology for constructing an ontology consists of 6 steps: (1) Identifying the purpose and scope of the ontology; (2) Feature modeling; (3) Establishing commonalities; (4) Determining tool ontologies; (5) Representation of the domain; (6) Documenting the ontology. The ontologies are important parts of the documentation repository to support collaboration between stakeholders.

(1) *Identifying the purpose and scope of the ontology*. One of the most important steps in constructing an ontology is to make an early decision about the purpose of the ontology. This purpose provides a controlling perspective on the terms, attributes of terms, and relationships captured in the ontology. The scope of the ontology provides a guide to the depth and breadth of the intended ontology, consistent with the purpose.

(2) *Feature modeling*. This step is to perform a domain analysis of software development tools by constructing and then considering the feature models of tools. Feature modeling is a method used to help define software product lines and system families, to identify and manage commonalities and variabilities between products and systems [14]. Feature models represent an explicit model of a device or system by summarizing the features and the variation points of the device/system. A feature model for software system captures the reusability and configurability aspects of reusable software. As an example, Figure 3 illustrates a feature model of a how PSDL timing constraints are implemented in CAPS.

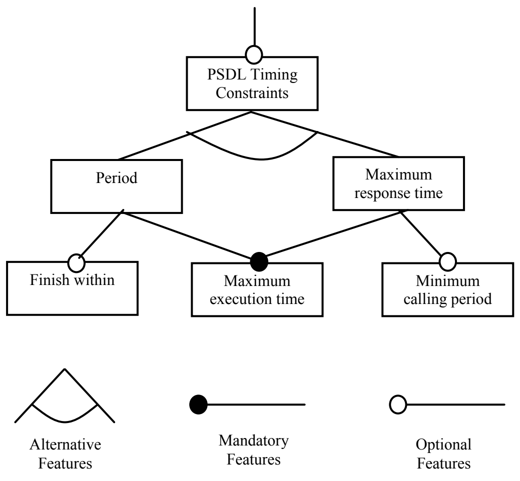


Figure 3. Feature Model of the PSDL Timing Constraints of CAPS

(3) *Establishing Commonalities*. This step is to isolate and annotate the commonalities that exist between the feature models. These common features then form the basis for the basic ontology terminology of the software development tool federation. The approach in this step is to reason about the feature diagrams, develop lists of potential terms from the feature diagrams, identify common terms between the lists, and then construct affinity diagrams of these common terms. Affinity diagrams are hierarchical Venn diagrams that provide groupings of related terms. The groupings of terms in the affinity diagrams then provide the basis for the hierarchy of terms in the software development tool ontology.

(4) *Determining tool Ontologies*. This step is the construction of the detailed ontologies of the tools to be used. In the case of tool ontologies, the detail needed for interoperability is dictated by the detail available through the API or source code (which ever is available) of the tool. Therefore, the ontology is derived from a selected set of classes and public methods related to the artifacts that are to be transmitted to (or received from) other software tools.

(5) *Representation of the Domain*. The fifth step requires that the relationships between all ontologies be identified and annotated. UML can be used to represent inter-relationship of ontologies. Such representations then make it possible to construct a set of all federation entities in the domain. When augmented with attribute computation rules, this representation can be made effective.

(6) *Documenting the Ontology*. The final step is to document the ontology. All assumptions about the domain and information about the meta-data used to describe the ontology should be annotated in the documentation repository in the form of template-based knowledge representation.

7. Methods and Models for Interoperability

We developed an Object Oriented Model for Interoperability (OOMI) to capture the information required for resolving the representational differences that exist in autonomously developed systems [33, 34]. Defining the interoperation between systems in terms of an object model provides a foundation for easy extension as new systems are added to an existing federation of systems.

The real-world entities and behavior information shared among a federation of interoperating systems are modeled in the OOMI using the concept of a *Federation Entity (FE).* For each FE, one or more *Federation Entity Views (FEVs)* are used to distinguish the differences in the state and behavior information used for representing the same real-world entity on different systems (Figure 3).

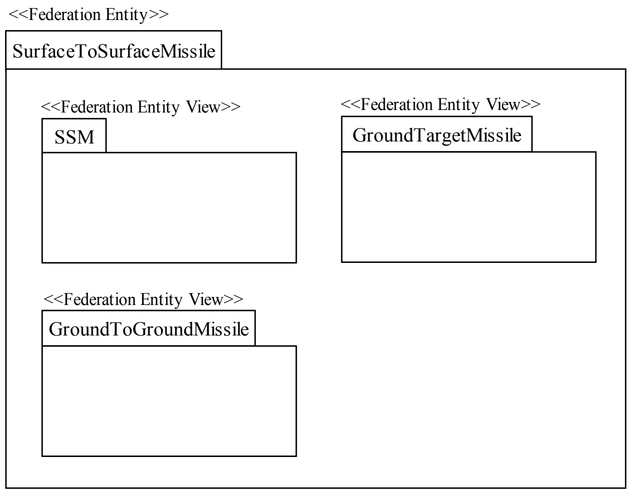


Figure 3. Defining Federation Entity (FE) and Federation Entity Views (FEVs) for Real-World Entity

It is expected that for a federation of heterogeneous systems, a number of real-world entities will be involved in the interoperation between systems. Under the OOMI, the collection of real-world entities used to define the interoperation of a specified federation of systems is termed a *Federation Interoperability Object Model (FIOM)* (Figure 4).

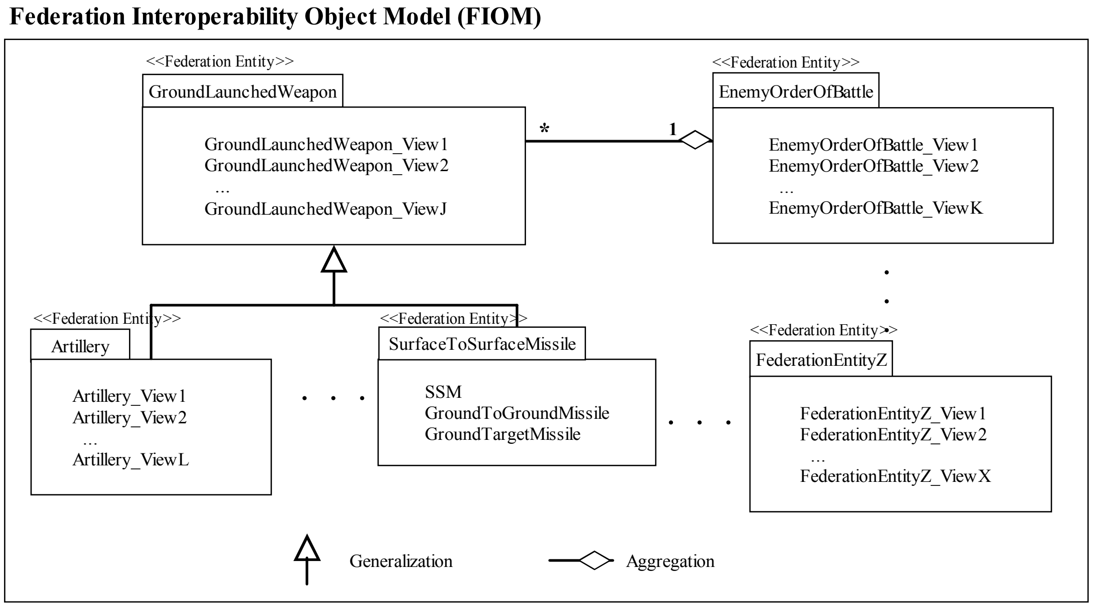


Figure 4. Federation Interoperability Object Model (FIOM) Representation

We also provided a Translation Generator for the Interoperability Engineers (IE) to define correspondences between the federation and component models' attributes and operations and generate the translation code skeletons, which can be modified to add functional or other transformations as necessary to resolve representational differences via the OOMI IDE facilities. The resultant wrapper-based Translator uses the FIOM, which the IE constructed using the OOMI IDE, to reconcile differences in real-world entity view and representation among component systems of a federation at run-time.

The initial use of the model is targeted for integration of legacy systems. Although these legacy systems generally have not been developed using object-oriented paradigm, an OOMI can easily be constructed from the external interfaces defined for most legacy systems (whether object-oriented or not).

We investigated formal models and mechanisms for describing the QoS attributes and techniques to assure the specified QoS. We developed a framework that allows an interoperation of heterogeneous and distributed software components. The framework incorporates (1) a meta-component model that describes the components, their services and service guarantees, and the infrastructure for integrating different component models and sustaining cooperation among heterogeneous components, (2) formal specification of components based on a two-level grammar, (3) validation and assurance of QoS based on event trace, and (4) generative rules for assembling a set of components out of available choices. We developed a Quality of Service behavior model based on the event trace analysis. The event trace approach allows us to directly examine specific quality of service actions that take place during program operation. In addition, we developed techniques to provide decision support for optimizing distributed object servers utilization, as well as the use software decoys to improve the security of systems of embedded systems [35, 36].

8. Applications of DDAD

8.1 Joint Tactical Radio System (JTRS)

The Joint Tactical Radio System (JTRS) is a revolutionary communications system that will be the foundation for all future Department of Defense tactical radios. JTRS will provide America's warfighters with state-of-the-art, software re-programmable, multi-band/multi-channel, network-capable systems that offer an interoperable, flexible and adaptable network for simultaneous voice, data and video communication [10]. It will create seamless interoperability and linkage among all military's air, land and sea legacy radio networks. Varied configurations of the system will advance communications mission requirements. The JTRS attribute of extendibility supports incorporating changes that are typical of many emerging requirements. In general, new requirements will be satisfied without hardware change provided the new waveform fits within certain bandwidth, data rate and transmission frequency bounds.

JTRS is a typical real-time, embedded, distributed, heterogeneous, and software-intensive system. The software implementation in JTRS should be able to dynamically adapt to the radio environment in which it is located at different times. A powerful documentation management system is needed for the JTRS program. Development of JTRS is complex and long-term. JTRS will be developed in several stages: Cluster 1 represents the first segment of the joint tactical radio system. The planned Clusters 2, 3, and 4 will address the handheld, maritime, and airborne needs. A team led by Boeing has been selected to begin building common tactical radios. The Boeing team is comprised of many sub-teams that take charge of different tasks [5].

A knowledge sharing and management environment can be constructed based on the idea of the documentation management system (DMS). This environment will support the decision coordination and cooperation between development teams. The documentation repository can be used in not only software development but also system and hardware development of JTRS as long as the related knowledge is appropriately represented in the form of template-based knowledge representation. The maintainability, traceability, consistency, understandability of documentation repository and the ability of quickly tracking and responding changes in requirements will increase the efficiency and decrease the risk of the development of JTRS. This application requires attention to the finer points of developing a distributed implementation of the DMS.

8.2 Ballistic Missile Defense Simulation Systems

The evolving ballistic missile defense problem must be solved to support a long-term strategy that calls for an integrated and adaptable "system of systems" to defend U.S. territory, forces, allies, and other interests worth protecting [2]. Credible Department of Defense models and simulations (M&S) of ballistic missile defense systems are expected by National- and Department-level decision-makers [6]. Many of these large-scale, software-intensive simulation systems were autonomously developed over time, and subject to varying degrees of funding, maintenance, and life-cycle management practices, resulting in heterogeneous model representations and data. Systemic problems with distributed interoperability of these non-trivial simulations in federations' persist, and current techniques, procedures, and tools have not achieved the desired results. Establishing credibility in DoD simulations involves many disciplines and knowledge areas including software engineering, processes, quality, product management and architecture. The Department's complex organizational dynamics, and complicated acquisition procedures also impact the level of M&S credibility, at times adversely.

There are two ways to apply the idea of DDAD to ballistic missile defense simulation systems. One way is to use DDAD directly in the development of simulation software that is credible. The other way is to apply the main idea of DDAD in simulations. A documentation management system for simulations can be built. This will enable all information involved in simulations to be well organized and manipulated so that the simulation processes are transparent, traceable and maintainable. Credibility of the simulation results will therefore be improved.

8.3 Joint Forces Program

Joint forces are now more important than ever because in today's world the traditional distinctions between maritime, land and air theatres of operations have become less relevant. By operating as a single, united force, the Navy, Army and RAF can produce a bigger punch, maximizing operational effectiveness and increasing the chance of success [7]. Interoperability requirements are critical to joint force programs. Since interoperability requirements are dynamic, and often poorly understood before systems are put to use in the field, the requirements and acquisition communities must have a flexible and powerful method to communicate in order to overcome these challenges.

Based on the idea of DDAD, we have proposed a unified repository of architectural data, with the ability to be viewed in several forms (i.e. with the ability to create multiple architectural views), each tailored to the needs of different stakeholders [12]. The power of this methodology is that it provides a mechanism by which functional and interoperability requirements are captured, defined, and levied on systems based on how they will be employed. This is a dynamic process, which can accept changes to requirements, system environments, and domains; and which supports time-phasing, spiral development, assessment of requirements vs. capabilities and operational vs. system needs.

9. Conclusions

This paper explores a new view of documentation that can better serve development of systems of embedded systems. The different views provided by the DDAD approach give project managers, developers, sponsors, maintainers and end-users the ability to express their opinions or propose requirements changes if needed by adding related documents via a user-friendly interface. This information will be recorded in a form that can be manipulated, automatically analyzed and made available throughout the rest of the development process. DDAD will track these changes and help to ensure that information will not be corrupted in transformation from one phase to another. DDAD provides a method that encourages stakeholder involvement while updating the requirements and consistently providing this information for later use. DDAD also supports automated software generation by using a computational model, rapid prototyping and other related techniques. This is helpful to achieve a good tradeoff between stakeholder interaction and process automation. DDAD also provide a method to monitor and respond to frequent changes in requirements. Consequently, agility of the development will be greatly increased.

By using the DDAD approach in every phase of development, even the automated processes, it should become practically feasible to record, compile and present information to different stakeholders and tools in a clear, understandable way at a level of complexity required to meet the stakeholders' needs. By having these different views available at various stages of development, stakeholders will be able to effectively monitor the development process and communicate with each other. This improved transparency provides valuable information needed for quality control and overall process improvement.

Software development processes from one phase to another are embodied as capture of relevant information (e.g., design specification, quantifiable attributes), definition of document information models and view presentation models, simulation of semantic behavior (e.g., executable specification), and transformation of documents exploited by various phases. With insight into the future development of documentation, the documentation repository will support transformation from high-level description (in some specification languages) to low-level description (in some programming languages) with mapping between those descriptions.

DDAD also provides comprehensive support for software maintenance and evolution. In DDAD, all the activities and information used by the development processes are accurately recorded and organized in a well-formulated documentation system that drives the system development and build processes. This will ensure overall system properties are precisely documented and consistently updated and transferred throughout successive phases and available after system release. The documentation will retain sufficient detail to provide a sound basis for fault tracing, bug repairing and overall system improvement. DDAD will keep track of system configuration, document dependencies and system status and enable the software to respond to future changes in requirements thereby supporting maintenance and evolution of the system.

Keeping track of accurate dependency information is critical for automatically locating the relevant parts of a maze of documents for resolving a given system evolution issue.

From the viewpoint of long-term system construction, technologies for computer-aided documentation repositories will drive the form of documentation standard needed for more effective regulatory management. Much of the presented infrastructure can be generalized from software development to the entire systems engineering and certification process.

DDAD will be a promising methodology to build a high confidence system of embedded systems. Three potential applications were presented in the paper, but the methodology and idea of DDAD can be used in many more industrial domains.

Reference

[1] B. Boehm, "Software Risk Management: Overview and Recent Developments", 17th International Forum on COCOMO and Software Cost Modeling, Los Angeles, CA, October 22-25, 2002, http://sunset.usc.edu/events/2002/cocomo17.

[2] D. C. Gompert, J. A. Isaacson, "Planning a Ballistic Missile Defense System of Systems", http://www.rand.org/publications/IP/IP181/.

[3] E. Hall, Managing Risk. *Methods for Software Systems Development.* Addison Wesley, 1997.

[4] J. M. Shridhar, S. Ravi, "Virtual Manufacturing: An Important Aspect of Collaborative Product Commerce", *Journal of Advanced Manufacturing Systems*, Vol. 1, No. 1, 2002, pp. 113-119.

[5] http://www.boeing.com/news/releases/2002.

[6] http://www.sc.army.mil/.

[7] http://www.mod.uk/aboutus/factfiles/jointforces.htm.

[8] http://www.extremeprogramming.org.

[9] http://www.dsdm.org.

[10] J. H. Reed, *Software Radio: A Modern Approach to Radio Engineering*, Prentice Hall, 2002.

[11] J. Highsmith, "Agile Software Development: A Review of Agile Methodologies," http://www.cutter.com/workshops, December, 2002.

[12] J. L. Parenti, "Engineering Software for Interoperability Use of Enterprise Architecture Techniques", *Master Thesis*, Naval Postgraduate School, March 2003.

[13] J. Puett, "Holistic Framework for Establishing Interoperability of Heterogeneous Software Development Tools", *Ph.D Dissertation* (advisor: Luqi), Naval Postgraduate School, June, 2003.

[14] K. Czarnecki, U. Eisenecker, *Generative Programming Methods, Tools, and Application*s, Addison-Wesley, 2000.

[15] L. Putnam, and W. Myers, *Industrial Strength Software*: *Effective Management Using Measurement*. IEEE Computer Society Press, 1997.

[16] V. Berzins, L. Qiao, Luqi, "Information Consistency Checking in Documentation Driven Development for Complex Embedded Systems", submitted to Monterey Workshop 2003, Chicago, USA, September 24-26, 2003.

[17] Luqi, M. Ketabchi, "A Computer-Aided Prototyping System", *IEEE Software*, March, 1988, pp. 66-72.

[18] Luqi, R. Steigerwald, et al, "CAPS as a Requirement Engineering Tool". in *Proceedings of Tri-Ada'91 International Conference*, San Jose, USA, Oct 22-25, 1991, pp. 75-83.

[19] Luqi, V. Berzins, R. Yeh, "A prototyping language for real time software", *IEEE Transactions on Software Engineering*, Vol 14, No 10, 1988, pp. 1409-1423.

[20] Luqi, Y. Qiao, L. Zhang, "Computational Model for High-confidence Embedded System Development", Monterey Workshop --- Radical Innovations of Software and Systems Engineering in the Future, October, 7-11, 2002, pp. 265-303.

[21] M. Lyu, *Software Reliability Engineering.* IEEE Computer Society Press. 1995.

[22] M. Murrah, "Enhancements and Extensions of Formal Models for Risk Assessment in Software Projects", *Ph.D Dissertation* (advisor: Luqi), Naval Postgraduate School, September, 2002.

[23] M. Saboe, "A Software Technology Transition Entropy Based Engineering Model", *Ph.D Dissertation* (advisor: Luqi), Naval Postgraduate School, March, 2002.

[24] M. Uschold, M. Gruninger, "Ontologies: Principles, Methods and Applications," *Knowledge Engineering Review*, Vol. 11, No. 2, June 1996.

[25] N. Johnson, and S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions*. Vol. 1. Wiley & Sons, 1994.

[26] P. Abrahamsson, O. Salo, J. Ronkainen, J. Warsta, "Agile Software Development Methods-Review and Analysis", *Technical Report*, ESPOO 2002.

[27] P. M. Nelson, "A Requirements Specification of Modifications to the Functional Description of the Mission Space Resource Center", *Master Thesis*, Naval Postgraduate School, June 2001.

[28] P. Young, V. Berzins, J. Ge and Luqi, "Use of Object-Oriented Model for Interoperability in Wrapper-Based Translator for Resolving Representational Differences between Heterogeneous Systems", *Monterey Workshop 2001 on Engineering Automation for Software Intensive System*, Monterey, CA, 2001, pp. 170-177.

[29] X. Liang, J. Puett and Luqi, "Perspective-based Architectural Approach for Dependable Systems", *Proc. of ICSE 2003 Workshop on Software Architectures for Depenable Systems*, Portland, OR, USA, May 3, 2003, pp. 1-6.

[30] Luqi, X. Liang, M. Brown, C. Williamson, "Formal Approach for Software Safety analysis & Risk Assessment via an Instantiated Activity Model", to appear in the 21th International System Safety Conference, August 4-8, 2003, Ottawa, Ontario, Canada.

[31] National Aeronautics and Space Administration, *NASA Œ STD Œ 8719.13A, Software Safety*, NASA Technical Standard, September 15, 1997.

[32] United Kingdom Ministry of Defense, *Ship Safety Management System™s Handbook*, JSP 430, UK.

[33] P. Young, V. Berzins, J. Ge and Luqi, "Using an Object Oriented Model for Resolving Representational Differences between Heterogeneous Systems", *Proceedings of 17th ACM Symposium on Applied Computing (SAC)*, Madrid, Spain, 10-14 March 2002, pp. 976 - 983.

[34] P. Young, "Integration of Heterogeneous Software Systems through Computer-Aided Resolution of Data Representation Differences", *Ph.D. Dissertation* (Advisor: Luqi), Naval Postgraduate School, Monterey, CA, March 2002.

[35] W. Zhao, B. Bryant, R. Raje, M. Auguston, A. Olson and C. Burt, "A Unified Approach to Component Assembly Based on Generative Programming", *Proceedings of 2002 Workshop on Generative Programming (GP 2002)*, Austin, Texas, April 2002, pp.195-199.

[36] J. Drummond, Luqi, W. Kemple, M. Auguston and N. Chaki. "Quality of Service Behavioral Model from Event Trace Analysis." *Proceedings of the 7th international Command and Control Research and Technology Symposium (CCRTS 2002),* Quebec City, Quebec, 16-20 September 2002.

[37] K. Beck et al., "Manifesto for Agile Software Development", www.agilemenifesto.org, February 2001.

[38] K. Back, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 2000.

[39] T. DeMarco, B. Boehm, "The Agile Methods Fray", IEEE Computer, Vol. 36, No. 6, 2003, pp. 90-92.

[40] Luqi, Z. Guan, "A Software Engineering Tools for Requirement Document based Prototyping", *Proceedings of the 7th World Multiconference on Systemics, Cybernetics and Infromatics*, Orlando, Florida, USA, July 27 - 30, 2003, Volume VI, pp.237-243.

Aggressive Model-Driven Development: Synthesizing Systems from Models viewed as Constraints

Tiziana Margaria[1,2] and Bernhard Steffen[2]

[1] METAFrame Technologies GmbH, Dortmund, Germany
{tmargaria}@METAFrame.de
[2] Chair of Programming Systems, University of Dortmund, Germany
{Tiziana.Margaria, Steffen}@cs.uni-dortmund.de

Position Paper

1 Motivation

The Problem

According to several roadmaps and predictions, future systems will be highly heterogeneous, they will be composed of special purpose code, perhaps written in different programming languages, integrate legacy components, glue code, and adapters combining different technologies, which may run distributed on different hardware platforms, on powerful servers or at (thin and ultra-thin) client sites. Already today's systems require an unacceptable effort for deployment, which is typically caused by incompatibilities, feature interactions, and the sometimes catastrophic behavior of component upgrades, which no longer behave as expected. This is particularly true for embedded systems, with the consequence that some components' lifetimes are 'artificially' prolonged far beyond a technological justification, since one fears problems once they are substituted or eliminated.

Responsible for this situation is mainly the level on which systems are technically composed: even though high level languages and even model driven development are used for component development, the system-level point of view is not yet adequately supported. In fact, in particular the deployment of a heterogeneous systems is still a matter of assembly-level search for the reasons of incompatibility, which may be due to minimal version changes, slight hardware incompatibilities, or simply to hideous bugs, which come to surface only in a new, collaborative context of application. Integration testing and the quest for 'true' interoperability are indeed major cost factors and major risks in a system implementation and deployment.

Hardware development faces similar problems with even more dramatic consequences: hardware is far more difficult to patch, making failure of compatibility a real disaster. It is therefore the trend of the late '90s to move beyond VLSI to

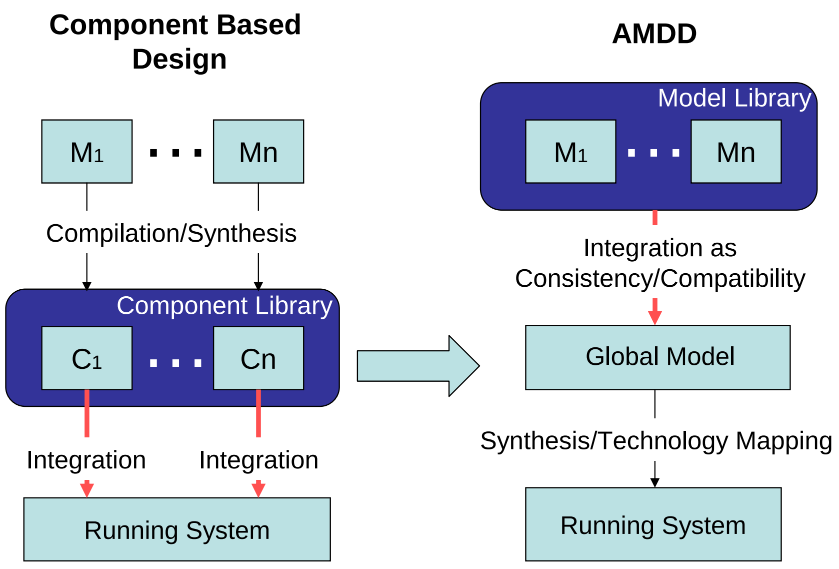


Fig. 1. The AMDD Process

Systems-on-a-Chip (SoC) to guarantee larger integration in both senses: physically, compacting complex systems on a single chip instead of on a board, but in particular also projectually, i.e. integrating the components well before the silicon level, namely at the design level: rather than combining chips (the classical way), hardware engineers start to combine directly the component's designs and to directly produce (synthesize) system-level solutions, which are homogeneous at the silicon level. Interestingly, they solve the problem of compatibility by moving it to a higher level of abstraction.

AMDD: Aggressive Model-Driven Development

At the larger scale of (embedded) system development, moving the problem of compatibility to a higher level of abstraction means moving it to the modelling level (see Fig. 1): rather than using the models, as usual in today's Component Based Development paradigm, just as a means of specification, which

– need to be compiled to become a 'real thing' (e.g., a component of a software library),
– must be updated (but typically are not), whenever the real thing changes
– typically only provide a local view of a portion or an aspect of a system,

models should be put into the center of the design activity, becoming *the* first class entities of the *global* system design process. In such an approach, as shown on the right side of Fig. 1,

– libraries should be established on the modelling level: building blocks should be (elementary) models rather than software components,
– systems should be specified by model combinations (composition, configuration, superposition, conjunction...), viewed as a set of constraints that the implementation needs to satisfy,

– global model combinations should be compiled (synthesized, e.g. by solving all the imposed constraints) into a homogeneous solution for a desired environment, which of course includes the realization of an adequate technology mapping,
– system changes (upgrades, customer-specific adaptations, new versions, etc.) should happen only (or at least primarily) at the modelling level, with a subsequent global recompilation (re-synthesis)
– optimizations should be kept distinct from design issues, in order to maintain the information on the structure and the design decisions independently of the considerations that lead to a particular optimized implementation.

With this *aggressive* style of *model-driven development* (AMDD), which strictly separates compatibility, migration, and optimization issues from model/functionality composition, it would be possible to overcome the problem of incompatibility between

– (global) models and (global) implementations, which is guaranteed and later-on maintained by (semi-) automatic compilation and synthesis, as well as between
– system components, paradigms, and hardware platforms: a dedicated compilation/synthesis of the considered *global* functionality for a specific platform architecture avoids the problems of incompatible design decisions for the individual components.

In essence, delaying the compilation/synthesis until all parameters are known (e.g. all compatibility constraints are available), may drastically simplify this task, as the individual parts can already be compiled/synthesized specifically for the current global context. In a good setup, this should not only simplify the integration issue (rather than having to be open for all eventualities, one can concentrate on precisely given circumstances), but also improve the efficiency of the compiled/synthesized implementations. In fact, AMDD has the potential to drastically reduce the long-term costs due to version incompatibility, system migration and upgrading, and lower risk factors like vendor and technology dependency. Thus it helps protecting the investment in the software infrastructure. We are therefore convinced that this aggressive style of model-driven development will become the development style at least for mass customized software in the future. In particular we believe that AMDD, even though being drastically different from state of the art industrial embedded system design, which is very much driven by the underlying hardware architecture right from the beginning, will change accordingly: technology moves so fast, and the varieties are so manifold that the classical platform-focussed development will find its limits very soon.

The Scope of AMDD

Of course, AMDD will never replace genuine software development, as it assumes techniques to be able to solve problems (like synthesis or technology mapping) which are undecidable in general. On the other hand, more than 90% of the software development costs arise worldwide for a rather primitive software development level, during routine application programming or software update, where there are no technological or design challenges. There, the major problem faced is software quantity rather than achievement of very high quality, and automation should be largely possible. AMDD is intended to address (a significant part of) this 90% 'niche'.

What does this mean? AMDD aims at making things that inherently *are* simple as simple as they should be. In particular this means that AMDD is (at least in the beginning) characterized by abstractions, neglecting interesting, but at a certain level of development unnecessary, details, like e.g. distribution of computation, methods of communication, synchronization, real time. General software development practices can be replaced here by a model and pattern-based approach, adequately restricted to make AMDD effective. The challenge for AMDD therefore is initially to characterize and then model specific scenarios where its effectiveness can be guaranteed. Typically, these will be application-specific scenarios, at the beginning rather restrictive, which will then be generalized and standardized in order to extend the scope of applicability.

Making AMDD work

In order to reach a practicable and powerful environment for AMDD there is still a long way to go:

- adequate modelling patterns need to be designed,
- new analysis and verification techniques need to be developed,
- new compilation/synthesis techniques need to be devised,
- automatic deployment procedures need to be implemented,
- systems and middleware need to be elaborated to support automatic deployment, and,
- at the meta-level, we need a theory for the adequate specification of the settings which support this style of development.

It should be noted, however, that there is an enormous bulk of work one can build upon. Thus there is room also for quick wins and early success: AMDD is a paradigm of system design, and as such, it inherently leaves a high degree of freedom in the design of adequate settings, which, as described in Section 3, can be successfully used already today.

In the following we will focus on the following main ingredients:

1. a *heterogeneous landscape of models*, to be able to capture all the particularities necessary for the subsequent adequate product synthesis. This concerns the system specification itself, the platforms it runs on together with their communication topology, the required programming style, exceptions, real time aspects, etc.

2. a rich collection of *flexible formal methods and tools*, to deal with the heterogeneous models, their consistency, and their validation, compilation, and testing.
3. *automatic deployment and maintenance support* that are integrated in the whole process and are able to provide 'intelligent' feedback in case of late problems or errors.

2 What we can build upon

2.1 Heterogeneous Landscape of Models

One of the major problems in software engineering is that software is multi-dimensional: it comprises a number of different (loosely related) dimensions, which typically need to be modelled in different styles in order to be treated adequately. Important for simplifying the software/application development is the reduction of the complexity of this multi-dimensional space, by placing it into some standard scenario. Such reductions are typically application-specific. Besides simplifying the application development they also provide a handle for the required automatic compilation and deployment procedures.

Typical among these dimensions, often also called **views**, are

– the *architectural view*, which expresses the static structure of the software (dependencies like nesting, inheritance, references). This should not be confused with the architectural view of the hardware platform, which may indeed be drastically different. - The charm of the OO-style was that it claimed to bridge this gap.
– the *process view*, which describes the dynamic behavior of the system. How does the system run under which circumstance (in the good case)
– the *exception view*, which addresses the system's behavior under malicious or even unforeseen circumstances
– the *timing view*, addressing real time aspects
– the various *thematic views* concerned with roles, specific requirements, ...

Of course, UML tries to address all these facets in a unifying way, but we all know that UML is currently rather a heterogeneous, expressive sample of languages, which lacks a clear notion of (conceptual) integration like consistency and the idea of global dynamic behavior. Such aspects are dealt with currently independently e.g. by means of concepts like *contracts* [1] (or more generally, and more complicatedly, via business-rules oriented programming like e.g. in [6]). The latter concepts are also not supported by systematic means for guaranteeing consistency. In contrast, AMDD views these heterogeneous specifications (consisting of essentially independent models) just as constraints which must be 'solved' during the compilation/synthesis phase (see also [13]).

Another recently very popular approach is Aspect Oriented Programming (AOP) [7, 2], which sounds convincing at first, but does not seem to scale for realistic systems. The programmer treats different aspects separately in the code, but has to understand precisely the weaving mechanism, which often is more complicated than programming all the system traditionally. In particular, the claimed modularity is only in the file structure but not on the conceptual side. In other words, in the good case one can write down the aspects separately, but understanding their mutual global impact requires a deep understanding of weaving, and, even worse, of the result of weaving, which very much reminds of an interleaving expansion of a highly distributed system.

2.2 Formal Methods and Tools

There are numerous formal methods and tools addressing validation, ranging from methods for correctness-by-construction/rule-based transformation, correctness calculi, model checkers, and constraint solvers to tools in practical use like PVS, Bandera, SLAM to name just a few. On the compiler side there are complex (optimizing) compiler suites, code generators, and controller synthesizers, and other methods to support technology mapping. A complete account of these methods would be far beyond the purpose of this paper. Here it is sufficient to note that there is already a high potential of technology waiting to be used.

2.3 Automatic Deployment and Maintenance Support

At the moment, this is the weakest point of the current practice: the deployment of complex systems on a heterogeneous, distributed platform is typically a nightmare, the required system-level testing is virtually unsupported, and the maintenance and upgrading very often turn out to be extremely time consuming and expensive, de facto responsible for the slogan "never change a running system".

Still, also in this area there is a lot of technology one can build upon: the development of Java and the JVM or the dotnet activities are well-accepted means to help getting models into operation, in particular, when heterogeneous hardware is concerned. Interoperability can be established using CORBA, RMI, RPC, Webservices, complex middleware etc, and there are tools for testing and version management. Unfortunately, using these tools requires a lot of expertise, time to detect undocumented anomalies and to develop patches, and this for every application to be deployed.

3 A Simple AMDD-Setting

The Application Building Center (ABC) developed at METAFrame Technologies in cooperation with the University of Dortmund is intended to promote the AMDD-style of development in order to move the application development for certain classes of applications towards the application expert. Even though the ABC should only be regarded as a first step of AMDD development, it already comprises some important AMDD-essentials (Fig. 2.3):

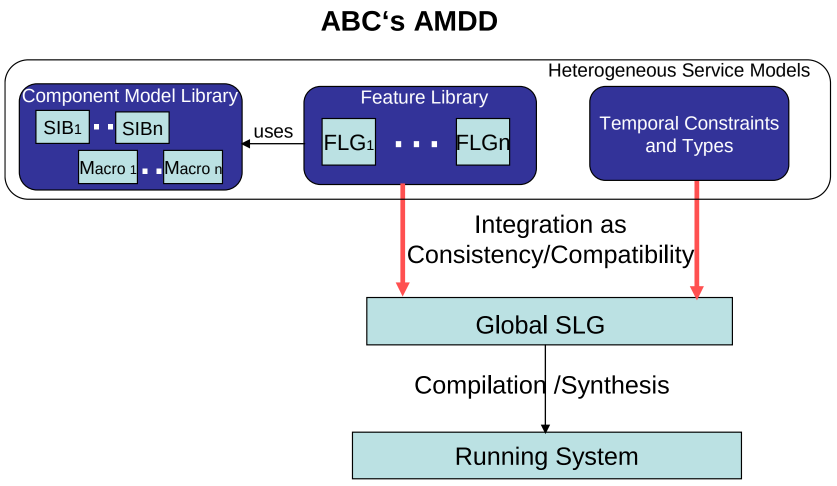


Fig. 2. The AMDD Process in the ABC

1. *Heterogeneous landscape of models*: the central model structure of the ABS are hierarchical Service Logic Graphs (SLGs)[14, 9]. SLGs are flow chart-like graphs. They model the application behavior in terms of the intended process flows, based on coarse granular building blocks called SIBs (Service-Independent Building blocks) which are intended to be understood directly by the application experts [14] – independently of the structure of the underlying code, which, in our case, is typically written in Java/C/C++. The component models (SIBs or hierarchical subservices called Macros), the feature-based service models called Feature Logic Graphs (FLGs), and the Global SLGs modelling applications are all hierarchical SLGs.

   Additionally, the ABC supports model specification in terms of

   (a) two modal logics, to abstractly and loosely characterize valid behaviors (see also [5]),
   (b) a classification scheme for building blocks and types, and
   (c) high level type specifications, used to specify compatibility between the building blocks of the SLGs.

   The granularity of the building blocks is essential here as it determines the level of abstraction of the whole reasoning: the verification tools directly consider the SLGs as formal models, the names of the (parameterized) building blocks as (parameterized) events, and the branching conditions as (atomic) propositions. Thus the ABC focusses on the level of *component composition* rather then on component construction: its compatibility, its type correctness, and its behavioral correctness are under formal methods control [9].
2. *Formal methods and tools*: the ABC comprises a high-level type checker, two model checkers, a model synthesizer, a compiler for SLGs, an interpreter, and a view generator. The model synthesizer, the model checkers and the type checker take care of the consistency and compatibility conditions expressed by the four kinds of constraints/models mentioned above.

3. *Automatic deployment and maintenance support*: an automated deployment process, system-level testing [10], regression testing, version control, and online monitoring [3] support the phases following the first deployment.
   In particular the automatic deployment service needs some meta-modelling in advance. In fact, this has been realized using the ABC itself. Also the testing services and the online monitoring are themselves strong formal methods-based [11] and have been realized via the ABC.

In this sense, the ABC can be regarded as a simple and restrictive but working AMDD framework. In fact, in the ABC, composition/coordination of components as well as their maintenance and version control happen exclusively at the modelling level, and the compilation to running source code (mostly Java and C++) and deployment of the resulting applications are fully automatic.

4 Conclusions and Perspectives

We have proposed an aggressive version of model-driven development (AMDD), which moves most of the recurring problems of compatibility and consistency from the coding and integration to the modelling level. Of course, AMDD requires a complex preparation of adequate settings, where the required compilation and synthesis techniques can be realized. Still, the effort to create these settings and their (application dependent) restrictions can be easily paid off by immense cost reductions in software mass construction and maintenance. In fact, besides reducing the costs, aggressive model-driven development will also lead (more or less automatically) to a kind of normed software development, making software engineering a true engineering activity.

This direction is also consistent with the perspective indicated by the joint GI-ITG position paper on *Organic Computing*[1] [12]: the blurring of borders between hardware and software (machines and programs) that initiated with embedded systems and with hardware/software codesign is going to reach a completely new dimension, where

– the systems are conceived, designed and implemented in terms of *services*,
– they are provided and used in a virtual space, and where
– the distinction on where (local, global, at which node, on which hardware) and how (hardware, software, network, ...) the services are available is relatively inessential information.

In particular, according to availability or convenience, the provider of services can be changed and the provision of services is not a permanent contract anymore.[2]

[1] GI, the Gesellschaft für Informatik and ITG, the Informationstechnikgesellschaft im VDE, the Verband der Elektrotechnik, Elektronik und Informationstechnik are the German counterparts of the ACM and IEEE, respectively.

[2] This is a scenario that concretizes the idea of Sentient Computing [4].

We are convinced that this aggressive style of model-driven development - which overcomes the problem of compatibility between model and implementation, as well as between system components, paradigms and hardware platforms - will become the development style for most of the applications in the future. AMDD is a paradigm of system design, and as such it inherently leaves a high degree of freedom in the design of adequate settings. In particular, we do not expect a single solution to emerge, but rather a collection of environments and settings optimized and tailored for this design paradigm in a number of relevant areas of application.

In particular, we envisage a coordinative design paradigm similar to the already successful paradigm of *feature-oriented design* [5]: in that setting, widely adopted in the telecommunication industry, systems are composed of a thin skeleton of basic functionality, enriched at need and on demand via additional features that deliver premium functionality (services) to the customers/end-users. A well studied example is the combination of POTS (Plain Old Telephone Service) functionality as a basic telephony service provided by a switch, enriched and virtualized by features like Call Forwarding, Conference Call, Collect Billing etc. In feature-oriented design, the structure that matters is not the technical structure (objects, classes) of and under the system, but rather the *structure of the application-domain* (what does the system do for me, when and under which conditions), together with the capability of mapping the what into the how and of changing the how on the fly. Indeed, the Intelligent Network standard is defined in this optic: it defines which features exist in that application domain and what they deliver to the user, and it says nothing about implementational issues, which are left free to the single vendors.

Even though it is only a very first step, we consider the ABC a kind of proof of concept motivating the design of more elaborate aggressive model-based development techniques. In fact, we have already reapt the benefits of this modelling style in one of our projects, in the Integrated Testing Environment projects (with Siemens ICN, Witten (D)). In an initial project phase we built a system-level test environment for complex Computer-Telephony Integrated applications that covered client-server third party application interoperating with telecommunication switches and communicating over a LAN [10]. In a second phase we were faced with the problem of the next generation of applications, that from the engineering point of view had a completely different, and much more complex, profile: we needed to capture internet-based applications that online, role-based, and remotely (over internet) reconfigure e.g. the complete routing and call management settings on a virtual switch implemented as a fault tolerant cluster of physical switches [8]. This meant a new quality of complexity along at least three dimensions: testing over the internet, testing virtual clusters, and testing a controlling system in a non-steady state (during reconfiguration). Thanks to our AMDD approach, this did not affect at all the *conceptual type* of the models we used in the ITE! Thus we were able to help the Siemens engineers to solve their new problem within the existing modelling framework, just by compatibly extending the libraries of models. In particular, they could reuse SIBs, features and SLGs from the previous project phase with no change.

References

1. L.F. Andrade, J.L. Fiadeiro: *Architecture Based Evolution of Software Systems*, http://www.atxsoftware.com/publications/SFM.pdf.
2. AspectJ Website: http://eclipse.org/aspectj/
3. A. Hagerer, H. Hungar, O. Niese, and B. Steffen: *Model Generation by Moderated Regular Extrapolation*. Proc. of the 5th Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2002), Grenoble (F), LNCS 2306, pp. 80-95.
4. A. Hopper: The Royal Society Clifford Paterson Lecture: *Sentient Computing*, 1999.
5. B. Jonsson, T. Margaria, G. Naeser, J. Nyström, and B. Steffen. Incremental requirement specification for evolving systems. Nordic Journal of Computing, vol. 8(1):65, Also in *Proc. of Feature Interactions in Telecommunications and Software Systems 2000*, 2001.
6. JRules, ILOG. http://www.ilog.com/
7. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, J.Irwin: Aspect-Oriented Programming. Proc. of ECOOP, Springer-Verlag (1997).
8. T. Margaria, O. Niese, B. Steffen, A. Erochok: *System Level Testing of Virtual Switch (Re-)Configuration over IP*, Proc. IEEE European Test Workshop, Corfu (GR), May 2002, IEEE Society Press.
9. T. Margaria, B. Steffen: *Lightweight Coarse-grained Coordination: A Scalable System-Level Approach*, to appear in STTT, Int. Journal on Software Tools for Technology Transfer, Springer-Verlag, 2003.
10. O. Niese, T. Margaria, A. Hagerer, M. Nagelmann, B. Steffen, G. Brune, and H. Ide. An automated testing environment for CTI systems using concepts for specification and verification of workflows. *Annual Review of Communication*, Int. Engineering Consortium Chicago (USA), Vol. 54, pp. 927-936, IEC, 2001.
11. O. Niese, B. Steffen, T. Margaria, A. Hagerer, G. Brune, and H. Ide. Library-based design and consistency checks of system-level industrial test cases. In H. Hußmann, editor, *Proc. FASE 2001*, LNCS 2029, pages 233–248. Springer Verlag, 2001.
12. *Organic Computing: Computer- und Systemarchitektur im Jahr 2010*, position paper of the VDE/ITG/GI. http://www.gi-ev.de/download/VDE-ITG-GI-Positionspapier
13. B. Steffen. Unifying models. In R. Reischuk and M. Morvan, editors, *Proc. STACS'97*, LNCS 1200, pages 1–20. Springer Verlag, 1997.
14. T. Margaria, B. Steffen: *METAFrame in Practice: Design of Intelligent Network Services*, in "Correct System Design - Issues, Methods and Perspectives", LNCS 1710, Springer-Verlag, 1999, pp. 390-415.

New Development Techniques--New Challenges for Verification and Validation

Mats Heimdahl, *University of Minnesota*

Abstract:

The new thrust towards model-based, or specification centered, development techniques promises to reduce development costs and increase software quality. Model-based techniques center the development effort around one or more high-level models of the system of interest, and relies heavily on tools for visualization, analysis, and code generation---tools that will largely replace tasks that were previously done manually. This transition from manual development to predominantly automated development relying on tools raises new and challenging verification and validation problems, in particular when developing critical software systems that require certification. This talk will discuss the verification and validation challenges arising from the increased reliance on tools and point out directions for future research efforts.

Software Architectures and Embedded Systems

Nenad Medvidovic Sam Malek Marija Mikic-Rakic
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781
{neno,malek,marija}@usc.edu

Introduction

Software architecture has emerged as an area of intense research over the past decade [25,32]. A number of approaches have been proposed to deal with architectural description and analysis [21], architectural styles [8], domain-specific and application family architectures [4,35], architecture-based dynamic system adaptation [29], and so forth. By and large, however, these approaches share assumptions that make them suited specifically to the domain of traditional, desktop-based, possibly distributed development platforms. Those (comparatively few) architecture-based solutions that have focused on software systems for embedded devices (e.g., [28]) have had to face some of the same challenges (e.g., applying solutions across an application family), but also appear to have had some different priorities (e.g., ensuring efficient, architecture-compliant system implementations).

The goal of this paper is to draw some general distinctions between "traditional" software architectures and those targeted at embedded systems. We focus on several areas that traditional software architecture research has studied to date and discuss their applicability and potential shortcomings in the context of embedded systems. As suggested above, our position is informed by the existing literature. Additionally, we will leverage insights drawn from a graduate course we have offered at USC for the past two years—*Software Engineering for Embedded Systems* [35]. Finally, we will also rely on the experience from *Prism*, an on-going research project whose goal is to develop software architectural solutions for the domain of highly distributed, mobile, resource-constrained, and possibly embedded computation [19,23]. The discussion provided in this paper should not be considered a definitive study of this issue, but rather a starting point for future discussions. We also attempt to provide a critical assessment of our Prism project with respect to the discussed areas, in the hope of outlining future directions that will improve Prism's suitability for the embedded systems domain.

Architectural Modeling

A large number of special-purpose architecture description languages (ADLs) [21] have been developed to represent different aspects of software architectures. More recently, the Unified Modeling Language (UML) [3] has gained widespread acceptance for a similar purpose. Table 1 shows an overview of several ADLs and their primary foci. Only two of these notations are specifically intended for the embedded systems domain:

1. MetaH models architectures in the guidance, navigation, and control (GN&C) domain. MetaH tries to ensure the schedulability, reliability, and safety of the modeled software system. It also considers the availability and properties of hardware resources.
2. Weaves [10] supports specification of data-flow architectures. In particular, Weaves is specialized to support real-time processing of large volumes of data emitted by weather satellites.

Table 1: – ADL Scope and Applicability

ADL	ACME	Aesop	C2	Darwin	MetaH	Rapide
Focus	Architectural interchange at the structural level	Specification of architectures in specific styles	Architectures of distributed, evolvable systems in a particular style	Architectures of distributed, dynamic systems	Guidance, navigation, and control system architectures	Modeling and simulation of the dynamic behavior of an architecture

ADL	ROOM	SADL	UniCon	Weaves	Wright
Focus	Graphical (structural and behavioral) models of architectures	Formal refinement of architectures across levels of detail	Glue code generation for interconnecting existing components	Data-flow architectures with real-time requirements on data processing	Modeling and deadlock analysis of concurrent systems

Two other notations also deal with issues that are relevant to the embedded systems domain: UniCon [32] supports modeling of runtime (though not necessarily real-time) scheduling, while ROOM [31] targets real-time computation with a combination of message sequence charts and state charts. The more recent Avionics ADL [5] tries to marry several of these ideas into a single language.

Despite the above examples, many questions remain unanswered. It is unclear whether modeling the architectures of embedded systems is inherently incompatible with semi-formal notations such as UML. The prevailing characteristics of the embedded systems domain would suggest that rigor and formality are a non-negotiable requirement. On the other hand, counter-examples exist. For instance, the software architecture team working on JPL's Mission Data Systems (MDS) architectural framework [39] had initially selected UML for representing MDS architectures; the team has recently replaced UML with an XML-based ADL [8], but one that still has only semi-formally defined semantics.

Another relevant issue is deciding which aspects of an embedded software system are critical from an architectural perspective. It is widely agreed that components, connectors, and their configurations are the basic building blocks of a traditional software architecture. While the same claim might be made about embedded system architectures, embedded systems have certain characteristics that require careful consideration. For example, a to-be-embedded software system is often built to a specification, with the actual platform(s) being unknown or even non-existent at the time of development. In such cases, an application domain-specific ADL (e.g., MetaH for the GN&C domain) may help in identifying at least those system aspects that are likely to remain stable across specific applications and platforms.

We have not focused on architectural modeling in the Prism project. Instead, we have chosen to rely on our existing architecture modeling infrastructure [20] in order to address other issues, as discussed below.

Analysis

A particular focus of existing ADLs has been on formal analysis of system properties at the architectural level. However, ADLs are hampered by two problems, both of which are likely to be further magnified in the case of embedded systems. The first issue is analysis *fidelity*: while languages such as Wright [1] allow modeling and sophisticated analyses of dynamic system properties (e.g., potential for deadlocks), they either make large numbers of simplifying assumptions about, or fail to consider altogether, the capacity, speed, power, and other properties of the hardware platforms on which the modeled software systems will execute. In addition, most existing ADLs provide scant support for transferring the desired architectural decisions into source code. This is absolutely critical in the case of embedded systems, where an elegant and "correct" software architecture will be of little use unless it results in an effective and efficient *running* system.

The lack of analysis fidelity suggests the second issue that has not been adequately addressed in existing ADLs. In order to gain confidence that the system will behave correctly in its target environment, *simulation* of that system model's execution behavior in the (simulated) environment becomes indispensable. While there is a lot of potentially relevant work on simulation, this work has taken place almost entirely outside the domain of software architectures. The lone exception to this is Darwin [16] which leverages its π-calculus underpinnings to support execution of "what if" scenarios. Much additional work is needed if ADLs are to be rendered suitable for use with embedded systems in this regard.

As in the case of modeling, thus far in the Prism project we have not particularly focused on architectural analysis of embedded system architectures.

Architectural Styles and Reference Architectures

Software *architectural styles* are recurring patterns of system organization whose application results in systems with known (desirable) properties [9,33]. As such, styles are key software design idioms. Examples of well known styles are layered, pipe-and-filter, client-server, push-based, peer-to-peer, event-based, and so forth. At the same time, very little is known about the applicability of these, or any other styles, to the embedded systems domain. There are a few exceptions to this (e.g., [12,38]), but they have emerged from problem domains (e.g., mobile robotics) in which software engineering issues, and software architectures in particular, are considered to be of secondary importance.

An issue related to styles is that of *reference architectures*. A reference architecture is applicable to systems across an application family and/or problem domain. Unlike a style, which provides a set of heuristics for arriving at a software system's architecture, a reference architecture only needs to be instantiated into a system architecture (i.e., it is already an architecture, but a generic one). Even though effective styles for embedded systems may be unknown, there are examples of successful reference architectures in this area. One such example is Phillips's Koala project [28], which is targeted at consumer electronics devices. Another example is IBM's ADAGE [6] reference architecture (illustrated in Figure 1 below), targeted at the avionics domain. Reference architectures are hard to come by because they require (extensive) existing experience within an application domain, but, once in existence, they do appear to be a natural fit with embedded systems.

We have placed special emphasis on architectural styles in the Prism project. Since, as discussed above, there is currently very little understanding of styles that are appropriate in the embedded systems setting, we have developed an architectural style *framework*, called *Prism-SF* [19], which may be instantiated into a number of individual styles. The framework fixes a small number of architectural notions: computation is performed within (autonomous) *components*, and their interaction enabled via *events*. All other architectural concepts (e.g., software *connectors*, communication *ports*) are available and configurable, but not required. Finally, Prism-SF allows the

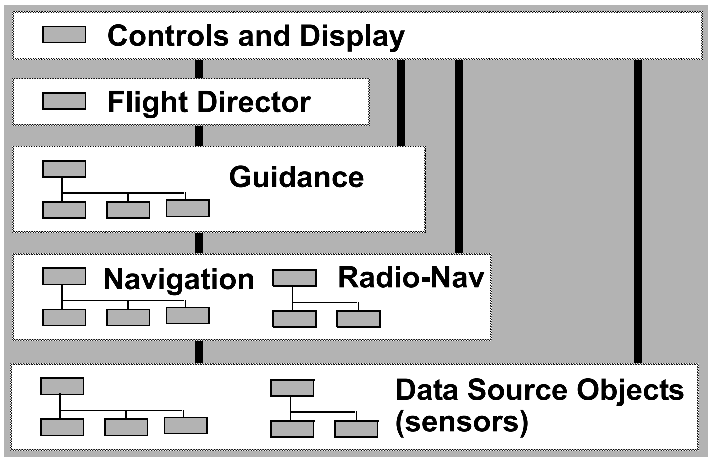


Figure 1. ADAGE – a five-layer reference architecture intended for systems in the avionics domain. Every ADAGE-compliant system will have the depicted five layers. Each of the five layers may, in turn, comprise its own layered internal architecture.

selection and combination of four types of interaction: symmetric (i.e., peer-to-peer) and asymmetric (i.e., master-slave), as well as synchronous and asynchronous. These facilities provided by Prism-SF are instantiated into one or more architectural styles, which are, in turn, used in the design of specific systems. For illustration, Figure 2 shows a partial application architecture designed using an instance of Prism-SF in which

- both symmetric and asymmetric connectors are included;
- each component has single master, slave, and peer communication ports; and
- symmetric and asymmetric connectors may not be attached to one another.

Implementation Support

Software architectures provide design-level models and guidelines for composing software systems. For these models and guidelines to be truly useful in a development setting, they must be accompanied by support for their implementation [18,32]. This is particularly important in the

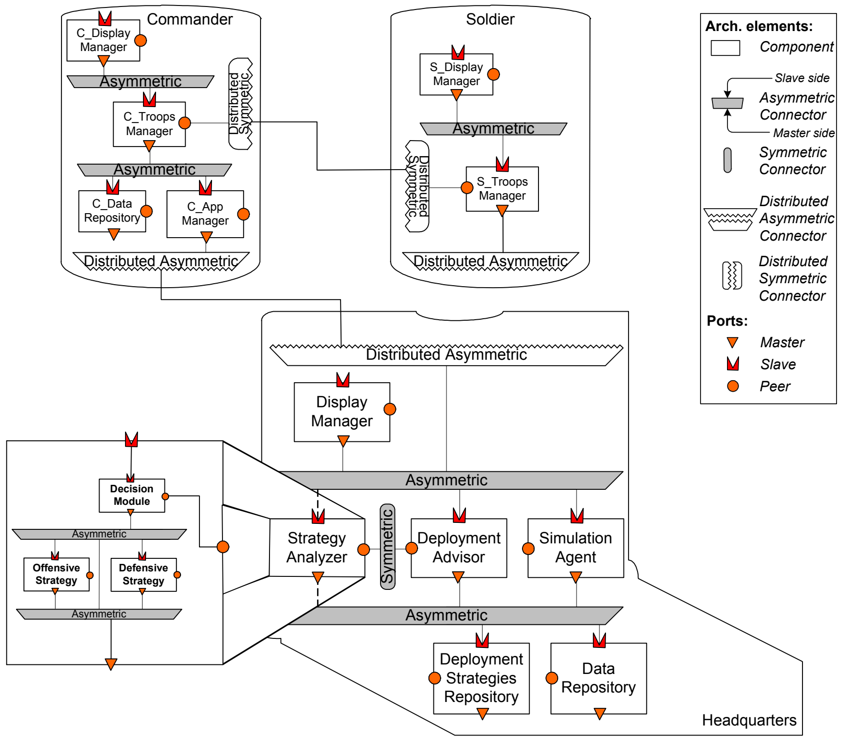


Figure 2. An application architecture designed using a particular instantiation of the Prism-SF architectural style framework.

context of embedded systems: they may be highly distributed, decentralized, mobile, and long-lived large-scale software systems, increasing the risk of architectural drift [25] unless there is a clear relationship between the architecture and its implementation. Recent studies [23,32] have shown that an effective way to realizing the system's architecture is to leverage the support provided by architectural middleware solutions. Typically an architectural middleware provides implementation-level support for the key aspects of system's architecture: components, connectors, architectural configurations, and communication events.

Embedded systems are usually characterized by resource constraints. However, middleware solutions introduce an abstraction layer and therefore raise the issue of its effect on the system's performance. For a middleware platform to be usable in the context of embedded systems, it needs to be highly efficient. Depending on the nature of the embedded environment, efficiency can entail minimum use of CPU, memory, battery power, network bandwidth, and so on. Recently, some vendors of popular middleware solutions (e.g., CORBA Orbix [13]) have started to tailor their support for use in the embedded systems domain. Table 2 below compares middleware solutions for mobile, resource constrained, possibly embedded systems along several pertinent dimensions. The table has been adapted from [23].

Table 2: Comparison of existing middleware solutions. **?** denotes unavailable data; ✓✓✓ denotes extensive support; ✓✓ denotes solid support; ✓ denotes some support; empty cells denote no support.

Property	*Orbix/E* [13]	*TAO* [30]	*JXTA* [26]	*.NET* [22]	*JINI* [34]	*XMIDDLE* [17]	*RCSM* [40]	*LIME* [15]	*Prism-MW* [23]
Architectural abstractions									✓✓✓
Efficiency [a]	16.6K	8K	**?**	**?**	**?**	**?**	**?**	**?**	20K
	95KB	0.5MB	**?**	**?**	**?**	156KB	**?**	**?**	4.6KB
Scalability	✓✓	✓✓	✓✓	✓✓	✓✓	**?**	**?**	**?**	✓✓✓
Delivery guarantees	✓	✓✓✓							✓
Mobility					✓✓	✓✓✓	✓✓	✓✓✓	✓✓
Reconfigurability		✓✓			✓✓		✓✓	✓✓	✓✓
Security	✓✓	✓✓	✓✓	✓✓	✓		✓✓		✓

a. Number of events per second (top) and memory usage (bottom).

Heterogeneity is another intrinsic characteristic of embedded systems [14,19]: unlike traditional software platforms, which have been standardized to a significant degree, many embedded systems run on one-of-a-kind hardware with special-purpose operating systems, programming languages, network protocols, data formats, and so forth. This poses a great challenge to embedded application developers. Techniques commonly employed to address heterogeneity in traditional software systems, such as XML encoding or platform independent programming languages (e.g., Java), also may not be viable options in the embedded systems domain due to resource scarcity and possible real-time requirements. Therefore, a practical architectural middleware solution in this domain needs to be either highly specialized (and thus narrowly applicable) or flexible in order to overcome the unpredictable and the heterogeneous nature of embedded systems.

Our implementation support for software architectures in the Prism project is embodied in the *Prism-MW* middleware platform [23]. Prism-MW provides implementation-level support for key abstractions provided by the Prism-SF architectural style framework discussed above. Prism-

Architecture initialization

```
class DemoArch {
  static public void main(String argv[]) {
  Architecture arch = new Architecture ("DEMO");
    // create components here
    ComponentA a = new ComponentA ("A");
    ComponentB b = new ComponentB ("B");

    // create connectors here
    Connector conn = new Connector("Conn");

    // add components and connectors to the architecture
    arch.addComponent(a);
    arch.addComponent(b);
    arch.addConnector(conn);

    // establish the interconnections
    arch.weld(a, conn);
    arch.weld(b, conn);
    arch.start();
  }
}
```

Component A sends an event

```
  Event e = new Event (“Event_a”);
  e.addParameter("param_1", p1);
  send (e);
```

Component B handles the event and sends a response

```
public void handle(Event e)
{
  if (e.equals(“Event_a”)) {
    ...
    Event e1= new Event(“Response_to_a”);
    e1.addParameter("response", resp);
    send(e1);
  }...
}
```

Figure 3. Illustration of application implementation fragments in Prism-MW. The created simple architecture has two components, *A* and *B*, communicating via a connector, *Conn*.

MW is, at the same time, optimized to exhibit a small memory footprint and good system speed (see Table 2), and extensible to address many relevant concerns, including distribution, mobility, security, data compression, and so forth. However, Prism-MW currently does not consider other hardware resources, such as battery power, or availability and properties of peripheral devices. Figure 3 illustrates how an architecture is “programmed” in the Java version of Prism-MW.

Deployment Support

An embedded software system is typically developed and tested in a simulated environment. The target hardware environment is frequently produced in parallel with the software system, and therefore may not be available before or during the software system’s development and testing. Alternatively, the actual target environment may be too expensive to replicate or it may be too distant (e.g., a space probe). However, the characteristics of the target environment directly influence certain software decisions, such as the distribution of software components onto hardware hosts (i.e., the system’s deployment architecture). Furthermore, the target environment often changes during the system’s execution (e.g., due to the mobility of hardware hosts). As a result of this, there is an increasing demand for deployment support that can assist with the installation and/or update of a software system. Traditional approaches to software deployment have often required sophisticated support (e.g., deployment agents [11]). These approaches have typically included their own facilities to inspect the target environment prior to deployment. Since these facilities are provided separately from the application’s implementation infrastructure, they introduce additional overhead to the target host. Therefore, these approaches are usually not directly applicable for resource constrained, embedded software systems.

Mainstream software deployment solutions have comprised large-scale "patches" that replace an entire application or set of applications (e.g., new versions of MS Office). This kind of coarse-grain deployment does not provide sufficient control over the deployment process and is usually not applicable to distributed embedded systems with low-bandwidth network connections through which the patches need to be exchanged. For these reasons, efficient, fine-grain control over the deployment process is required in the context of embedded systems.

Finally, in highly distributed embedded systems, deployment and/or update process may need to be initiated from multiple sites, each of which is in charge (and possibly aware) of only a part of the overall system. Furthermore, the source locations of software components that need to be installed or updated may be themselves distributed, requiring efficient support for exchanging the necessary software components between the source and destination hosts.

Current support for deployment in the Prism project is accomplished via an extension to MS Visio and a "skeleton" configuration in Prism-MW consisting of a special purpose *Admin* component and a distribution-enabled connector. The resulting tool, *Prism-DE*, is shown in Figure 4. Prism-DE allows one to specify a configuration of hardware hosts (with known IP addresses), operating system processes on each host, and software configurations (comprising software components and connectors) within each process. Once a suitable deployment is depicted, it is effected with a single button push. The *Admin* components on each host receive and instantiate (using Prism-MW's API shown in Figure 3) the application-specific components. Further details of this process are discussed in [24]. Prism-DE then continuously monitors the network connectivity of the depicted hardware configuration. As indicated in the above discussion, Prism-DE currently sup-

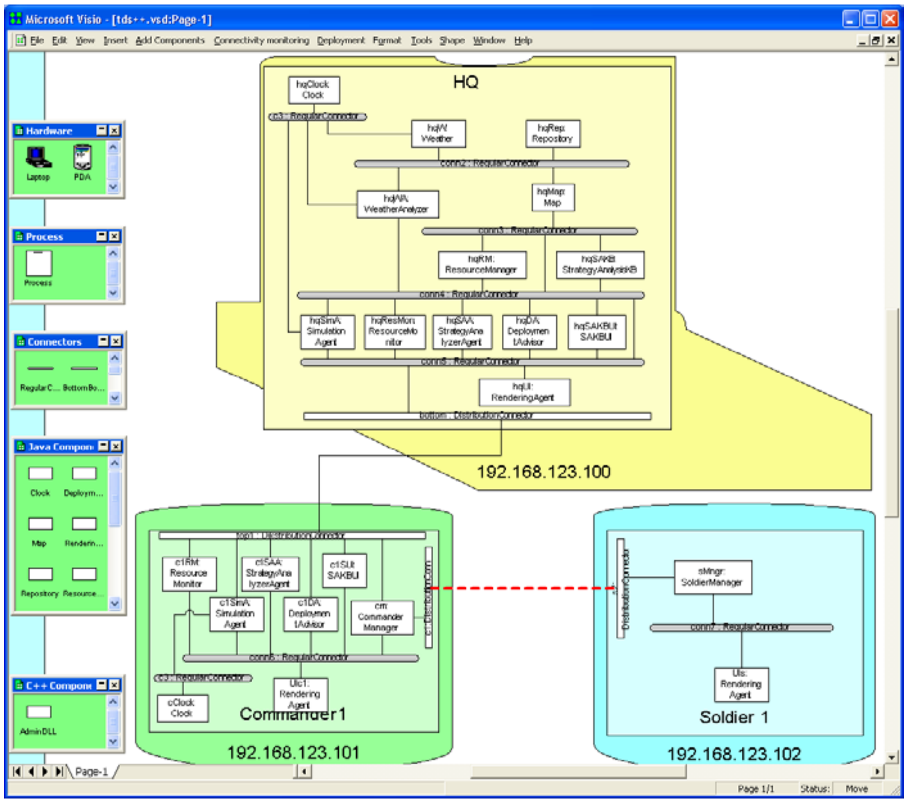


Figure 4. Screenshot of the Prism-DE deployment environment. The dotted line between the two bottom device icons denotes a network disconnection.

ports only *centralized* deployment, in which the entire system's deployment architecture, as well as the locations of all component implementations are known *a priori*.

Dynamic Adaptability

Many embedded systems are safety-critical systems that concurrently engage the physical world. These systems must be capable of adapting to changes in their execution environment. However, usually these systems cannot be brought down for updates [32]. Dynamic adaptability provides a solution to this problem, as it enables one to modify a running software system without stopping its execution. However, system properties (e.g., availability, safety) may get affected during the system's dynamic manipulation and therefore need to be accounted for when performing the adaptation. This can be achieved by analyzing (both statically [1] and dynamically [27]) the likely effects of the proposed changes before they are enacted.

Dynamic adaptation may modify the system's architecture. These modifications need to be captured and maintained at the architectural level to ensure consistency between the architectural model and the running system. Depending on the origin of changes and the degree of distribution of the software system, the task of maintaining the consistency may be trivial (i.e., updating the architectural model before initiating the change or after the change is completed) or highly complex (if there are many changes on many target hosts, the information about these changes needs to be propagated to the host maintaining the architectural model). Furthermore, in decentralized embedded systems there may not be a single host capable of maintaining the system's overall architectural model. We are aware of no existing solutions for ensuring correct, consistent, and safe dynamic adaptability of such systems.

As indicated in the above discussion, the current support for dynamic system adaptability in Prism assumes a centralized architectural model of the system. We leverage Prism-MW's API for adding, removing, and reconnecting components in an architecture (recall Figure 3), as well as programming language-level facilities (e.g., dynamic class loading in Java) and operating system-level facilities (e.g., DLLs in Windows), to enable dynamic system manipulation at the software component level. While system analysis prior to dynamic change has not been a particular focus of our work to date, as a "proof of concept" we have provided special purpose *Architectural Analysis* components within Prism-MW. These components implement a variation of our static analysis capabilities for software architectures [20].

Conclusion

It has been claimed that software architectures have the potential to revolutionize large-scale software development by allowing developers to shift their focus away from a system's implementation details. However, in the case of embedded systems, the devil is indeed in the details: a number of implementation-level issues have direct implications on a system's success and even its viability. These issues, therefore, must be captured and properly assessed at the level of an embedded system's software architecture; otherwise the architecture will be of little use. In this paper, we have used the traditional "playing field" of software architectures to highlight some of the unique challenges introduced by the embedded systems domain and briefly introduce emerging techniques to address these challenges; we have also provided an evaluation of our Prism project with respect to these challenges. We hope to have provided a useful first step in what is surely a topic worthy of further study.

References

[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, pp. 213-249, July 1997.

[2] D. Batory, L. Coglianese, S. Shafer, and W. Tracz. The ADAGE Avionics Reference Architecture. *A1AA Computing in Aerospace-10*, San Antonio, Texas, March 28-30 1995.

[3] G. Booch, J. Rumbaugh, I. Jacobson. The Unified Modeling Language User Guide. *Addison Wesley*, 1999.

[4] J. Bosch. Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. *Addison-Wesley (Pearson Education)*, May 2000.

[5] E. Colbert, B. Lewis, and S. Vestal. Developing Evolvable, Embedded, Time-Critical Systems with MetaH. *34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 34)*, Santa Barbara, August 2000.

[6] L. Coglianese and R. Szymanski, DSSA-ADAGE: An Environment for Architecture-based Avionics Development, *In Proceedings of AGARD,* 1993.

[7] D. Daniel, R. Rasmussen, G. Reeves, A. Sacks. Software Architecture Themes In JPL's Mission Data System. *AIAA Space Technology Conference and Exposition*, Albuquerque, NM, September 1999.

[8] E. M. Dashofy, A. Van der Hoek, R. N. Taylor. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. *24the International Conference on Software Engineering*, Orlando, Florida, May 2002.

[9] R. T. Fielding. Architectural Styles and the Design of Network-Based Software Architectures. *Ph.D Thesis, University of California Irvine*, June 2000.

[10] M. M. Gorlick and R. R. Razouk. Using Weaves for Software Construction and Analysis. *13th International Conference on Software Engineering*, Austin, TX, May 1991.

[11] R. S. Hall, D. M. Heimbigner, and A. L. Wolf. A Cooperative Approach to Support Software Deployment Using the Software Dock. *21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.

[12] B. Hayes–Roth et. al. A Domain-Specific Software Architecture for Adaptive Intelligent Systems. *IEEE Transactions on Software Engineering*, Vol. 21, No. 4, April 1995.

[13] IONA Orbix/E Datasheet. *http://www.iona.com/whitepapers/orbix-e-DS.pdf*

[14] E. A. Lee. Embedded Software. *Advances in Computers* (Marvin V. Zelkowitz, ed.), Vol. 56, Academic Press, London, 2002.

[15] LIME. http://lime.sourceforge.net/

[16] J. Magee, J. Kramer. Dynamic structure in software architectures. *4th ACM SIGSOFT symposium on Foundations of software engineering*, San Francisco, CA, October 1996.

[17] C. Mascolo et. al. XMIDDLE: A Data-Sharing Middleware for Mobile Computing. *Personal and Wireless Communications*, Kluwer, April 2002.

[18] N. Medvidovic, N. R. Mehta, M. Mikic-Rakic: A Family of Software Architecture Implementation Frameworks. *The Working IEEE/IFIP Conference on Software Architecture 2002*, Montreal, Canada, August 2002.

[19] N. Medvidovic, M. Mikic-Rakic, N. Mehta, S. Malek. Software Architectural Support for Handheld Computing. *IEEE Computer, special issue on handheld computing,* September 2003.

[20] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. *21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.

[21] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, January 2000.

[22] Microsoft .NET. http://www.microsoft.com/net/

[23] M. Mikic-Rakic and N. Medvidovic. Adaptable Architectural Middleware for Programming-in-the-Small-and-Many. *ACM/IFIP/USENIX International Middleware Conference*, Rio de Janeiro, Brazil, June 2003.

[24] M. Mikic-Rakic and N. Medvidovic. Architecture-Level Support for Software Component Deployment in Resource Constrained Environments. *Component Deployment, IFIP/ACM Working Conference*, Berlin, Germany, June 20-21, 2002.

[25] D.E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture**.** *ACM SIGSOFT Software Engineering Notes*, Vol. 17, No.4, pages 40-52, October 1992.

[26] Project JXTA. http://www.jxta.org/

[27] M. Rakic, N. Medvidovic. Increasing the Confidence in Off-the-Shelf Components: A Software Connector-Based Approach. *2001 Symposium on Software Reusability (SSR 2001)*, Toronto, Canada, May 2001.

[28] R. V. Ommering. Building Product Populations with Software Components. *24th International Conference on Software Engineering*, Orlando, Florida, May 2002.

[29] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based Runtime Software Evolution. *20th International Conference on Software Engineering,* Kyoto, Japan, April 1998.

[30] D. Schmidt. TAO. http://www.cs.wustl.edu/~schmidt/TAO.html

[31] B. Selic. Real-Time Object-Oriented Modeling (ROOM). *2nd IEEE Real-Time Technology and Applications Symposium,* June, 1996

[32] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Trans. on Software Engineering*, April 1995.

[33] M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. *Prentice Hall*, 1996.

[34] Sun Microsystems. JINI(TM) Network technology. http://wwws.sun.com/software/jini/

[35] W. Tracz. Domain-Specific Software Architecture Pedagogical Example. *ACM Software Engineering Notes*, July 1995.

[36] University of Southern California's "Software Engineering for Embedded Systems" class website: *http://sunset.usc.edu/classes/cs599_2002/index.html*

[37] S. Vestal. MetaH Programmer's Manual, Version 1.09. *Technical Report, Honeywell Technology Center,* April 1996.

[38] B. Werger. A Situated Approach to Scalable Control for Strongly Cooperative Robot Teams. *Ph.D. Thesis.* University of Southern California, May 2001.

[39] X2000/Mission Data System (MDS) project. *http://x2000.jpl.nasa.gov/nonflash/technology/mds.html*

[40] S. S. Yau and F. Karim, Context-Sensitive Middleware for Real-time Software in Ubiquitous Computing Environments. *Proceedings of the International Symposium on Object-oriented Real-Time Distributed Computing 2001*, Magdeburg, Germany.

On Design Framework of Context Aware Embedded Systems

Abhay Daftari Nehal Mehta Shubhanan Bakre Xian-He Sun

Department of Computer Science
Illinois Institute of Technology
{daftabh, mehtneh, shubh, sun}@iit.edu

Abstract

The primary goal of embedded systems is "Human-centered computing," that is providing service anywhere, anytime, and automatically. While electrical devices become smaller and smaller and more powerful, context awareness becomes more and more important for embedded systems. Although some of the later systems have been developed with context awareness in mind, how to design a context aware embedded system systematically is still an issue that eludes researchers in the field. This study introduces the importance of context awareness in today's embedded systems, divides the design of context aware systems into context aware applications and infrastructure. It further applies aspect orientation in the design of context aware infrastructure to model the architectural/system from the developer's point of view. We show that applying aspect orientation in the development of context aware embedded systems is feasible and has real potential.

Keywords

Context awareness, embedded systems, aspect oriented software development, context aware infrastructure, human-centered computing.

1. Introduction

The Early 1990s saw many efforts towards reducing the size of the computer and its portability. Small but powerful devices become widely available that can be embedded anywhere and can be carried by the user wherever he goes. During the same period, growth in the wireless technology gave rise the popularity of coordinating small powerful devices to form an embedded computing environment for the "Human-centered computing." The size and the power of the computing devices are no longer the determining factors in the design of embedded systems, but the quality of service of the "Human-centered computing" is a key factor. Context awareness is becoming an important factor in the embedded system design.

The vision of mobile computing [1] is to provide some form of freedom to the end user by providing computing support when the user is in mobile mode. Mobility plus the availability of embedded systems, or so-called "smart spaces" constitute pervasive computing, a broader view of providing 'Human Centered computing'. Pervasive computing talks about providing computing everywhere and at all times [1]. This is done through providing smartness into embedded devices and systems, and providing balance between assistance and interference to the end user [2].

It is clear that understanding the user's surroundings will play a significant role in realizing the goals of Pervasive computing as well as embedded systems. Context awareness is directed effort towards understanding the environment around the user with the help of smart devices embedded within its environment and improve the end-user's experience by using this knowledge. Hence, context awareness is the key research area in embedded systems. It has been explained what constitutes a context aware application, and what context is and what are the definitions and categories of context and context-aware (CA)[3].

To understand what context awareness is, first consider the traditional classroom scenario:

> Professor T informs students about the updated course website that contains lecture slides for the day's lecture and that they need to bring the slides in the class for better understanding as professor T is going to use projector for presentation. Even after receiving this notification, some of the students either did not read the notification or some of them forgot about it before the class. Hence, class is conducted with some students have the slides in front of them while some of them do not.

Now, consider another scenario of smart classroom environment, where:

> If professor T is moving towards the projector and lights in the room are off, then the environment pervasively transfers the presentation slides from the professor's handheld device to students' handheld device and the projector starts the presentation.

Discussion:

> The second scenario takes into account the environment context information in which it is executed. In the above example, the context information is the location of the professor (classroom), the state of the lights in the room (off), the activity of the professor (moving towards the projector), and the number of students in the classroom. Although the same set of information was available in the traditional classroom scenario, it was not utilized towards better end user experience. This type of utilization is in line with the vision of context aware computing. In the later sections, we will base our discussions upon the above smart classroom scenario.
>
> There have been efforts in achieving context awareness through various approaches. The major goal of these approaches is to capture surrounding context and adapt it as per end-user needs. But still, context awareness is in inception stage and current approaches provide limited context information. As anticipated [1], future context aware approaches expected to support variety of context information and in large number. Hence, these systems should be able to meet the requirements of scalability and ability to evolve to fulfill the future needs.

Aspect Orientation (AO) is a relatively new methodology for Software Development (SD) [4] that promises better design leading to properties including scalability and ability to evolve. This study analyzes context-awareness from the perspective of applying AO in context aware system/infrastructure design.

2. Aspect Oriented Software Development

A requirement is analyzed and some design is made which then is implemented. But during the process, implementation is based on design documents that are in turn based on requirement documents. Most of the times, it is difficult to trace requirements into the design and then in the implementations. Modification in any one of these – requirements, design or implementation - will require propagating it into all of them, which is complicated and causes problems. The main reason behind this is that the system is in different forms at different levels.

The need for dealing with issues 'one at a time,' was coined as the principle of 'separation of concerns' [5]. Aspect Orientation Software Development (AOSD) is relatively a new software development methodology, based on principle of separation of concerns, for achieving a number of desired properties in a system such as extensibility, modularity, etc.

A 'concern' is a functional or non-functional property of a system like security, synchronization, logging, etc. During expression of design into implementation, due to the nature of some concerns and/or limitations in the expression techniques used, it is not implemented in a modular way and it tends to crosscut the rest of the implementation causing code tangling. Such a concern is called crosscutting concern. An aspect, on the other hand, is modular realization of a crosscutting concern. After incorporating aspects within the system, they need to interact with each other to form an operational system. The process of achieving this interaction is termed as weaving. An aspect consists of two parts, the first part – called advice – that contains the functional details of concern, and the other part – called pointcut – which identifies the points of interaction with the rest of the system. A joinpoint is a principled point in the execution of a program; for example, a method call is one form of joinpoint. A pointcut, defined above, can be imagined as a collection of such joinpoints.

3. CA Infrastructure requirements

Brief list of projects towards context awareness is Aura (CMU)[8], Oxygen (MIT)[1], RCSM (Re-configurable Context Sensitive Middleware, ASU)[9], GAIA (UIUC)[10], Context Toolkit (Georgia Tech)[11], SOLAR (Dartmouth)[12], Rome (Trigger Based Context awareness Infrastructure, Stanford)[13], Rapidware (Michigan State University)[14], Multi-Sensor context Aware Clothing (Lancaster, UK)[15], Charade (Gesture Recognition, University of Paris, France)[16], One.World (University of Washington, Seattle)[17].

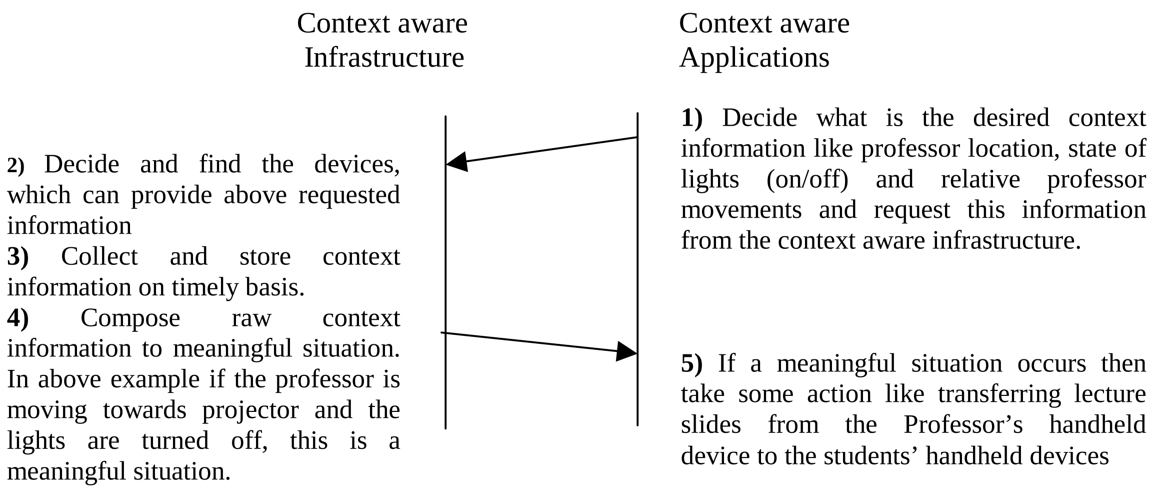
Based on the analysis of above system, we have observed that preliminary approach was to just provide context awareness for one or two context that too embedded into platform. This was platform specific and not sufficient for any complex context awareness.

The improvement over the past approach identified the need for many types of context and the design of standalone applications. Going back to our old example of Smart Classroom, in order to provide a smart environment a number of activities are needed including:

1. Decide what the desired context information is, like professor location, state of light (on/off) and relative professor movements.
2. Decide and find the devices, which can provide above context information
3. Collect and store the above context information on a timely basis.
4. Compose raw context information to meaningful situations. In the above example if the professor is moving towards the projector and the lights are turned off, this is a meaningful situation.
5. If a meaningful situation occurs then action should be taken like transferring the lecture slides from the Professor’s handheld device to the students’ handheld devices.

It can be observed that the context aware applications themselves would have to deal with activities like communicating with context sources, collecting context data, storing and managing context data for future, and finally using and adapting context data as per user’s needs. The approach of storing context data on individual basis resulted in data redundancy. Moreover, to design a similar system, all the steps need to be redone from the beginning. So the issues in terms of extensibility and ability to evolve to future needs of context aware system needed to be addressed. Learning from this approach, the current research in context awareness took an architectural approach to towards system design, in which the total responsibilities are divided between CA infrastructure and CA applications. The context aware infrastructures are responsible for collecting, managing and delivering context on behalf of CA applications, whereas the CA applications are responsible for adaptation.

In the above example, such a division would be:



1) Decide what is the desired context information like professor location, state of lights (on/off) and relative professor movements and request this information from the context aware infrastructure.

2) Decide and find the devices, which can provide above requested information

3) Collect and store context information on timely basis.

4) Compose raw context information to meaningful situation. In above example if the professor is moving towards projector and the lights are turned off, this is a meaningful situation.

5) If a meaningful situation occurs then take some action like transferring lecture slides from the Professor's handheld device to the students' handheld devices

Responsibilities in steps 2, 3 and 4 lie with the infrastructure and those in steps 1 and 5 should lie with the application. Applications only need to describe the required context information and request context aware infrastructure to collect this information on its behalf. The responsibilities of finding the sources for the required context, and dealing with context source specific details are given to context aware architecture. This approach will help the application designer to focus on designing adaptation behavior of context aware application and provide freedom from dealing with unnecessary source specific details. On the other hand, similar approach can be used for many other context aware applications because the infrastructure for collection and delivery is already in place, making architecture scalable to many context aware applications. So, a context aware computing environment should consist of two parts, the system infrastructure and the application software. The CA system is responsible for collecting, storing - managing and delivering it to context aware applications, and the CA applications are responsible for adapting it.

Based on the separation of infrastructure and application principle and based on the analysis of existing works, we have identified the following goals that any context aware system should fulfill:

Goals:

- Collect context on behalf of application
- Provide ways to subscribe and deliver collected context to respective applications
- Store and manage past context for future needs
- Mask Heterogeneity and provide independence in terms of programming languages used, types of system support.
- Systems should be modular, and allow only required components to load.
- Systems should be extensible to future context need
- Systems should be scalable to many context types and many context aware applications
- Components can be reusable for future context need

Context aware is still a relatively new concept. Many of the proposed context-aware systems are the result of convergence of independent projects targeted at realizing different sets of scenarios in pervasive computing. For example, CODA and Odyssey file system – although in the beginning they were designed for mobile data access, and eventually converged to provide basic form of context awareness into Project Aura.

The abstract goals can be transformed into two types of requirements for system design: (1) functional requirements that support the core functionality of system, and (2) non-functional requirements for orthogonal system needs like extensibility and scalability etc. Functional requirements are those that are specific to a CA system whereas, non-functional are orthogonal requirements that are generally true for any system. Non-functional requirements are also known as quality requirements.

3.1 Functional Requirements

Following are the identified functional requirements for a design of a context-aware system:

1. Context Collection: This deals with collection of data from the sensors. This involves dealing with questions about the data model to be used for collection (Push / Pull / Request / Reponses), Insertion / Removal of Context Providers.

2. Context Storage/Management: This requirement pertains to the storage of the collected context data and its management. The storage of context data is significant for a number of purposes such as reproduction of context, logging, mining and future predictions. Also, context data needs to be communicated at various places such as from sensors to storage, from storage to consumers and most probably proxy consumers. Often the data is time sensitive. Hence when, where and how to move the data, needs to be managed.

3. Context Subscription/Delivery: There needs to be ways for consumers to acquire the collected and stored context data. The System thus needs context Subscription methods, and a model for distribution (Push / Pull / Request / Reponses).

4. Context Analysis and Composition Ability: This is the most significant functional requirement for context aware systems. Looking at various definitions, one can derive that a context could be a composition of multiple raw data provided by sensors. And not all the permutations of raw data will result in sensible contexts. Hence, the task of context-composition is complex and is believed to involve decision-making on the basis of history and experience.

3.2 Non-Functional Requirements

1. Quality of Service: Due to the dynamic nature of resources like sensors, devices, and network bandwidth, it becomes essential for context-aware systems to provide the ability to the applications for specifying such resource constraints.

2. Security: As with any other system, security is one of the prime non-functional requirements. This requirement arises from the basic question - To what extent should the private information be exposed to context-aware services transparently to the user? Also, how should the issues like access control, authentication, authorization, and data encryption be addressed in such an environment?

3. Heterogeneity: Heterogeneity arises due to different hardware platforms and also due to the varying capabilities of the devices used. For achieving portability of applications across multiple platforms, it is necessary to fulfill this requirement.

4. Scalability: The Ad-hoc discovery of resources, the changes in the number of users and resources, and the limited computing capabilities of devices, introduces the problem of scalability.

5. Adaptability: In pervasive computing when a user switches from a resource-rich device to a resource-strapped device, the applications either should be able to adapt to these changes to provide seamless service, or alternatively scale down according to the new surroundings.

6. Fault Tolerance: The Ability of the system to adapt to compensate for errors can be termed as fault tolerance. Doing things right even if pre-conditions deviate to a limited extent is especially desired in CA systems.

7. Extensibility: Evolution is a part of the software life cycle. In order to provide the support for new features, extensibility at different levels is necessary. For example, in a context-aware system context should be extensible to new context and composition.

8. Functional Modularity: There should be clear distribution of responsibilities between the devices, the applications, the infrastructure and the components within this infrastructure. All the components of the system should be developed in an independent manner. This requirement arises out of the principle of separation of concerns that inherently brings the number of advantages to the system such as comprehensibility, reusability, re-configurability, and other such properties.

9. Mobility: Pervasive computing is synonymous with user-centric computing. Unlike traditional computing that forces the user to follow computing and the data, pervasive computing focuses on providing the computing and data whenever and wherever the user needs it. This introduces mobility constraints on context-aware applications.

Discussion:
Functional requirements present the core functionality of the architecture and are necessary to fulfill the need of the context aware application support. For a better understanding, consider the 'smart classroom environment' example scenario. Once the application has decided the types of context information such as the information about the location, the state of the lights, and the movement of the professor, it describes these context needs to the context aware architecture.

1. Context Collection:
   The infrastructure decides what context providers to use to get the context information. Once they are known, it will collect the information from them.

2. Context Storage/Management: The collected context data for the location and the status of the light are stored for future use. Although this data is not currently used by 'Smart classroom environment', it could be utilized in the future. It is not necessary that all applications need it, but as context awareness is an evolving area, this data will be useful to predict the system's behavior on the basis of past history.

3. Context Subscription/Delivery: In the beginning, the application will describe the context needs. When some specific situation occurs, like 'presentation is going to start,' the application will be notified for the same.

4. Context Analysis and Composition Ability: This function accumulates many raw contexts information like 'light is off and professor is moving towards projector,' then decides that the required situation 'presentation is going to start' is generated.

Here, we showed that the above functional requirements are necessary and sufficient for any context aware architecture to support context aware application.

4. Applying AO to CA Infrastructure

The above section has mapped the goals into functional and non-functional requirements. Based on the principle of separation of concerns, AOSD tries to modularize the requirements into the aspects. Once the aspects are identified and their weaving accomplished, one could easily trace the requirements to the design and from the design to the implementations. Many approaches within AOSD understand the system in terms of the core functional components and the aspectual components that constitute the non-functional components [19], whereas, others understand it in terms of the aspectual components composed of both the functional and the non-functional components [20]. But the major issue of identifying and distributing concerns into core and/or aspects still remains an open issue.

Before discussing application of AO in Context Awareness, we mention few research projects that have understood the potential of AO and acknowledged it by applying it to their system. Some of the areas are operating system [24], middleware [25], etc. It can be observed that all these areas have crosscutting concerns and code-tangling phenomenon. It has been demonstrated AO has helped in separating these concerns and achieved better non-functional goals. As can be observed from the discussion above regarding requirements of context aware system, there is good extent of crosscutting concerns in CA system that makes it prospective candidate for application of AO.

We will walk you through an example by keeping context awareness in focus and show how the application of AO will result in better design. To analyze the application of AO in context awareness, let us focus on the design and implementation of one of the functional requirements – 'Context Collection'.

> To design a context collection module to support number of context providers. Currently the system needs to support only a few number of context providers, but this number could increase in future. The location of context providers can possibly be local and/or remote thereby involving communication over the network.

Considering the functionality needed for context collection, the pseudo code for the context collection module can be represented as follows:

Looking at the requirements the concerns in the system are not clearly visible. At first instance it looks that entire functionality is related to context collection. Further examination clarifies that the functionality of context collection is tangled with the policies related to scalability. Following are the part of functionality that refer to scalability,

1. What should be the frequency at which the context data is collected from the providers? – Context Collection Frequency.

```
/* Start up work */
Set some default value for context collection frequency.
Initialize the context location store
Find the network latency and how fast then can switch between connections

/* Loop for new context providers */
Loop
        Look for new context provider
        If found any,
                Get location information
                Store in the location store for future need
        Get the context collection frequency information
        …….
Collect context from all registered context providers periodically.
/* end of loop*/

Context collector is shutting down.
```

2. What are the capabilities of individual providers in terms of operating frequency? – Context Provider Details.
3. What are the communication constraints such as bandwidth availability, network protocols, etc.? – Other Dependencies.

Ideally, the modifications or changes in the above policies should be independent to the context collection functionality. This becomes crucial when there is change in the number of context providers of the system or in the network bandwidth, for example. As these kind of changes will require modifications in the policies that are scattered in the number of functional components of the system. But, in the above design of context collection module establishing such independence is difficult to achieve and sometimes impossible due to the tight coupling between the functional and non-functional requirements. Following section explains how AO will make above design better and makes to achieve better scalability.

When aspect oriented technique is used, emphasis is given to the separation of concerns such as the basic context-collection concern and scalability concern that relates to a number of variables mentioned above. Following figure shows what would be the implementation of solution of above issues using an AO approach.

```
class ContextCollector {
        public void collect() {//Collect the context Information}
        public void send(){//send the collected information}}

aspect Scalability {
        pointcut repeat() : call(ContextCollector.collect());
        pointcut send() : call(ContextCollector.send(//context information));
        before() : repeat () {
                //Decide the frequency for context collection
                        //and introduce the loop   }
        after () : repeat () {
                //end the loop  }
        before () : send () {
                //Get the network parameter and adapt accordingly }}
```

In the above pseudo-code for context collection, using AO, the developer is oblivious to any other concerns while implementing of context collection functionality. The two concerns for context collection and scalability are clearly understood and realized as two separate modules.

Discussion:
In the AO technique, concerns are separated and modularized in one place with the specification of the interaction between these separated modules. These modules (aspects) are then weaved together to form a complete system. The functional code for context collection is completely free from the code for scalability due to separation achieved. Moreover, if the scalability policies change in the future, as it will be modularized at one place, its replacement will be multiple times easier due to reuse of weaving code as a result of the properties of quantification and obliviousness inherent in the system. Meaning, the need to visit the places of crosscut and modify the code will be eliminated.

With conventional implementation techniques, such as procedural or object oriented programming, it is clear from [19][21][22][23] that if a requirement like 'need to add new context provider,' is not adequately addressed in the design phase then it is difficult to incorporate it later on in the life cycle of the software system. On the other hand, with AO, requirements would be easily traceable into design in terms of aspects and also in the implementation, as aspects are weaved by a weaver to generate the implementation of the design. Hence incorporating modifications in existing requirements and also adding new requirements using AO will result better design due to added flexibility, extensibility and ability for evolution. Thus, following needs to be done in order to add a new context provider that needs to use existing authentication policies, while using AO in the system: With AO, the existing authentication policies is modularized at one place and it is 'inserted' into appropriate places within the functional code through a standardized weaving mechanism similar to the logging example presented in the introduction of AOSD.

The above discussion is presented as a thought-provoking concept. Similar discussion holds true for applying AO to fulfillment of other non-functional requirements of a CA system.

In order to see if AO can help towards improvement of existing CA systems, one needs to analyze their design and implementation. We have observed that current literature in context aware work does not mention AO at all or just started [9] [14] to use it as a tool for their system design. Due to limitation in available literature, it is difficult to support this observation. But, there is support available from AO in terms of language extensions, frameworks, and pre-processors that would be helpful to examine and improve existing CA systems. Some of the examples of AO tools are: AspectJ [19] - an extension to Java, HyperJ [20] – a different approach to AO in Java, AspectC++ [26] - an extension to C++, etc.

5. Conclusion

We present our position in embedded system development: context awareness is becoming an increasingly important factor; the need of separating the context aware infrastructure development and context aware application development; and applying aspect orientation in context aware system design. A context aware scenario is used to illustrate the proposed concepts and arguments at various levels during analysis of the CA system. The goals and requirements of context aware system are identified and formally introduced. Through an example, we illustrate how aspect orientation can be applied in the development of a context aware system. The ability to evolve at a high rate has become crucial for the context aware systems because of the ever changing and the diverse needs in context awareness. This study highlights the importance of context awareness in embedded system design and builds a framework for future context aware system development. Currently, we are using the framework in developing a context aware system, Scarlet, at IIT.

6. References:

[1] "Project Oxygen Homepage," http://oxygen.lcs.mit.edu/index.html (current Aug. 2003).
[2] M. Satyananrayanan, "Pervasive Computing: Vision and Challenges," IEEE Personal Communications, Aug. 2001.
[3] Survey – context aware refer term report
[4] AOSD http://www.aosd.net/
[5] E.W. Dijkstra, "A Discipline of Programming," Prentice Hall, Englewood Cliffs, NJ, 1976.
[6] R.E. Filman and D.P. Friedman, "Aspect-Oriented Programming is Quantification and Obliviousness," Workshop Advanced Separation of Concerns, OOPSLA 2000, October 2000, Minneapolis.
[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," ECOOP, vol. 2072, pp. 327--353, 2001
[8] D.Garlan, D. Siewiorek , A. Smailagic and P. Steenkiste, "Project Aura: Toward Distraction-Free Pervasive Computing," IEEE Pervasive Computing, vol. 1, no. 1, Apr.-Jun. 2002, pp. 22-31.

[9] S.S. Yau, F. Karim, Y. Wang, B. Wang, and S.K.S. Gupta, “Reconfigurable Context-Sensitive Middleware for Pervasive Computing,” IEEE Pervasive Computing, Jul.-Sept. 2002, vol. 1, no. 3, pp 33-40.
[10] M. Román, C.K. Hess, R. Cerqueira, A. Ranganathan, R.H. Campbell and K. Nahrstedt, “Gaia: A Middleware Infrastructure to Enable Active Spaces,” IEEE Pervasive Computing, Oct.-Dec. 2002, vol. 1, no. 4,pp. 74-83.
[11] A.K. Dey and G.D. Abowd, “The Context Toolkit: Aiding the Development of Context-Aware Applications,” Workshop Software Engineering for Wearable and Pervasive Computing, Limerick, Ireland, Jun. 2000.
[12] G. Chen and D. Kotz, “Solar: An Open Platform for Context-Aware Mobile Applications,” Proc. 1st Int’l Conf. Pervasive Comp., Zurich, Switzerland, Jun. 2002, pp. 41-47.
[13] A.C. Huang, B.C. Ling, S. Ponnekanti and A. Fox, "Pervasive Computing: What Is It Good For?" Workshop Mobile Data Management (MobiDE) in conjunction with ACM MobiCom '99, Seattle, WA, September 1999.
[14] RAPIDware, http://www.cse.msu.edu/rapidware/
[15] K.V. Laerhoven, A. Schmidt and H.W. Gellersen, “Multi-Sensor Context-Aware Clothing,” Proc. 6th Int’l Symposium Wearable Computers, ISWC 2002.
[16] T. Baudel and M. Beaudouin-Lafon, "CHARADE: remote control of objects using free-hand gestures," Comm. the ACM, Jul. 1993, vol. 36, no. 7, p. 28-35.
[17] L. Arnstein, R. Grimm, C. Hung, J.H. Kang, A. LaMarca, G. Look, S.B. Sigurdsson, J. Su and G. Borriello, “Systems support for ubiquitous computing: A case study of two implementations of Labscape,” Proc. 2002 Int’l Conf. Pervasive Computing, Zurich, Switzerland, Aug. 2002.
[18] A K. Dey, “Providing Architectural Support for Building Context-Aware Applications,” doctoral dissertation, College of Computing, Georgia Institute of Technology, Atlanta, Nov. 2000.
[19] AspectJ, http://eclipse.org/aspectj/
[20] HyperJ, http://www.research.ibm.com/hyperspace/index.htm
[21] C. Constantinides, A. Bader, and T. Elrad, “A framework to address a two-dimensional composition of concerns,” Proc. 1st Workshop Multi-Dimensional Separation of Concerns in Object-Oriented Systems at OOSPLA’99, 1999.
[22] G. Kiczales, J. Lampoing, A. Mendhekar, C. Maeda, C. Lopez, J. Loingtier, and J. Irwin, “Aspect-oriented programming,” Proc. European Conference on Object-Oriented Programming (ECOOP), LNCS 1261, 1997.
[23] B. Tekinerdogan and M. Aksit, “Deriving Design Aspects from Conceptual Models,” Position paper ECOOP ’97 Workshop Aspect-Oriented Programming, pp. 410-413, 1998.
[24] P. Netinant, C.A. Constantinides, A. Bader, and T. Elrad, “Supporting the Design of Adaptable Operating Systems Using Aspect-Oriented Frameworks,” Int’l Conf. Parallel and Distributed Techniques and Applications, PDPTA 2000, Oct. 2000
[25] AspectIX - Aspect Oriented Middleware, http://www.aspectix.org/
[26] A. Gal, W. Schroder-Preikschat and O. Spinczyk, “AspectC++: Language Proposal and Prototype Implementation,” OOPSLA 2001 Workshop Advanced Separation of Concerns in Object Oriented Systems, Tampa, Florida, Oct. 2001

Techniques for Improving Test-Driven Design

Martin Wirsing, Hubert Baumeister, and Alexander Knapp
LMU München, Institut für Informatik, Oettingenstr. 67, D-80638 München
Email: {wirsing, baumeist, knapp}@informatik.uni-muenchen.de

Abstract

Early test development and specification enhance the quality and robustness of software as experience with new agile software development methods shows. The methods propagate test-first techniques and early prototyping through executable design models. For UML, Model-Driven Architecture is oriented towards executable models. Several authors propose scenarios specified by sequence diagrams as test cases for state diagrams; more generally, using software model checking one may automatically verify whether state diagrams or code satisfy properties defined by sequence diagrams. Other approaches use OCL invariants and pre-/post-conditions for instrumenting Java code with assertions.

Also, Extreme Programming (XP) requires to write tests before writing the code as a means for making the software robust and more easily refactored. Popular tools for XP are the family of xUnit tools—with JUnit as the most well known instance—for writing automated unit tests and Fit for writing automated acceptance tests.

In this paper we propose techniques for extending and improving such test-driven development methods, where executable tests drive the development process. Scenarios and properties serve us as a combined basis for system specification and test cases. Scenarios are defined by sequence diagrams written in a powerful sublanguage of UML 2.0 which allows us to specify not only possible scenarios but also forbidden scenarios (failure traces). A forbidden scenario is a scenario where one wants to say that after legally performing some steps, the next step should now occur. This is not expressible UML 1.5 sequence diagrams. Further extensions w.r.t. other approaches are the use of nested method invocations and state invariants.

Scenarios are examples of successful or un-successful system runs. By extracting common properties of several scenarios we obtain invariants and pre-/post-conditions written in OCL or JML. The behaviour of the system is described either by models such as state diagrams or activity diagrams, or by code e.g. written in Java.

For testing we insert invariants and pre- and post-conditions as assertions in the code and the behaviour models. Then we test the instrumented system behaviour with respect to the possible and forbidden scenarios. This is done by translating possible and forbidden scenarios to Fit tests for scenarios involving user interaction and to JUnit tests for system scenarios.

Due to the addition of the assertions to the system behaviour we obtain a more complete test coverage and further possibilities for checking dynamically the internal consistency of the system specification.

For verification, we propose two approaches: interactive theorem proving combined with symbolic evaluation and model checking. To be successful with the latter technique we have to restrict the models to finite domains. Therefore we construct suitable abstractions of the scenarios and the system behaviour and verify the abstractions using a model checker. For verifying the general case, symbolic evaluation helps to reduce considerably the number of necessary interactions with an interactive theorem prover.

Currently we are integrating these techniques into a user-oriented collaborative development environment.