UNIVERSITY OF CALIFORNIA, SAN DIEGO

Composing Crosscutting Concerns: A Service-Oriented View

A dissertation submitted in partial satisfaction of the requirements for
the degree Doctor of Philosophy

in

Computer Science

by

Massimiliano Menarini

Committee in charge:

Professor Ingolf Krüger, Chair
Professor Bernd Finkbeiner
Professor Ranjit Jhala
Professor Sorin Lerner
Professor Ramesh Rao
Professor Maurizio Seracini

2012

The Dissertation of Massimiliano Menarini is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

____

____

____

____

____

____
Chair

University of California, San Diego

2012

DEDICATION

To my brother and sister who followed me across the ocean, and to my parents that are home.

EPIGRAPH

Per correr miglior acque alza le vele
omai la navicella del mio ingegno,
che lascia dietro a sé mar sì crudele;

Dante Alighieri, Divina Commedia
Purgatorio I, 1-3

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

AATC	Advanced Automatic Train Control
ABS	Anti-lock Braking System
ACL	Access Control List
ADL	Architecture Description Language
AOM	Aspect-Oriented Modeling
AOP	Aspect-Oriented Programming
API	Application Programming Interface
BART	Bay Area Rapid Transit System
BPEL4WS	Business Process Execution Language for Web Services
CAN Bus	Controller Area Network Bus
CLS	Central Locking System
CORBA	Common Object Request Broker Architecture
COTS	Commercial-Off-The-Shelf
CTL	Computation Tree Logic
ECU	Electronic Control Unit
ESB	Enterprise Service Bus

FMEA	Failure Modes and Effects Analysis
FSM	Finite State Machine
FTA	Fault Tree Analysis
FT-CORBA	Fault Tolerant CORBA
FT-SOAP	Fault Tolerant SOAP
HMSC	High Level MSC
HTTP	Hypertext Transfer Protocol
IPS	Interaction Pattern Specifications
LET	Logical Execution Time
LSC	Live Sequence Chart
LTL	Propositional Linear Temporal Logic
MARTE	Modeling and Analysis of Real-Time and Embedded Systems
MBE	Model-Based Engineering
MDA	Model-Driven Architecture
MDE	Model-Driven Engineering
MOTT	Message Origination Time Tag
MSC	Message Sequence Chart

OCL	Object Constraint Language
OEM	Original Equipment Manufacturer
OMG	Object Management Group
ORB	Object Request Broker
PIM	Platform Independent Model
PSM	Platform Specific Model
QoS	Quality of Service
RAS	Rich Application Service
RIS	Rich Infrastructure Service
RSML	Requirements State Machine Language
RT-CORBA	Real Time CORBA
RTRM	Run-Time Resource Management
S3EL	Service-Oriented Software & Systems Engineering Laboratory
SADL	Service Architecture Description Language
SADT	Structured Analysis and Design Technique
SASS	Structured Analysis and System Specification
SAT	Satisfiability

SCR	Software Cost Reduction
SDC	Service/Data Connector
SLA	Service Level Agreement
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SRD	Structured Requirements Definition
SSA	Structured System Analysis
TDL	Timing Definition Language
UCSD	University of California San Diego
UDDI	Universal Description, Discovery, and Integration
UML	Unified Modeling Language
UMO	Universal Message Object
VSL	Value Specification Language
WS-BPEL	Web Services Business Process Execution Language
WS-CDL	Web Services Choreography Description Language
WSDL	Web Services Description Language
XML	Extensible Markup Language

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to acknowledge my advisor Professor Ingolf Krüger for his support and advice. This work would not have been possible without his help. I want to thanks all my coauthors and all members of my research team for their valuable suggestions. In particular I thank Barry Demchak and Filippo Seracini for their support in getting this thesis written. Finally, I thank all the members of my thesis committee for reading this manuscript and providing feedback.

Chapter 1, in part, is a reprint of material as appeared in C. Farcas, E. Farcas, I. H. Krueger, and M. Menarini, "Addressing the Integration Challenge for Avionics and Automotive Systems - From Components to Rich Services," in The Proceedings of the IEEE Special Issue on Aerospace and Automotive Software, K. V. Prasad (Ed.), vol. 98, no. 4. IEEE, Apr. 2010, pp. 562-583. The dissertation author was the primary investigator and author of the text used in this chapter.

Chapter 2, in part, is a reprint of material as appeared in V. Ermagan, I. H. Krüger, and M. Menarini, "A Fault Tolerance Approach for Enterprise Applications," in Proceedings of the IEEE International Conference on Services Computing (SCC). Jul. 2008. The dissertation author was the primary investigator and author of the text used in this chapter.

Chapter 3, in part, is a reprint of material as appeared in C. Farcas, E. Farcas, I. H. Krueger, and M. Menarini, "Addressing the Integration Challenge

for Avionics and Automotive Systems - From Components to Rich Services," in The Proceedings of the IEEE Special Issue on Aerospace and Automotive Software, K. V. Prasad (Ed.), vol. 98, no. 4. IEEE, Apr. 2010, pp. 562-583. The dissertation author was the primary investigator and author of the text used in this chapter.

Chapter 4, in part, is a reprint of material as appeared V. Ermagan, I. H. Krüger, and M. Menarini, "Aspect Oriented Modeling Approach to Define Routing in Enterprise Service Bus Architectures," in MiSE '08: Proceedings of the 2008 international workshop on Models in software engineering, Leipzig, Germany. New York, NY, USA: ACM, May 2008, pp. 15-20. The dissertation author was the primary investigator and author of the text used in this chapter.

Chapter 6, in part, is a reprint of material as appeared in E. Farcas, I. Krueger, and M. Menarini, "Consistency Management of UML Model," Real-time Simulation Technologies: Principles, Methodologies, and Applications, K. Popovici and P. J. Mosterman (Eds.), ch. 12, p. 38, CRC Press, 2012. The dissertation author was the primary investigator and author of the text used in this chapter.

Chapter 7, in part, is a reprint of material as appeared in 5 papers: 1) in C. Farcas, E. Farcas, I. H. Krueger, and M. Menarini, "Addressing the Integration Challenge for Avionics and Automotive Systems - From Components to Rich Services," in The Proceedings of the IEEE Special Issue on

Aerospace and Automotive Software, K. V. Prasad (Ed.), vol. 98, no. 4. IEEE, Apr. 2010, pp. 562-583.

2) V. Ermagan, I. H. Krüger, and M. Menarini, “A Fault Tolerance Approach for Enterprise Applications,” in Proceedings of the IEEE International Conference on Services Computing (SCC). Jul. 2008.

3) V. Ermagan, I. H. Krüger, and M. Menarini, “Aspect Oriented Modeling Approach to Define Routing in Enterprise Service Bus Architectures,” in MiSE '08: Proceedings of the 2008 international workshop on Models in software engineering, Leipzig, Germany. New York, NY, USA: ACM, May 2008, pp. 15-20.

4) E. Farcas, I. Krueger, and M. Menarini, “Consistency Management of UML Model,” Real-time Simulation Technologies: Principles, Methodologies, and Applications, K. Popovici and P. J. Mosterman (Eds.), ch. 12, p. 38, CRC Press, 2012.

5) E. Farcas, I. Krueger, and M. Menarini, “Modeling with UML and Its Real-Time Profiles,” Real-time Simulation Technologies: Principles, Methodologies, and Applications, K. Popovici and P. J. Mosterman (Eds.), ch. 5, p. 36, CRC Press, 2012

The dissertation author was the primary investigator and author of the text used in this chapter.

Proceedings of the IEEE Special Issue on Aerospace and Automotive Software, and 04/2010

© 2008 IEEE. Reprinted, with permission, from V. Ermagan, I. H. Krüger, and M. Menarini, A Fault Tolerance Approach for Enterprise Applications, Proceedings of the IEEE International Conference on Services Computing (SCC), and 07/2008

This work is based on an earlier work: Aspect Oriented Modeling Approach to Define Routing in Enterprise Service Bus Architectures, in Proceedings of the 2008 international workshop on Models in software engineering (MiSE '08), © ACM, 2008. http://doi.acm.org/10.1145/1370731.1370735

Copyright 2012 From Real-time Simulation Technologies: Principles, Methodologies, and Applications by K. Popovici and P. J. Mosterman. Reproduced by permission of Taylor and Francis Group, LLC, a division of Informa plc.

VITA

2003	Laurea, University of Bologna, Italy
2004-2005	Software Architect for California Institute for Telecommunications and Information Technology
2008	Master of Science, University of California, San Diego
2012	Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

I. Krüger, M. Menarini, F. Seracini, M. Fuchs, and J. Kohl, "Improving the Development Process for Automotive Diagnostics," Proceedings of the 2012 International Conference on Software and Systems Process (ICSSP), Zurich, Switzerland, pp. 63-67, IEEE, 2012.

I. Krüger, B. Demchak, and M. Menarini, "Dynamic Service Composition and Deployment with OpenRichServices," Software Service and Application Engineering, vol. 7365, M. Heisel, Ed. pp. 120–146,Springer Berlin / Heidelberg, 2012.

E. Farcas, I. Krueger, and M. Menarini, "Modeling with UML and its Real-Time Profiles," Real-time Simulation Technologies: Principles, Methodologies, and Applications, K. Popovici and P. J. Mosterman (Eds.), ch. 5, p. 30, CRC Press, 2012.

E. Farcas, I. Krueger, and M. Menarini, "Consistency Management of UML Model," Real-time Simulation Technologies: Principles, Methodologies, and Applications, K. Popovici and P. J. Mosterman (Eds.), ch. 12, p. 38, CRC Press, 2012.

C. Farcas, E. Farcas, I. H. Krueger, and M. Menarini, "Addressing the Integration Challenge for Avionics and Automotive Systems - From Components to Rich Services," in The Proceedings of the IEEE Special Issue on Aerospace and Automotive Software, K. V. Prasad (Ed.), vol. 98, no. 4. IEEE, Apr. 2010, pp. 562-583.

I. Krueger, C. Farcas, E. Farcas, and M. Menarini, "Requirements Modeling for Embedded Realtime Systems," Model-Based Engineering of Embedded Real-Time Systems, H. Giese, B. Rumpe, and B. Schätz (Eds.), Lecture Notes in

Computer Science (LNCS), vol. 6100, ch. 7, pp. 155-199, Berlin, Heidelberg: Springer-Verlag, 2010.

J. Oldevik, M. Menarini, and I. Krüger, "Model Composition Contracts," in Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09),, A. Schürr and B. V. Selic (Eds.), vol. LNCS 5795, Denver, Colorado, USA, Berlin, Heidelberg: Springer Verlag, Oct. 2009, pp. 531-545.

I. H. Krüger, M. Meisinger, and M. Menarini, "Interaction-based Runtime Verification for Systems of Systems Integration," Journal of Logic and Computation, Nov. 2008.

B. Demchak, V. Ermagan, E. Farcas, T.-J. Huang, I. Krüger, and M. Menarini, "A Rich Services Approach to CoCoME," The Common Component Modeling Example, Comparing Software Component Models, A. Rausch, R. Reussner, R. Mirandola, and F. Plásil (Eds.), Lecture Notes in Computer Science, vol. 5153, pp. 85-115, Springer Berlin / Heidelberg, Aug. 2008.

V. Ermagan, I. H. Krüger, and M. Menarini, "A Fault Tolerance Approach for Enterprise Applications," in Proceedings of the IEEE International Conference on Services Computing (SCC). Jul. 2008.

B. Demchak, V. Ermagan, C. Farcas, E. Farcas, I. H. Krüger, and M. Menarini, "Rich Services: Addressing Challenges of Ultra-Large-Scale Software-Intensive Systems," in Proceedings of the ICSE 2nd International Workshop on Ultra-Large-Scale Software-Intensive Systems (ULSSIS 2008), Leipzig, Germany. New York, NY, USA: ACM, May 2008, pp. 29-32.

V. Ermagan, I. H. Krüger, and M. Menarini, "Aspect Oriented Modeling Approach to Define Routing in Enterprise Service Bus Architectures," in MiSE '08: Proceedings of the 2008 international workshop on Models in software engineering, Leipzig, Germany. New York, NY, USA: ACM, May 2008, pp. 15-20.

V. Ermagan, C. Farcas, E. Farcas, I. H. Krüger, and M. Menarini, "A Service-Oriented Approach to Failure Management," in Proceedings of the Dagstuhl Workshop on Model-Based Development of Embedded Systems (MBEES). Apr. 2008.

V. Ermagan, I. H. Krüger, and M. Menarini, "Model-Based Failure Management for Distributed Reactive Systems," Composition of Embedded Systems. Scientific and Industrial Issues. 13th Monterey Workshop 2006 Paris, France, October 16-18, 2006 Revised Selected Papers, F. Kordon and O. Sokolsky (Eds.), Lecture Notes in Computer Science, vol. 4888, pp. 53-74, Springer Berlin / Heidelberg, Dec. 2007.

M. Arrott, B. Demchak, V. Ermagan, C. Farcas, E. Farcas, I. H. Krüger, and M. Menarini, "Rich Services: The Integration Piece of the SOA Puzzle," in

Proceedings of the IEEE International Conference on Web Services (ICWS), Salt Lake City, Utah, USA. IEEE, Jul. 2007, pp. 176-183.

V. Ermagan, C. Farcas, E. Farcas, I. H. Krüger, and M. Menarini, “A Service-Oriented Blueprint for COTS Integration: the Hidden Part of the Iceberg,” in Proceedings of the ICSE Second International Workshop on Incorporating COTS Software into Software Systems: Tools and Techniques (IWICSS'07), Minneapolis, MN, USA. Washington, DC, USA: IEEE Computer Society, May 2007, p. 10.

I. H. Krüger, M. Meisinger, and M. Menarini, “Runtime Verification of Interactions: From MSCs to Aspects,” Runtime Verification, 7th International Workshop, RV 2007, Vancover, Canada, March 13, 2007, Revised Selected Papers, O. Sokolsky and S. Tasiran (Eds.), Lecture Notes in Computer Science, vol. 4839, pp. 63-74, Springer Berlin / Heidelberg, Mar. 2007.

V. Ermagan, T.-J. Huang, I. Krüger, M. Meisinger, M. Menarini, and P. Moorthy, “Towards Tool Support for Service-Oriented Development of Embedded Automotive Systems,” in Proceedings of the Dagstuhl Workshop on Model-Based Development of Embedded Systems (MBEES'07), Informatik-Bericht 2007-01. Fakultät für Informatik, Technische Universität Braunschweig, Jan. 2007.

V. Ermagan, I. Krueger, M. Menarini, J.-I. Mizutani, K. Oguchi, and D. Weir, “Towards Model-Based Failure-Management for Automotive Software,” in Proceedings of the ICSE Fourth International Workshop on Software Engineering for Automotive Systems (SEAS'07), Minneapolis, MN, USA. Washington, DC, USA: IEEE Computer Society, 2007, p. 8.

I. H. Krüger, M. Meisinger, and M. Menarini, “Applying Service-Oriented Development to. Complex System: a BART case study,” Reliable Systems on Unreliable Networked Platforms, 12th Monterey Workshop 2005, Laguna Beach, CA, USA, September 22-24, 2005. Revised Selected Papers, F. Kordon and J. Sztipanovits (Eds.), Lecture Notes in Computer Science, vol. 4322, pp. 26-46, Springer Berlin / Heidelberg, 2007.

I. H. Krüger and M. Menarini, “Queries and Constraints: A Comprehensive Semantic Model for UML2,” Models in Software Engineering. Workshops and Symposia at MoDELS 2006, Genoa, Italy, October 1-6, 2006, Reports and Revised Selected Papers, T. Kühne (Ed.), Lecture Notes in Computer Science, vol. 4364, pp. 327-328, Springer Berlin / Heidelberg, 2007.

I. H. Krüger, M. Meisinger, M. Menarini, and S. Pasco, “Rapid Systems of Systems Integration - Combining an Architecture-Centric Approach with Enterprise Service Bus Infrastructure,” in Proceedings of the 2006 IEEE International Conference on Information Reuse and Integration (IRI'06), Waikoloa, Hawaii, USA. IEEE Systems, Man, and Cybernetics Society, Sep. 2006, pp. 51-56.

F. Doucet, M. Menarini, I. H. Krüger, R. Gupta, and J.-P Talpin, “A Verification Approach for GALS Integration of Synchronous Components,” Electronic Notes in Theoretical Computer Science, vol. 146, no. 2, pp. 105-131, Jan. 2006.

FIELD OF STUDY

Major Field: Software Engineering

Sub-fields: Software Architecture

Service-Oriented Systems

Software Reliability Methods

Formal Methods

ABSTRACT OF THE DISSERTATION

Composing Crosscutting Concerns: A Service-Oriented View

by

Massimiliano Menarini

Doctor of Philosophy in Computer Science

University of California, San Diego, 2012

Professor Ingolf Krüger, Chair

Developing complex distributed software systems challenges the methodologies currently available for the crucial task of integrating crosscutting concerns. Crosscutting concerns are concerns that, albeit important, do not fit into the problem decomposition schema chosen for an application architecture. While service-oriented architectures (SOAs) have been proposed as a viable solution for this integration problem, the current state of the art is still insufficient.

This dissertation extends the current body of work on service-oriented systems to ease the integration of crosscutting concerns in large scale systems, though the applicability of this work spans from large enterprise systems-of-systems to the embedded systems domain. To achieve this result, the research presented here leverages two key elements: 1) an architectural blueprint called Rich Services, which supports the hierarchical decomposition of systems into a set of services communicating via a message based infrastructure, and 2) a model-based approach to capturing systems requirements and modeling systems according to the Rich Service blueprint.

This manuscript presents three contributions: 1) an aspect-oriented language for interaction models that support Rich Service compositions, 2) a technique that addresses failure management, which is an important example of a crosscutting concern, and 3) an approach to verifying the consistency of different views of the same system in different modeling languages.

The contributions presented in this dissertation are validated using case studies from the business applications and the automotive domains. The business application example is an important representative of problems relating to enterprise applications, and the automotive case study exemplifies the problems found in embedded systems.

INTRODUCTION

The complexity of developing software systems has increased in the last years due to increasingly complex requirements and need for distribution [1]. In fact, with the advent of the Internet, many software systems that were traditionally local to a single computer started to interact with remote servers. This is especially true with the current push for Cloud based technologies. Examples of distributed systems do not limit themselves to internet related technologies. Complex embedded systems, such as cars, are also distributed systems with the additional complexity of real-time requirements.

A winning strategy to tackle these types of complex systems is decomposition. For distributed systems this means identifying communication interfaces and interaction patterns between each component. A viable approach in this domain leverages service-oriented architectures (SOAs). SOAs promise loosely coupled components and support the challenges of geographically distributed systems, where components are managed by different companies.

This thesis focuses on two domains that have different requirements: embedded and enterprise systems. A first difference is that embedded systems are often real-time while enterprise systems promise only best effort response time. Another difference is that embedded systems are often required to optimize their responses to prioritize critical events while enterprise systems typically strive to optimize the global throughput. Moreover, embedded systems are subject to power constraints and computation power limitations while enterprise systems can often leverage seemingly unlimited resources of large data centers.

In my research, however, I have realized that clear cut boundaries between embedded and enterprise systems have been blurring. On the embedded system side, phones are now shipping with four CPU cores and cars are connecting to the internet and driving themselves. On the other hand, new trends in data centers research focus on minimizing the power consumption of server computers for example leveraging virtualization to share existing resources between multiple applications. Virtualization poses new challenges in the enforcement of quality of service, and requires techniques to prioritize and even move computations.

While some of the requirements of embedded and enterprise systems are converging, there are still big differences. However, I identified a fundamental common denominator in developing both types of systems: managing interactions. From an abstract point of view, the composition of different subsystem entails defining how they interact and ensure that some properties of these interactions are fulfilled. Thus, the remainder of this thesis focuses on methodologies to model interactions, compose them, verify that the models are correct, and use the models in the development of real systems. In particular, I address the key problem of how to compose concerns that are crosscutting the main business logic of the application.

Crosscutting concerns are an active area of research in computer science. Albeit important, crosscutting concerns are not yet fully understood. The most successful treatment of such concerns is articulated mainly at code level with aspect-oriented programming. So far, however, composition of aspects is poorly understood and rarely exploited beyond trivial code examples (e.g. the classical case study for aspects is "logging"). While some work has been done in the context of aspect-oriented modeling [2] and aspect-oriented workflows [3], [4], composition of aspects is still plagued by the problem of unanticipated interactions. Rich Services embodies architectural principles that support composition of crosscutting concerns explicitly, but a language to model the composition of crosscutting interactions is needed. I discuss crosscutting concerns and the Rich Service blueprint in Chapter1.

In this thesis I first show case studies about fault tolerance. Fault tolerance is a crosscutting concern that is arguably more complex than "logging", the standard example on which aspect-oriented programming (AOP) has proven successful. Using thes case studies I can show how complex crosscutting concerns challenge the current state of the art and how Rich Services address the existing problems. In this context, Chapter 2 introduces and enterprise system example, CoCoME, and describes a first contribution: how to leverage Rich Services for fault tolerance.

Leveraging the understanding of crosscutting concerns, applied to CoCoME in Chapter 2, I show how service models can be exploited for verification. Using Rich Services and the architectural principles it embodies, Chapter 3 introduces a formal approach to verify system reliability. In this case, the case study is from the automotive domain: a car central locking system.

While the case studies and approaches presented in Chapters 2 and 3 solve the specific problem of failure management, they do not address crosscutting concerns in general. The key element that is missing in these approaches is a clear articulation of composition of crosscutting concerns.

After understanding the issues of the current approaches in supporting crosscutting concerns composition, the next step is to establish a foundation for composing crosscutting concerns. To this end Chapter 4 discusses how to model crosscutting concerns in general. This chapter introduces an extension to message sequence charts (MSCs) that supports aspects.

A core contribution of this thesis is an orchestration language, which supports crosscutting concerns, and addresses the limitations of other aspect-oriented techniques. This language, that I call Orca, extends the Orc [5] orchestration language. Orc is an elegant language; it addresses many orchestration requirements with just 3 operators, and has a well understood formal semantics. Orca extends Orc by adding operators that extract message flows from expressions variables; the orchestration of aspects is then performed using Orc itself.

Orca provides a formal definition to the modeling technique introduced in Chapter 4, and is a good candidate to act as a general framework to formally describe Rich Services. Chapter 5 presents Orca.

A final requirement for a seamless integration of crosscutting concerns in a model based approach is model consistency. Chapter 6 discusses the issue of model consistency. In particular, it presents an approach to deal with the consistency of different models of the same system, even when the models use different languages, a common situation in real world system development. While this chapter does not use Orca directly as its target modeling language, the system model used in it is an abstraction to which the Orca model "compiles into". The semantic model presented is intended to work with Orca and other modeling languages as well.

Another important part of this thesis is Chapter 7 that covers a broad spectrum of related work, and Chapter 8, which contains my concluding remarks and an outlook on future research.

CHAPTER 1

CROSSCUTTING CONCERNS

In this chapter I outline an approach to software development based on a service-oriented architecture. The goal of this approach is solving the integration issues that still dominate software development. This approach leverages two key elements: models, capturing the application domain and the business logic, and a service architecture that fosters decoupling and separation of concerns. The service-oriented solutions I propose here leverage both an improved architectural pattern and a model-based development process.

The main problem the architecture presented here addresses is the integration of different software features. The goal of the Rich Service pattern presented in this chapter is decoupling the different services and support the composition of features that cross-cut the main application business logic.

Crosscutting concerns are concerns that cross-cut the structure that has been chosen in the decomposition of a problem. For example, at the programming level, crosscutting concerns do not fit neatly in the chosen class hierarchy. The concept of aspect-oriented programming was introduced [6] to solve this issue in software code. Similarly, aspect-oriented techniques have been introduced in modeling languages to address crosscutting concerns at the software model level [7–9].

This chapter presents both an architectural pattern and an associated development process which support managing crosscutting concerns. This approach results from the acknowledgement that crosscutting concerns are an inevitable byproduct of the creation of large systems and successful system development must support them.

APPLICATION DOMAINS

The service-oriented approach I present in this thesis supports different domains with different characteristics. In particular, the approach addresses two seemingly very different domains, large, distributed enterprise systems and embedded automotive systems. While the implementation technologies and some of the requirements of these two domains are vastly different, they have a key element in common, they are complex distributed systems, often developed in different authority domains with the need for reusing existing components.

While the implementation details for these domains are different, the abstract model of how the various part interacts, and how crosscutting concerns must be addressed is similar. In this work I propose the use of model-based engineering. In fact, modeling techniques allows for separate logical and physical models. Logical models specify how systems satisfy their business, functional, and nonfunctional requirements at an abstract level. Physical models specify how the system is implemented and deployed on real hardware.

The use of model-based techniques enables the approach presented here to solve similar problems in different domains. To support integration of crosscutting concerns of large and complex systems, the proposed approach uses a service-oriented logical model. Such model supports the creation of independent services addressing the different requirements of the system. Mappings to domain-specific physical models support the creation of concrete systems in the different domains.

MODEL-BASED ENGINEERING

Various terms are used in the literature to denote the use of models in the development process (e.g., Model-Driven Architecture (MDA) [10], model-based design [11], model-driven engineering (MDE) [12], and model integrated computing [13], [14]). We use the general term MBE for Model-Based Engineering as a superset for all model-based approaches.

To tackle complexity, MBE approaches support multiple perspectives, with associated modeling languages, each focusing on a particular subset of system properties. Each perspective can cover a separate aspect of the same part of the system, or depict the same aspect with different notations to clarify or stress a modeling concept. For instance, we could use a sequence diagram to show the communication protocol between two class instances and two state machine diagrams to describe the proper ordering of method calls. These two perspectives clearly overlap.

A popular language to model enterprise software systems is the Unified Modeling Language (UML) from the Object Management Group (OMG). It comprises many languages (fourteen types of diagrams), each emphasizing a different structural or behavioral modeling aspect. The most recent version is UML 2.3, whose specification consists of the UML Superstructure [15] defining the notation and semantics for diagrams and the UML Infrastructure [16] defining the language on which the Superstructure is based. Constraints can be expressed in the textual Object Constraint Language (OCL) [17].

A specific type of MBE is the Model-Driven Architecture (MDA) [10], an approach that distinguishes between a Platform Independent Model (PIM) and a Platform Specific Model (PSM). The PIM captures the core system entities and their interactions without specifying how these are implemented. PIM can be mapped to multiple PSMs, each capturing all deployment aspects for a given architecture. UML is the language choice of MDA, where both PIM and PSM are expressed as UML models.

MODEL-BASED ENGINEERING IN EMBEDDED SYSTEMS

Embedded systems are often developed by integrating components that have been designed and implemented by different teams, often specialized in different disciplines such as mechanical, electronics, and software engineering. As the system behavior emerges from the interplay of multiple distributed components, a key challenge is the *correct* integration of all these components. System integration is often performed in vertical design chains such as in automotive, and the development chain typically involves several tools that are not integrated.

In the automotive domain, model-based approaches leveraging tools such as MATLAB® [18] and Simulink® [19] from the MathWorks and ASCET [20] from ETAS are used to model control functions and generate implementations for different platforms. However, in practice, there is no formal model for integration that is exchanged between parties. Consequently, it is impossible to validate the design and anticipate problems in putting together the various components in the later phases of the development. The lack of an integrated model also limits the reuse of functions across models and generations of cars.

Models of interactions and MBE hold promise for overcoming these exemplar challenges. Models can serve as a common interface between requirements and architecture specification – using models is the only systematic way to ensure that parties can communicate across all development phases from requirements to acceptance tests. The ultimate goal of MBE is that engineers will spend most of their time modeling the system under consideration, and then generate code for a specific target platform. This goal is already supported by various tools (including MATLAB® /Simulink®), but the models often do not include all aspects of the system, as explained above. When automatic code generation is not feasible at the level of the entire system, there is still significant benefit if modeling is used for requirements gathering and architecture verification before deploying the actual system.

In the past decade, significant advances in the area of model specification, transformation, analysis, and synthesis, have brought the vision of MBE within reach. Challenges for a comprehensive methodology include providing modeling techniques that result in a consistent, integrated specification. Models must be expressive enough to support both generic and domain-specific aspects of the system. Moreover, proper modeling languages must guarantee model reusability, support integration, and enable model execution. To this end, a seamless tool suite that supports the modeling language is a key requirement to make MBE a viable solution.

REQUIREMENTS FOR MODELING LANGUAGES

In the following, I present a set of requirements that a modeling language should meet to support a comprehensive MBE methodology for service-oriented systems. Such language could support both enterprise and real-time systems. I have identified these requirements based on the experience gathered working on several projects. Some of these projects required developing large scale enterprise systems-of-systems, others, in collaboration with automotive industry partners, focused on automotive embedded systems. The list is not intended to be exhaustive, as further requirements have been presented elsewhere (e.g., [21]). The focus of these requirements is the specification capabilities of a modeling language for services. The requirements do not cover the entire end-to-end MBE approach or the tools necessary to implement it.

- **Consistency.** A modeling language should allow grouping of requirements, structure, behavior, and analysis in a single, integrated system model. Therefore, the language should allow consistency checking for models expressed in different notations, developed in different design iterations, or models that are part of different views/slices of the same system.
- **Traceability.** Requirements should be mappable to a precise specification of the system and from there to implementation while the mapping should be kept current during the system evolution. Traceability also applies to models at different levels of abstraction enabling conformance checking for refinement operations.
- **Realizability**. Models often represent partial specifications that are refined in successive iterations in the development cycle. Models also represent different views on the system. The underlying question is whether the models allow a system to be constructed such that all requirements are fulfilled. At the very least, we would like to know which requirements stand in the way of realizability.
- **Distribution and integration**. System behavior emerges as the interplay of the functionality provided by sub-systems, often developed independently by different parties. Thus, models should be capable of expressing concurrency, synchronization, and integration constraints. For example, in the automotive domain the Original Equipment Manufacturer (OEM) is

responsible for the integration of sub-systems. Modeling should support overarching system specification addressing the integration requirements as well as concerns that cut across the individual components such as resource optimization across the integrated system.

- **Interdisciplinary domains**. Embedded systems design involves multiple domains such as mechanical, electronics, and software. The system components are often designed at different stages in the development process, by different teams, using different tools and languages. A common modeling language should ease integration and tradeoff analysis, and it should reduce the need for *disruptive* feedback iteration cycles.
- **Non-functional properties.** A modeling language should allow specifying non-functional properties (e.g., security, authentication requirements, performance, reliability, and power consumption) associated with behaviors, refinement relationships, deployment models, etc. Moreover, because my goal is to have a modeling language that can support multiple domains, the set of non-functional properties should not be predefined and the language should support the specification of application-specific properties.
- **Resource models.** Those models are useful both for enterprise and embedded systems. Enterprise systems can often run on clouds or grids where the hardware is shared between multiple applications. Therefore, the ability to clearly specify the resource requirements of the application is

key to properly define a service level agreement with the resource provider and fairly share bandwidth and computation power with other applications. On the other hand, embedded systems interact with the physical world, and are constrained by the resources provided by the hardware and software platforms. Therefore, a specification should support modeling of platforms and resources, as well as allocation and optimization of resources to meet functional and non-functional requirements.

- **Timing**. Time plays a critical role in real-time systems and, therefore, a modeling notation should express timing requirements in various temporal models: (i) causal models, which are concerned only with the order of activities, (ii) synchronous models, which use the concept of simultaneity of events at discrete time instants, (iii) real-time scheduled models, which take physical durations and the timing of activities as influenced by CPU speed, scheduler, utilization, etc., into account and (iv) logical time models (e.g., Giotto [22], [23]), which consider that activities take a fixed logical amount of time, assuming that the platform can execute all activities to meet their constraints.
- **Heterogeneous models of computation and communication**. Real-time systems are often embedded systems that control physical processes, which are often represented in terms of mathematical models. A modeling specification should support continuous behaviors, discrete event-based or time-based behaviors, or combinations thereof.

Some requirements expressed here are common to all applications domains, while others (e.g., heterogeneous models of computation and communication) are specific of the embedded domain. However, a language that wants to address all these domains should satisfy all requirements.

RICH SERVICES PATTERN

SOAs have emerged as an accepted solution to integrate heterogeneous systems. SOA-based approaches typically use standards-based infrastructure to map existing systems into standardized forms of services. Services can then be orchestrated by means of choreography engines and specialized languages in different ways to provide new business value to different stakeholders. New functionality can be created by either adding new services or changing the message flow among existing services. Because of these features, many SOA projects are particularly amenable to agile development processes.

SOAs are typically flat, meaning that a composed service is obtained by finding and aggregating different services available across the net. There is no structure or interface in place to decompose the system according to different concerns. Moreover, each composed service needs to take into account not only the main business concerns but all the crosscutting concerns such as encryption, authentication, failure management etc. However, the integration of the particular concerns of the enterprise and automotive domains requires a scalable framework that provides decoupling between the various concerns and allows for subsystem integration and multiple hierarchical decomposition choices. The Rich Services architecture [24] is a type of SOA that addresses these issues and provides a direct and easy deployment mapping to various middleware including Enterprise Service Buses (ESBs) [25].

Figure 1 depicts our logical service-oriented architecture, inspired by ESB architecture/implementations [25–27]. The main element of the architecture is the notion of *Rich Service* [24], which, in the most generic form, encapsulates various capabilities and functionalities pertaining to the business logic and the applicable concerns.

We start with a set of *Rich Application Services* (RASs) ❶ that encapsulate core application functionality, defining the business flow. In most applications, existing services (provided by subsystems or system components)

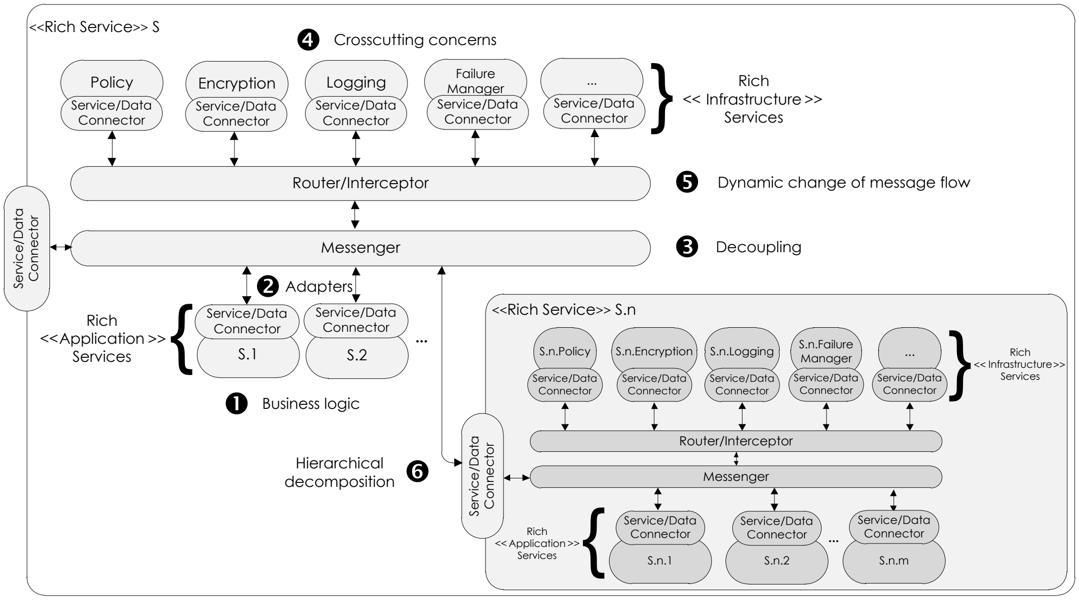

Figure 1. Composite Rich Services

can map 1:1 to RAS. To facilitate the interaction between services and hide their internal complexity, we attach to each RAS a Service/ Data Connector ❷, which performs the necessary adaptation of the service inputs and outputs, presents the service capabilities to other services, and encapsulates their value-added internal logic. In the tradition of [28], [29], we associate both a structural and a behavioral view with the Service/Data Connectors.

The RASs are decoupled through a message-based communication infrastructure. The *Messenger* ❸ layer is responsible for transmitting messages between services and provides the means for implementing the service orchestration. Encapsulated Rich Services are connected to the communication infrastructure via their own Service/Data Connectors ❷. This approach is very important for future-proofing system design – the Connectors do not just integrate existing services; they also prepare services for future integration into other larger systems.

We then focus on the crosscutting concerns expressed as services. Using the same architectural pattern, we distinguish between the *Rich Application Services* ❶ (RASs) and *Rich Infrastructure Services* (RISs) ❹. In contrast with RAS that implement business logic, the RISs do not initiate any communication by themselves, but reroute or filter messages defined by RASs. Examples of RISs are policy enforcement, encryption, and authentication.

The *Router/Interceptor* ❺ layer intercepts messages placed on the Messenger and then routes them among all services involved in providing a particular capability. RISs connect to the Router/Interceptor to define the proper RASs orchestration. Hence, new services can be plugged into the architecture without changing the existing services. To integrate an encryption mechanism, for instance, only the communication infrastructure needs to be aware of the encryption RIS: the Router/Interceptor changes the routing tables to ensure that every message sent to the external network is first processed by this service.

This two-layer communication infrastructure enables loose coupling and seamless communication between services. The use of a Router/Interceptor layer removes dependencies between services and their relative locations in the logical hierarchy. Thus, services from different levels of the hierarchy - possibly from different authority domains in the case of large business systems - can interact with each other seamlessly with the help of appropriate infrastructure services and routing tables.

To address complexity, any Rich Service, instead of being a simple functionality block, could be hierarchically decomposed into further Rich Services ❻. A Rich Service S that exports functionality to some client – perhaps an external integration framework – is implemented by RASs S.1, S.2, through S.n., along with the associated RISs (Figure 1). Likewise, Rich Services S.1 and S.2 are shown as simple Rich Services whose interfaces are defined by Service/Data Connectors. Rich Service S.n is shown decomposed into another Rich Service, whose interface is also defined and exposed by a Service/Data connector; it has its own message bus, router, RASs, and RISs. Note that both

RASs and RISs can be further decomposed into Rich Services. In addition, RISs (e.g., Policy used in different Rich Services) can be instances of the same service, different services, or parts of a larger crosscutting service from a different system model view. This design strategy enables flexible deployment choices that can take into account specific platform enhancements (including, but not limited to, hardware-optimized cryptographic engines, enterprise -wide access control lists (ACL), policy enforcers).

MODEL-BASED DEVELOPMENT PROCESS

The Rich Services hierarchical framework manages the complexity of enterprise systems-of-systems integration and automotive applications by decomposing complex problems into primary and crosscutting concerns, providing flexible encapsulation for these concerns, and generating a model that can be easily leveraged into a deployment. The associated model-based development process [30], outlined in Figure 2, encompasses activities from the high-level use case elicitation through physical network deployment. The top part of the picture represents the logical architecture loop, which deals mostly with platform-independent models (PIM); on the other hand, the lower part of the picture represents the deployment loop and entails the creation of platform-specific models (PSM).

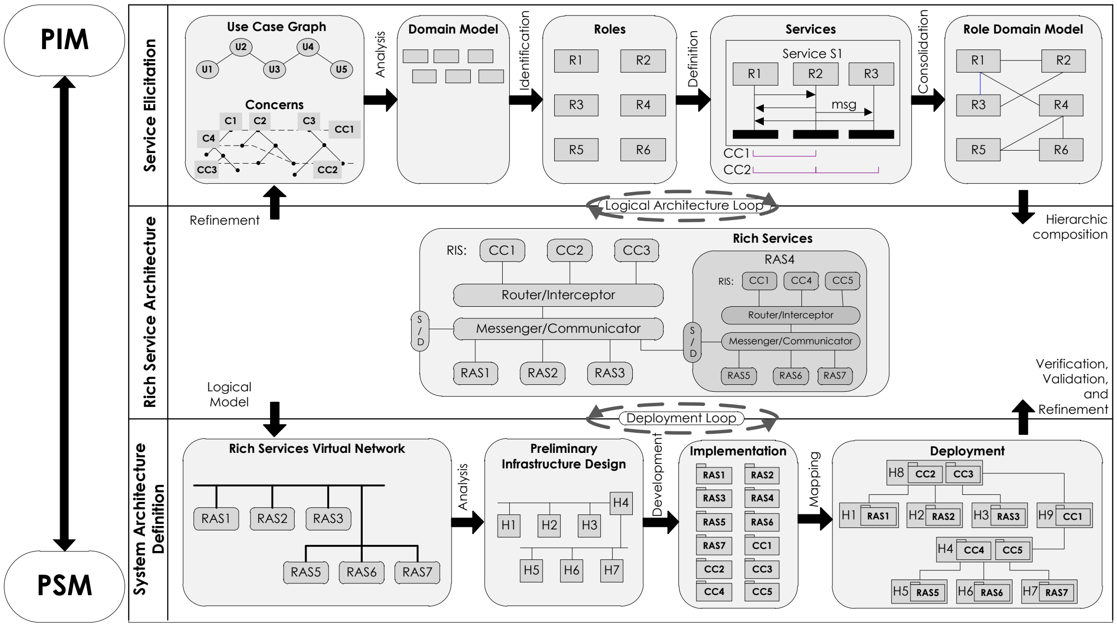

Figure 2. Model-based development process for Rich Services

Our process leverages the spiral development process [31] model and embraces agile development methodologies. Requirements often resolve to partial specifications, and refinements or additions of requirements at one stage can trigger iterations beginning at some appropriate earlier stage. Thus, the artifacts produced at some stage are fed back into new iterations of the development process where they are revisited and refined.

This iterative process accommodates *architectural spiking*. This means taking a partial set of use cases and generating a system architecture and implementation based on them, then adding more and more use cases over subsequent rounds. Architectural spiking allows domain and application knowledge to be developed incrementally instead of in grand exercises, thereby managing complexity and mitigating development risks.

A major feature of the Rich Services development process is that crosscutting concerns are identified early. In fact, addressing such concerns as afterthoughts increases the integration costs and leads to incomplete or incorrect system implementations

The first phase, Service Elicitation, captures the system requirements in a service repository. At the same time a domain model is created. This model integrates the main business concerns with functional requirements and crosscutting system concerns such as security, access control, encryption, fault tolerance, tracing, and transaction support. For each concern, we can leverage an existing technique of requirements gathering. For instance, for security we can employ elements from the Common Criteria [32] to determine assets, risks, and mitigation strategies. In the automotive domain, the approach of SPUR [33] can be used for modeling increasingly important attributes such as security, privacy, usability, and reliability.

We define services as interaction patterns between roles [34] for the realization of each use case. The service repository is the collection of all identified services. Each service "orchestrates" interactions among system entities to achieve a specific goal [35]. Within a service, roles exchange messages, thereby switching from one state to another. A Message Sequence Chart [28] can be used to capture the interaction and various role states. The complete set of states and state transitions of a role obtained from all services in which it participates defines its full behavior.

In the second phase, Rich Services Architecture, we use the role domain model and the service repository to define a hierarchic set of Rich Services as a logical model of the system. Of the many possible hierarchical decompositions, the ones we choose for consideration are driven by client values such as architecture comprehensibility, business manageability, performance, and organization domains. The most important value determines the dominant decomposition.

In the process of creating RASs and RISs, it is common to discover additional opportunities for crosscutting processing such as Quality of Service (QoS) property monitoring [36], failure detection and mitigation [37], and role interaction monitoring. Strictly speaking, these concerns reflect functional and non-functional facets of requirements, which may generate additional use cases resulting in re-iteration of one or more stages of the Service Elicitation phase. However, for crosscutting concerns having only local effect, such iterations can be safely deferred. For more global concerns, spiraling back to a previous development stage is usually warranted. In complex systems as encountered in the avionics and automotive domains, domain modeling may result in multiple largely orthogonal system views representing corresponding crosscutting concerns. In such situations, each view may generate its own Rich Service model, and each view may be represented as one or more RISs in the models for the other views.

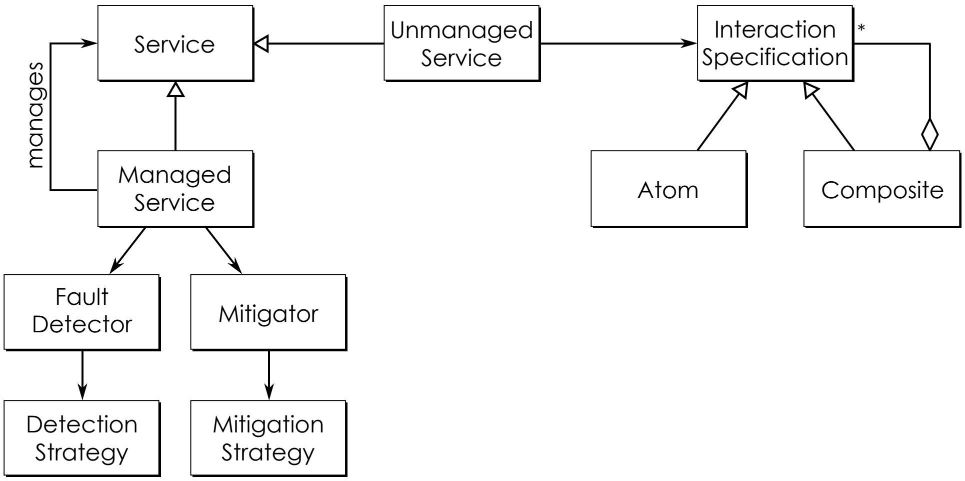

Figure 3. Domain model for services addressing failure management

In Figure 3, for instance, we depict a class diagram capturing the relations between services and failure management. We see two types of services. Unmanaged services address regular interactions that do not take into account the crosscutting concern of managing failures. The model is enriched with managed services which take care of detecting failures and recovering from them. Following the Rich Service approach we have decoupled the two issues. We first define the regular services (unmanaged); then, we group the concern of detecting failures and managing them in a different service (managed), which modifies the regular service behavior. This is represented in the class diagram by managed services having a reference to the service they manage.

The collection of Rich Services defined at this point represents the Rich Services logical model of the system, or Rich Services Repository, which is transformed in the System Architecture Definition phase into a deployment model and implementation. We model interactions between RASs as messages across a communication facility, which are subject to routing by a Router/Interceptor layer. Interactions involving RISs are implemented by using the Router/Interceptor layer to weave RAS-to-RAS messages into a RAS-to-RIS-to-RAS template (or other similar interception-based processing). With such an interception capability, using role interaction monitors and Quality of Service monitors to assess the correctness and quality of services is possible without disturbing the RAS (or RIS) processing already in place. Similar reasoning applies to failure detection/mitigation and other crosscutting processing. Because the logic needed to orchestrate the message flow is captured by MSCs [28], we can leverage our work on state machine generation to synthesize the routing required [34]. Alternative options are to describe the orchestration logic by means of Web Services Business Process Execution Language (WS-BPEL) or, a discussed in Chapter 5, using Orca.

The third phase, System Architecture Definition, establishes a relationship between the Rich Services model of the system and its implementation. We first inventory and analyze the subsystems and software components already available, the topology of the existing systems in terms of computational, input/output, control, and storage nodes, and available networks. Operational and maintenance use cases are also refined at this point to address the system evolution concerns.

We create an idealized network of the identified Rich Services, where each RAS is represented as a *virtual host* connected to a common bus. This stage focuses on logical connections between RASs and their message exchange patterns, recognizing that duplication of services may occur. Analysis of the message flow, volume and frequency of the data exchange, and the relationships between the virtual hosts to implement the business logic is important to identify possible bottlenecks and best places to address the crosscutting concerns.

Based on the virtual network of the previous stage and the current Rich Services architecture of the system, we design a preliminary infrastructure to accommodate the current understanding of the system. With the platform identified, we can proceed to develop the code for each Rich Service.

In the last stage – deployment, we perform a mapping of the implemented Rich Services to the middleware running on physical hosts. Depending on the system requirements and the available resources, some services can be duplicated, whereas other duplicated services can be replaced with proxies for unique services. Furthermore, levels in the hierarchy can be flattened. The resulting view is still a projection of the overall system model, just tailored for a particular deployment platform to yield better performance, or to improve some other relevant quality aspect. Approaches such as [38] can be incorporated into a Rich Services process, thereby shifting the virtual network mapping to runtime, and taking advantage of a network manager's ability to dynamically allocate services to available network resources.

SUMMARY

In this chapter I presented an architectural pattern and a development process that support separation of concerns and enable composition of crosscutting concerns. In the rest of this thesis I will present specific examples of crosscutting concerns, identify shortcomings with the existing approaches for addressing them, and provide a new modeling language to address these shortcomings.

ACKNOWLEDGEMENT

This chapter, in part, is a reprint of material as appeared in C. Farcas, E. Farcas, I. H. Krueger, and M. Menarini, “Addressing the Integration Challenge for Avionics and Automotive Systems - From Components to Rich Services,” in The Proceedings of the IEEE Special Issue on Aerospace and Automotive Software, K. V. Prasad (Ed.), vol. 98, no. 4. IEEE, Apr. 2010, pp. 562-583. The dissertation author was the primary investigator and author of the text used in this chapter.

CHAPTER 2

FAULT TOLERANCE: A CASE STUDY IN HANDLING CROSS-CUTTING CONCERNS WITH RICH SERVICES

In the previous chapter I presented an approach for modeling the crosscutting concerns using Rich Services. In this chapter I discuss an example of crosscutting concern that is fundamental to both embedded and enterprise systems: fault tolerance. In fact, building reliable composite systems from unreliable services has emerged as an important aspect of system of systems integration and has captivated researchers' attention in recent years. Many attempts have been made towards building reliable and fault tolerant services [39], [40]; however, a comprehensive approach has yet to emerge.

A viable solution for failure management has to fulfill few key requirements. First, it has to be scalable and support geographical and organizational distribution. To that end, a the solution should separate the business logic from the fault handling logic, making it possible to add fault tolerance to composite services that are only available as Commercial-Off-The-Shelf (COTS) applications. Second, because the supporting technology in this domain is rapidly evolving, a solution should be technology independent. Various failure types in the application domain, such as servers failing to respond, or race conditions in accessing a database concurrently, should be addressed, and appropriate recovery policies should be exploited and activated based on the type of failure detected [41]. Self-healing architectural approaches such as [42] suggest that the detection mechanism should be decoupled from the entities that provide the recovery. This decoupling facilitates the reuse of both types of components in addition to dynamic modifications to recovery policies without the need to modify the detection mechanism.

Rich Services, thanks to the decoupling provided by RIS and RAS, are good candidates to address the set of requirements identified for failure management. This chapter presents an approach to address fault management in enterprise systems and uses the CoCoME system as case study.

THE COCOME CASE STUDY

The CoCoME Trading System case study models an Enterprise with a collection of Stores (e.g., department stores). The application simulates the behavior of a typical enterprise: customers pick up products and pay for them at the cash desks, managers can view stock reports, and they can order products from various suppliers to restock the store. Each Store has an Inventory that keeps track of the locally available products. There is also an enterprise-wide repository – Enterprise Repository – that can be synchronized with the Inventories of each store. All systems (Stores, Enterprise, Bank, Suppliers) communicate by means of Web Services.

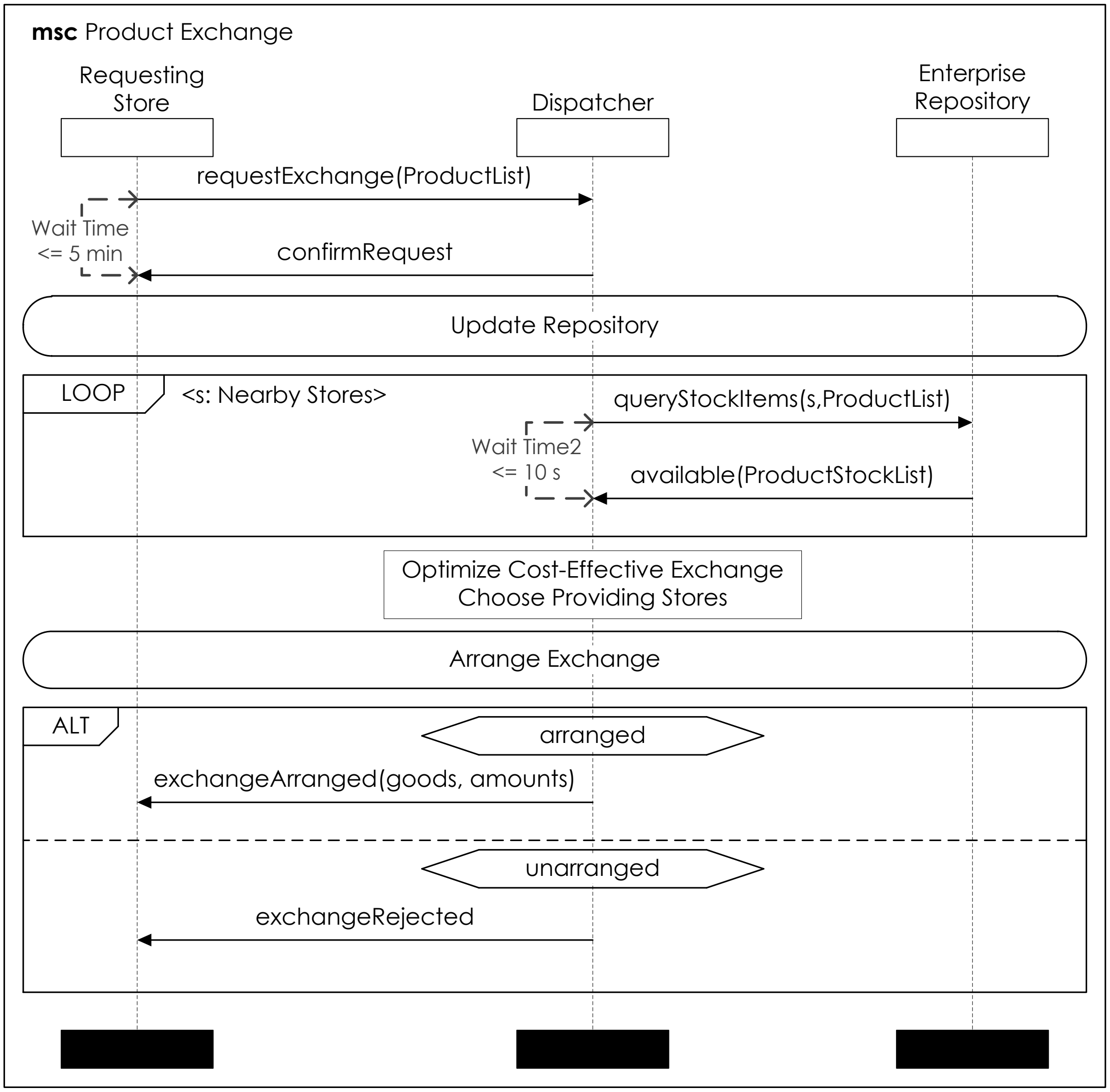

Figure 4. Product Exchange interaction specification

Here I focus on a particular use case – Product Exchange. If a Store is running low on some products, it can request to receive supplies from other nearby stores. If some nearby store has enough stock of the requested products, a product exchange is arranged. This process is managed by an enterprise-wide Dispatcher. Figure 4 depicts this use case using a simple MSC.

The Trading System case study presented here was derived from the case study for the CoCoME modeling contest [43]. The version presented here is slightly modified to include support for failure management. This modified CoCoME allow for the evaluation of the failure management technique for the Web Services presented in this chapter.

Figure 5 depicts part of the logical model for the Trading System. It captures the structural elements that appear in the case. For the purposes of this example the model includes three Stores in the Enterprise. The internal structure of the Stores is not shown in the figure.

Figure 4 depicts the interaction of the Product Exchange use case that forms the basis of the case study. A Store (playing the role Requesting Store) requests the Dispatcher to receive some products from a nearby store (requestExchange(ProductList)). The Dispatcher confirms that the request has been accepted and starts the Update Repository interaction. When the Enterprise Repository has been synchronized with the Inventory of the Nearby Stores, the Dispatcher gathers the data on the requested products, decides

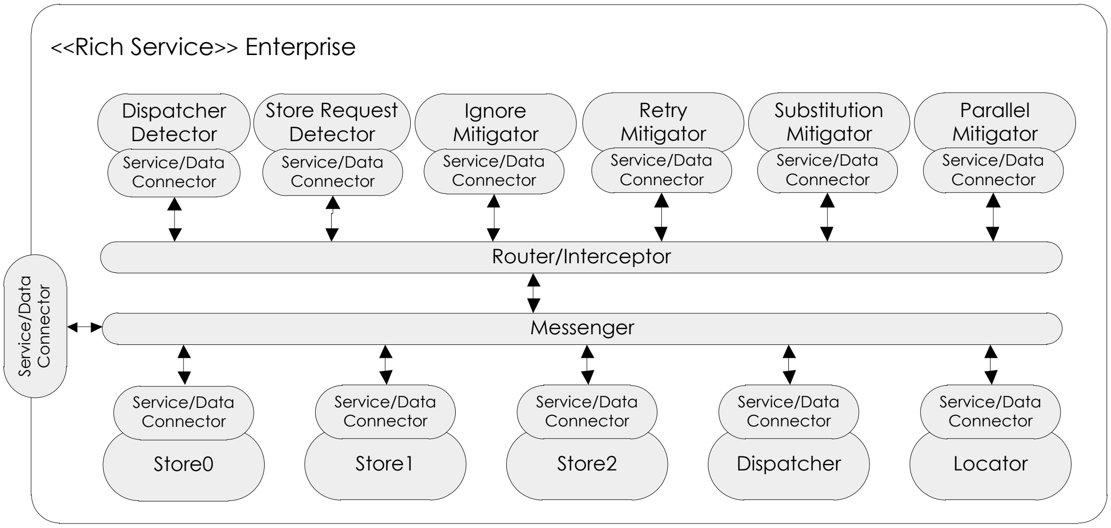

Figure 5. Trading System, Enterprise Rich Service view

which stores should provide them, and arranges the Exchange. If this process is successful, the Dispatcher returns the information about the products to the Requesting Store; otherwise, it informs the Requesting Store that the exchange has been rejected.

In the Update Repository interaction (Figure 6), the Dispatcher requests the Store Locator to identify the nearby stores and to send them a request for flushing their recent inventory updates. This activity is needed to synchronize the Enterprise Repository with the Repository of the Stores. The store locator identifies which stores are nearby the Requesting Store and forwards the request to them. Nearby Stores then flush their inventory to the Dispatcher, which updates the Enterprise Repository accordingly and confirms the

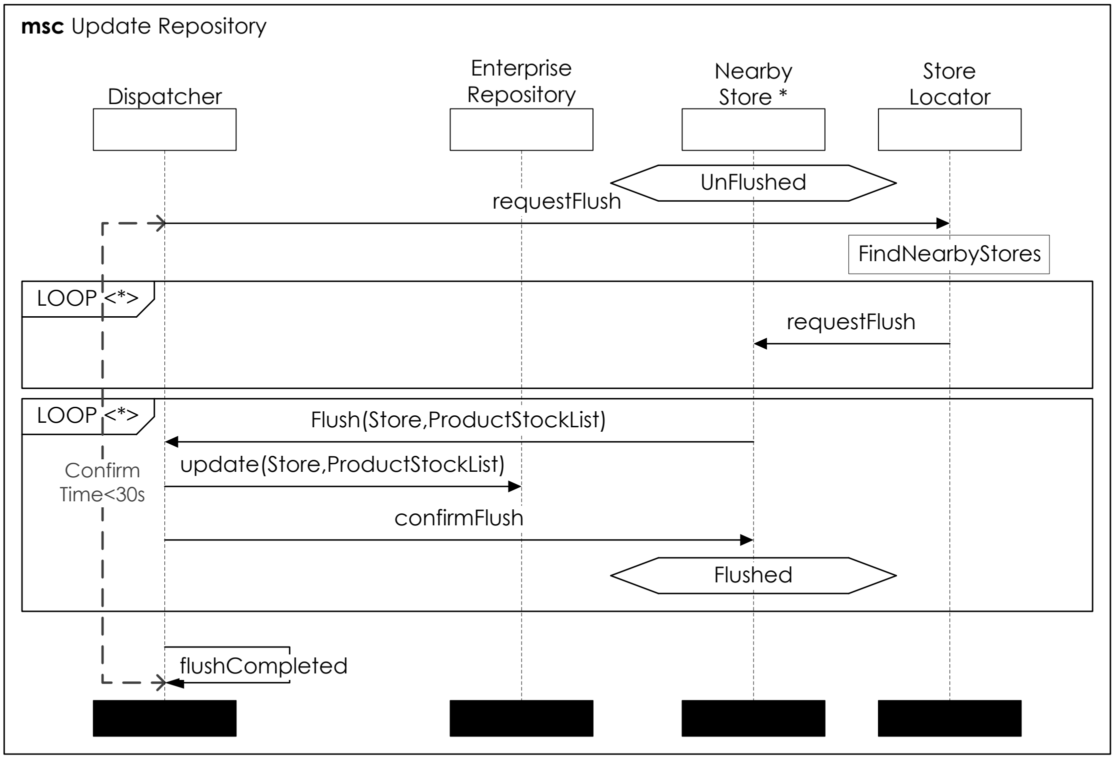

Figure 6. Update Repository Interaction

reception of the data. When all Nearby Stores have flushed their data, a flush complete message is generated. A deadline of thirty seconds is defined between the request event and the flush completion.

RELIABLE WEB SERVICES

Reliability of Web Services is an active area of research. In order to address fault tolerance in Web Services, first we need to identify and classify the typical failures that can occur during service execution. Three classes of failures are identified in the literature: behavioral and business logic failures, operational failures and quality of service failures [44]. Business logic failures refer to cases where the service fails to complete its task, or delivers incorrect results, because of computational or logical failures. Operational failures refer to failures of the underlying middleware and communication infrastructure that the hosting servers rely on. Quality of service failures refer to failures in delivering the service with the promised quality of service properties. Service Level Agreement (SLA) failures are in this class. An SLA is a contractual agreement between a service provider and its consumers. It often mandates response times and other quality of service metrics. The focus of this chapter is on behavioral and quality of service failures.

A second step is to investigate failure detection mechanisms. Two predominant levels of detection have been studied in the literature. The simple case is to detect and mitigate the failures at the single invocation level. The approach presented here supports a more complex failure detection based on an interaction pattern among multiple participants/roles in service choreography. As discussed later in the chapter, current Web service techniques focus mainly on exception handling techniques resulting in single invocation based detection.

A model-based approach to failure management for Rich Service-based SOAs is introduced as follows. First I present a failure model, including the notion of a failure hypothesis. Then I introduce an approach to defining Detectors that identify occurrence of failures at run time. Finally, I introduce strategy-based Mitigators that provide recovery mechanisms after a failure is detected. This approach has the following benefits: 1) it can base the identification of a failure on the system model, 2) the logic to detect an error is separated from the logic to recover from it, 3) the mitigation strategies can be reused in different contexts.

FAILURE HYPOTHESIS

A failure hypothesis captures the assumptions about which parts of the system can fail, and how many faults are allowed to happen concurrently (and in which combinations) for the system to still be considered fail-safe. In a model-based approach, this information is used as an input for validation and verification activities.

I consider mainly two types of failures: failures where a service does not complete its task, or where an error causes an unexpected message flow. The approach presented in this chapter can deal with both types of failures. It can detect unexpected message flow by comparing the messages exchanged with the sequences defined in the MSCs. Furthermore, the model presented here allows for the specification of deadlines between events enabling the detection of failures where messages are not sent. This capability is important in the web service domain to address the requirements of SLA. Deadline assertions can be leveraged to encode SLA, and detect possible violations.

In the CoCoME Trading System case study, the failure hypothesis is as follows: Stores can get permanently disconnected, Stores can fail intermittently (refusing to respond with a given probability), and the Dispatcher can fail by not identifying that all Stores have flushed. For example, let's analyze a subtle race condition that can occur in the implementation of the case study. Consider the following scenario. An arbitrary number of nearby stores can be selected as candidates for an exchange and requested to flush their inventories. The Dispatcher uses a relational database to keep track of how many stores have answered. At the end of each flush cycle, the Dispatcher decides if the update is finished and, if so, it triggers the *flushCompleted* message. The use of a database for storing this information is justified by the fact that, for scalability reasons, the Dispatcher is implemented as a set of servers that listen to flush requests. The problem is that depending on the isolation level for transactions supported by the database, a race condition can occur such that the information about a store flush is lost. This could be observed if the database used in the implementation supports only the dirty read isolation level. This is a failure that falls into the category of Dispatcher not identifying that all stores have flushed.

In my experiments, I have added failure injection code to the system to trigger the failures defined in the Failure Hypothesis. Thus, even failures that seldom occur, such as the mentioned race condition, can be evaluated quantitatively.

FAILURE DETECTION

Recall that the model of a service with failure specification consists of an interaction pattern, augmented with deadline specifications. From this model, failure Detectors we can automatically derived. Such detectors monitor the compliance with both the interaction pattern and the deadlines [45]. Because the Rich Service architecture implements such interactions via a messaging infrastructure, Detectors can leverage the interface between the Rich Services and the messaging infrastructure to detect differences between the runtime behavior and the model. In the Web Services domain, the implementation of the participating services of a composite Web Service may not be available to the developer. Usually these are heterogeneous services from different geographical and authority domains. The proposed technique, by leveraging the communication infrastructure in order to detect and mitigate failures, enables the implementation of reliable composite services. Because it does not require any change to the constituent services, this technique is a good candidate for addressing failures in the web service composition domain.

The Rich Service framework provides this capability by enabling the RISs to intercept and reroute messages transparently. A RAS does not need to be aware of the RISs that process and possibly even alter the messages the RAS sends and receives. As a result, RISs can be used to construct Detectors – this decouples RASs from the corresponding Detectors. Each Detector receives and monitors the messages specified by an MSC and detects a failure if the communication does not match the specified interaction pattern.

In order to guarantee that an interaction failure can be detected within finite time, a deadline between the starting and ending messages of the interaction can be specified. This also captures SLA in the model and supports detecting potential SLA violations. For example, Figure 6 shows a deadline of 30 seconds for the Update Repository service to complete.

As a result, the approach proposed can detect behavioral failures where a service does not respond within the defined time interval, or when the interactions take place with messages in an order different than specified. Upon detection of a failure, the Detector activates the Mitigator responsible for recovering from the detected failure.

FAILURE MITIGATION

In the literature ([42], [46], [47]) two classes of mitigation strategies are applied to the web service composition domain: backward and forward recovery. Backward recovery is mainly achieved by transaction-based mechanisms resulting in a "roll-back" of actions that have not gone through as a whole. The main forward mitigation strategies studied in the Web Services domain are: ignoring the failed invocation (*Ignore*), retrying the same invocation to the same service (*Retry*), and retrying the same invocation but substituting the called service with an alternative but "equivalent" service (*Substitute*). Another fault tolerance pattern is the parallel execution of the same invocation on multiple equivalent services and taking the first response or voting on all responses (*Parallel*). In most cases, a combination of these strategies is applied (*Composite*), such as retrying for a specified number of times and then substituting the callee with an alternative service. The work in this chapter focuses on forward recovery mitigation strategies.

The goal is to reuse the standard mitigation strategies mentioned above. Mitigators are logical components activated by Detectors when a failure is identified. Mitigators are specified as RISs and are responsible for recovering from the failure; they are given enough context information to be able to apply the standard mitigation strategy correctly.

We can analyze, for instance, how the *Substitute* strategy is applied in the case study. A generic Mitigator sends the request to an alternative service in case of a failure. In our case study, the request is the *flush* request sent to *Store 1*. If the Detector identifies that the request has failed, it triggers the substitution Mitigator providing, as context information, the failed message sent to *Store 1* with its parameters. An application specific configuration defines the alternative service provider for *Store 1* (*Store 3*). Different Detectors can trigger the same Mitigator implementation to deal with failures of different Stores, provided that they supply the correct context information.

The case study uses the *Ignore* strategy to mitigate the Dispatcher failure described earlier. In this case, the Detector invokes the ignore Mitigator if the 30 second deadline is not met. The Detector provides the Mitigator with some context information about the state of the system and the next required action as parameters. The Mitigator receives a state machine that defines the expected message sequence and the current state in the message sequence. If the current state is such that all flushes have been performed but no message flushCompleted has been sent, the *Ignore* Mitigator executes the next step defined by the state machine sending the flushCompleted message.

Retry, *Substitute*, and *Parallel* are used in the case study to mitigate the Store connection failures. The failure hypothesis defines different ways a Store can fail to respond. It can, for example, fail intermittently or it can stop responding forever. To evaluate which mitigation strategy better addresses which failure, next section presents a set of experimental results.

Finally, because in the presence of multiple failures a single mitigation strategy does not provide complete reliability, the approach proposed supports *Composite* mitigation strategies. A *Composite* Mitigator is achieved by adding multiple Detector / Mitigator pairs identifying and recovering from the same failures. If the first Detector detects a failure, the associated Mitigator is triggered. The second Detector continues to observe the messages exchanged and can execute the second mitigation strategy in case the first one was unsuccessful.

IMPLEMENTATION AND EXPERIMENTS

In order to measure the reliability and the performance overhead of the approach proposed, I implemented the Trading System case study and conducted a number of experiments. This section describes the implementation details and the experimental results.

The challenge in implementing the case study was (a) to find a mapping of Rich Services into a deployment architecture, and (b) to leverage this deployment architecture for representing Detectors and Mitigators at runtime. I address challenge (a) by using Mule, an open source Java ESB, as an implementation platform. I implement RAS as Mule Universal Message Objects (UMOs) (for details, see the following paragraphs). I address challenge (b) by using Mule interceptors to implement Detectors and Mitigators.

Implementation based on Mule. The Rich Services architectural blueprint is inspired by ESB technologies. In particular, the Messenger and the Router/Interceptor components of the blueprint map nicely to the infrastructure provided by various ESB platforms. To establish the independence of the Rich Services architectural pattern and my fault tolerance approach from a specific implementation platform, I deployed the same service- and failure models on two different ESB target platforms: Mule ESB [25], and ServiceMix [48]. Both are open source frameworks based on Java; however, they reflect different implementation decisions. Mule is lightweight and does not mandate a specific format or medium for messages being exchanged. ServiceMix, on the other hand, converts all messages to an Extensible Markup Language (XML) standard format and uses ActiveMQ to transfer them between services. For reasons of brevity, in the remainder of the paper we focus on the Mule implementation.

Use of UMOs to implement RAS. The main functionality blocks of any Mule application are the UMOs. They are Java classes that are instantiated by the ESB and send and receive messages via Mule Application Programming Interface (API). Mule provides facilities to connect such UMOs to a variety of communication technologies. In particular, the case study implementation exposes services as standard web services using Hypertext Transfer Protocol (HTTP) and Simple Object Access Protocol (SOAP). Mule facilitates the creation of UMOs by allowing the programmer to write regular Java classes and an XML configuration file. The ESB uses Java reflection to identify the right method to call depending on the message received. Mule UMOs are used to implement Stores, Dispatcher, and Store Locator as separate application services.

Use of Mule Interceptors to implement Detectors and Mitigators. The implementation of Detectors and Mitigators is done using Mule Interceptors. Interceptors are used in Mule 1.x series (the case study implementation uses Mule 1.4.3) to transparently inject additional behavior when messages are sent or received by services. In the Mule framework, using Interceptors is the logical choice for implementing a Detector that observes the messages exchanged and compares them with the expected communication modeled by an MSC. Mitigators are also executed as part of an interceptor. The interceptor first detects if a failure has happened, then allows the specified Mitigator to process the message as needed. Mitigators can use the Mule API to dispatch additional messages. For example, the *Parallel* and *Substitution* Mitigators use application specific information on an alternative store to send the request flush to. The application is not aware of the fact that the request is forwarded to an alternative store. The *Parallel* Mitigator, in particular, sends the request to the alternative store even if there is no failure. Only the first flush received is passed to the Dispatcher. I performed experiments with multiple mitigation strategies to demonstrate the flexibility of the approach; changing between different strategies and combining different strategies was a matter of minutes.

Implement Detectors using state machines. Each Detector has a state machine associated with it that recognizes the language specified by the MSC [45]. This is convenient because, as discussed earlier, these state machines can be automatically generated. The detector can, therefore, observe the messages exchanged and take the corresponding transition in the generated state machine. If no transition exists with the given message guard the Detector identifies a message order failure.

Table 1. Failure management experimental results

Experiments		Detectors/Mitigators				100 Requests		Faults	Request Time (ms)		
		ret	ign	sub	par	Success	Failure		Max	Min	Avg
One Flush per Request	No Failure	X	X	X	X	100	0	0	8667	206	2109
	No Failure	✓	✓	X	X	100	0	0	9385	226	2258
	10% Dispatcher Failure	X	X	X	X	91	9	9	8167	204	2191
	10% Dispatcher Failure	X	✓	X	X	100	0	9	30921	170	4684
	10% Dispatcher/Stores Failure	X	X	X	X	74	26	26	10141	193	2039
	10% Dispatcher/Stores Failure	✓	X	X	X	92	8	15	7256	229	1928
	10% Dispatcher/Stores Failure	✓	✓	X	X	100	0	23	30401	172	5200
	100% First Store Failure	✓	X	X	X	0	100	200	N/A	N/A	N/A
	100% First Store, 10% Backup Store Failure	X	X	✓	X	90	10	110	11561	170	2463
	100% First Store, 10% Backup Store Failure (order: retry, substitute)	✓	X	✓	X	95	5	205	11873	273	2653
	100% First Store, 10% Backup Store Failure (order: substitute, retry)	✓	X	✓	X	100	0	108	12042	218	2481
	No Failure	X	X	X	✓	100	0	0	13070	238	2936
	10% Stores Failure	X	X	X	✓	100	0	17	11197	145	2709
Two	10% Dispatcher Failure	X	X	X	X	93	7	7	9636	337	2888
	10% Stores Failure	✓	✓	X	X	98	2	21	30035	404	3482

Keep track of the session identifier. Because of the concurrency of client requests, multiple interactions implementing the same pattern are active in the system at a given time. All calls carry a session ID that enables Detectors to update the state machine and other information on a per session basis. Furthermore, each Detector can start a timer is deactivated by receiving the message that ends the deadline. If the timer is not deactivated, the Detector triggers the specified Mitigator depending on the current state of the system when the timer expires. The timers are implemented using the

Quartz Java library. For each session the Detector keeps track of the state machine's state, the timer state, and the message parameters.

Experiments Description. The first two columns of Table 1 describe the type of experiment executed. The difference between experiments with One Flush per Request versus the ones with Two is in the Store locator behavior. In the first case, the store locator identifies just one nearby store and issues just one flush request. In the second, two nearby stores are identified and two requests are issued. The second column describes the failure injection behavior during the experiment. The Failure injection is performed using a certain probability function. The first type of failure we inject is in the Dispatcher. A 10% Dispatcher Failure means that every time the flush is concluded (all stores have flushed), there is a 10% probability that the dispatcher does not recognize it and does not issue the *flushCompleted* message. In a run with 10% Stores Failure, with a 10% probability a store might not respond when requested to flush. 'Two' experiments show a more complex failure injection. 100% First Store, 10% Backup Store Failure means that the store that is requested to flush is completely down (always returning an exception) but the backup store (used by the Substitute mitigation strategy) has only a 10% chance of failing. In the 'two' experiments, the order in which the mitigation strategies are applied is important. Therefore, the table also contains the order in which the strategies are applied. All results in the columns named Success, Failure and Faults are based on 100 exchange request runs. All timings are expressed in milliseconds. In the

Detectors/Mitigators columns, specifying the retry, ignore, substitute, and parallel execution mitigation strategies, we have a ✓ if the Mitigator and the corresponding Detector are loaded in the system, and an x otherwise. More than a ✓ is present in a line for composite mitigations.

Experimental Results The experiments were conducted on a dual core Linux virtual machine with all services running locally, communicating via HTTP on the loopback device. In these experiments, the test case triggers product exchange requests concurrently from two stores. Each of the two stores requests 10 exchanges, each exchange includes 10 different products. The store then waits 15 seconds and starts requesting again. The given test case is intended to permit an evaluation of the overhead introduced by Detectors and Mitigators and is not intended to reproduce a normal usage scenario.

The experiment results can be summarized as follows: the Detector and Mitigator approach, leveraging the Mule ESB framework, increases the reliability of web service composition without changing the component services. It allows encoding and enforcing SLAs that mandate timing requirements on service execution. Furthermore, the performance cost of using this approach is very limited.

Overhead Evaluation. The first two lines of Table 1 evaluate the cost of adding Detectors and Mitigators when there are no errors. This is an important measurement to validate our claim that the cost of using this approach is not too high. The results show an average increase of 7% in execution time. The network delay was minimized by using local communication. Therefore, the result evaluates the overhead of executing the detection over the normal processing of the service.

Failure Management Performance. The rest of the table shows various failures being triggered and how the presence of Detectors and Mitigators can mask the injected faults. The overall result is that by using the proper Detectors and a suitable set of the standard Mitigators, all of the injected failures could be masked.

SUMMARY

This chapter described how to leverage the Rich Service pattern to address an important crosscutting concern: failure management. The approach presented leverages the Rich Services blueprint, and the fact that it is based on a messaging infrastructure, to implement Detectors as RISs that monitor the interactions of a service by intercepting messages exchanged. This decouples the business logic from the fault tolerance components. Mitigators are also RISs that are activated by a Detector upon failure detection, and are responsible for recovering from the failure. Mitigators are strategy based and hence are decoupled from the failure detection logic. Using the case study I evaluated the performance of all recovery strategies commonly used in the Web Services domain.

From the measurements of the performance overhead and the reliability of the CoCoME trading system I conclude that, even in the case of multiple failures, by using a composition of different mitigation strategies the system can recover from all injected failures. This comes at the fairly modest price of 7% overhead in response time on average in the absence of failures.

In the next chapter I present a different type of experiment. Instead of testing the implementation of a rich service system to evaluate the performance of RIS in improving its reliability, I execute a formal verification of the system model to prove that properties are satisfied even when failures occur.

ACKNOWLEDGEMENT

This chapter, in part, is a reprint of material as appeared in V. Ermagan, I. H. Krüger, and M. Menarini, “A Fault Tolerance Approach for Enterprise Applications,” in Proceedings of the IEEE International Conference on Services Computing (SCC). Jul. 2008, The dissertation author was the primary investigator and author of the text used in this chapter.

CHAPTER 3

EXPLOITING CROSS-CUTTING CONCERNS IN VERIFYING FAULT TOLERANCE PROPERTIES

In this chapter I extend to embedded systems the study of fault tolerance discussed in Chapter 2 for enterprise systems. The case study I use in this chapter is the central locking system. Here I present a verification approach that follows the rich service process presented in Chapter 1. I create interaction models for the Central Locking System (CLS) case study and use them to generate model checking code. I use this code to formally verify properties of the CLS in presence of failures.

To leverage the proposed Rich Services pattern, I developed notations and theories that support the description and manipulation of services accordingly. To this end I created a dedicate Service Architecture Description Language (SADL). My goal is to apply the principle of separation of concerns. The RAS models are assumed to have no software bug or hardware failure. Detection of failures and recovery from them is a concern addressed by RIS.

THE CLS CASE STUDY

For the embedded systems domain I chose a case study from the automotive domain. I demonstrate my service-oriented model-based approach using the central locking system. The CLS is simple enough to be described here and has requirements that show the main challenges common to the automotive domain.

In modern high-end cars, CLSs are very complex and require the integration and cooperation of up to 20 electronic control units (ECUs). In fact, a complex set of comfort functions are connected to the locking and unlocking of the vehicle. Examples include the movement of the driver's seat, the setting of radio presets, and, in some cases, the automatic control of windows and moon roof. Other functions are related to vehicle security, such as engaging and disengaging alarm systems and engine electronic locks. Finally, safety and regulatory aspects of the CLS impose additional constraints, such as unlocking the doors in case of accidents or locking them after a certain speed is reached. Consequently, implementing a CLS for cars is a complex integration problem, and the resulting system must guarantee tight real time constraints and adherence to various regulations imposed by different bodies in different countries. Table 2 depicts a small subset of the CLS's functional and quality requirements.

Figure 7 shows a use case diagram derived from these requirements, capturing two actors: the driver and the system. A line connecting an actor to

Table 2. CLS Requirements (simplified).

Functional Requirements
Unlock the door when the key fob unlock button is pressed.
Lock the door when the key fob lock button is pressed.
Unlock all doors when an impact is detected.
Quality Requirements
The time between an impact and the door unlock must be less than 0.1 seconds.
The emergency door unlock must be guaranteed even in presence of ECUs failures.

a use case indicates participation in the use case execution. For instance, the driver takes part in the unlocking and locking operations, whereas the system takes part in crash management.

Both of the unlocking and locking use cases include a sub use case to make the car lights blink. The unlocking operation also includes an authentication use case, while the locking operation includes arming the security module in the car. The crash management use case includes a sub use case performing the immediate unlocking of all doors.

The next step is to identify the proper roles for the entities interacting in the services implied by use cases and requirements. Note that this step disentangles the logical model from deployment concerns: a role is the behavioral contribution a deployment component makes to a given service. In other words, a role is a behavioral proxy for a deployment component in a

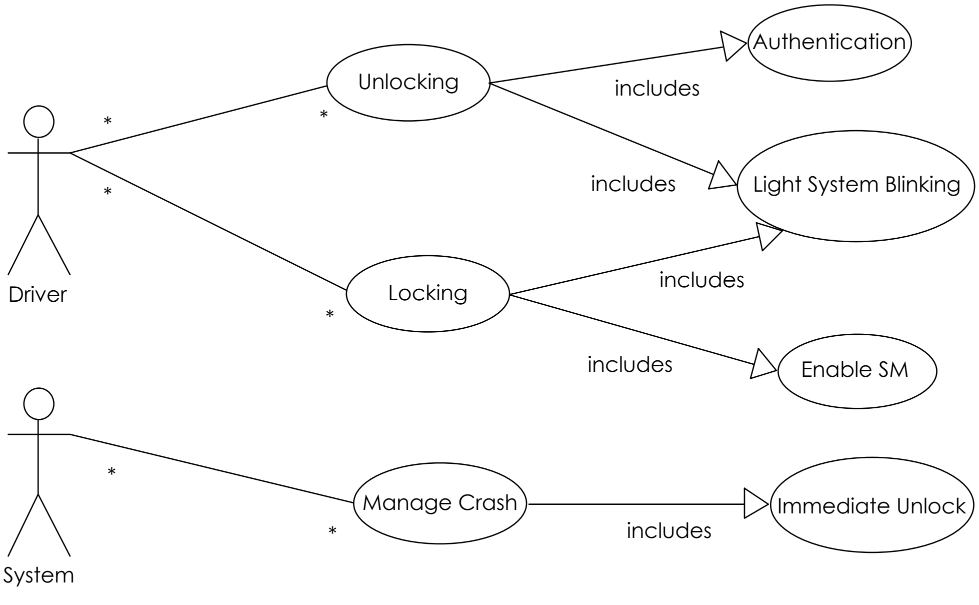

Figure 7. Use case diagram for the Central Locking System (CLS) case study

service specification; we also say that the deployment component “plays” that role. Roles such as the driver and the key fob are directly mentioned in the requirements and use cases. Other roles are inferred by knowledge of existing system implementations and are part of the design.

For example, a CLS controller manages the interactions that occur during the locking and unlocking, a security manager supports the execution of the authentication use case. For this case study, I identified seven roles: the car key fob (KF), a lock motor (LM), a security manager (SM), a database (DB) (which is usually played by a Controller Area Network (CAN) Bus or other in-vehicle bus), an impact sensor (IS), the light system (LS), and a lock controller (CONTROL).

Having identified the roles and use cases, I proceed with the definition of services. In the Rich Services framework there are two types of services: rich application services (RAS) and rich infrastructure services (RIS). RAS capture interaction patterns between roles to perform a function directly useful to the system user (a business function). On the other hand, infrastructure services change the interaction patterns to enrich or modify some functionality leveraging the service infrastructure (e.g., authentication and encryption of communication). These functions do not provide business value by themselves but need to be applied to other interactions (through an infrastructure or middleware). Another important example of RIS is a failure management service that identifies deviations of the expected behavior of other services and injects mitigation strategies to recover from them. This results in an end-to-end view of failure management across the subsystems in the vehicle.

MODELING THE RAS

I capture the interactions defining services using MSC. I model the basic scenario where the vehicle is initially locked. A key fob is used to remotely unlock or lock the doors. Unlocking and locking can iterate indefinitely. In parallel, the impact sensing operation is performed – when an impact occurs, the system unlocks all doors. We separate the locking and unlocking operations into two services: LCK1 (UNLK1) performs the (un)locking the doors and signaling the light system (LS), whereas LCK2 (UNLK2), performs the key fob authentication and arms the security manager (SM).

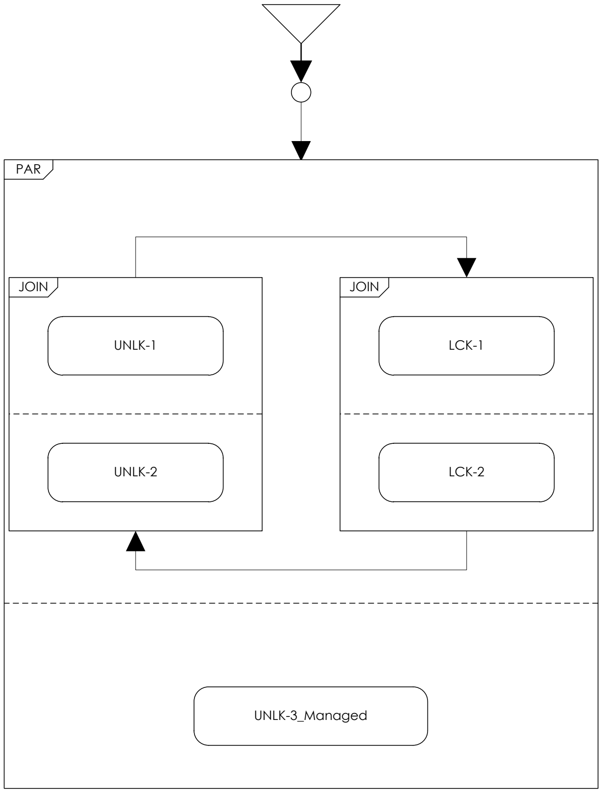

Figure 8. CLS-1

Figure 8 shows this model as a high level MSC (HMSC) capturing the global view of the CLS RAS of our case study. The two boxes labeled JOIN represent an operation that performs each of the interactions defined by the operands (in this case LCK1 and LCK2, and UNLK1 and UNLK2 respectively) in parallel but synchronizes them on common messages.

Figure 9 and Figure 10 specify the interactions UNLK1 and UNLK2, respectively. When the driver tries to unlock the doors, the lock controller notifies the security manager to authenticate the key fob. When the key fob ID is validated, then the unlocking operation is successful. Since the unlocking operation and the ID verification runs simultaneously, it is appropriate to join

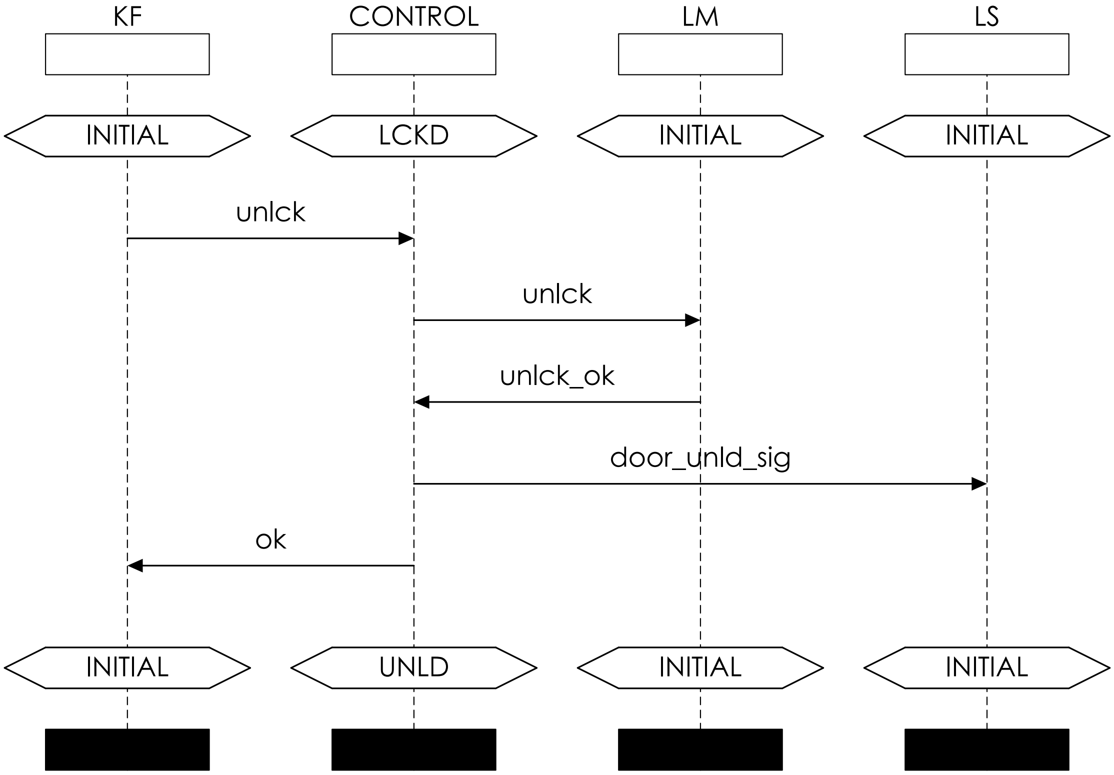

Figure 9. UNLK-1

UNLK1 and UNLK2 services to run simultaneously and synchronize through the same messages they share. This means that a single *unlck* message from KF to CONTROL starts both interactions. Also, a single *ok* message from CONTROL to KF indicates the completion of both interactions. The 2 UNLK MSCs are combined using the *join* operator. This operator ensures that both interactions are completed before the shared ok message is returned. This description technique specifies the scenario in which both the opening of the door and the authentication are successful. In this case we don't care in which order the messages are sent. Additional scenarios are needed to describe what happens in case of failure of the authentication. For example, those additional scenarios could enforce that if the authentication fails the door is never opened. The power of the notation used is that additional scenarios may impose constraints on message ordering (i.e. the security manager should confirm the *id* before the unlock request is sent to the lock manager).

In UNLK-1, CONTROL is initially in the locked (LCKD) state. After unlocking the doors, CONTROL also sends a signal to LS, to handle the blinking of the light system. After finishing the unlocking process, CONTROL reaches the unlocked (UNLD) state.

Figure 11 shows a scenario where a crash occurs and the related unlocking procedure is performed. Upon impact, an Impact Sensor (IS) sends an *Impact* message to CONTROL, which has the command center role of the CLS. CONTROL sends an *unlck* message to LM, upon receipt of the *Impact* message. The service ends by CONTROL receiving the acknowledgement of

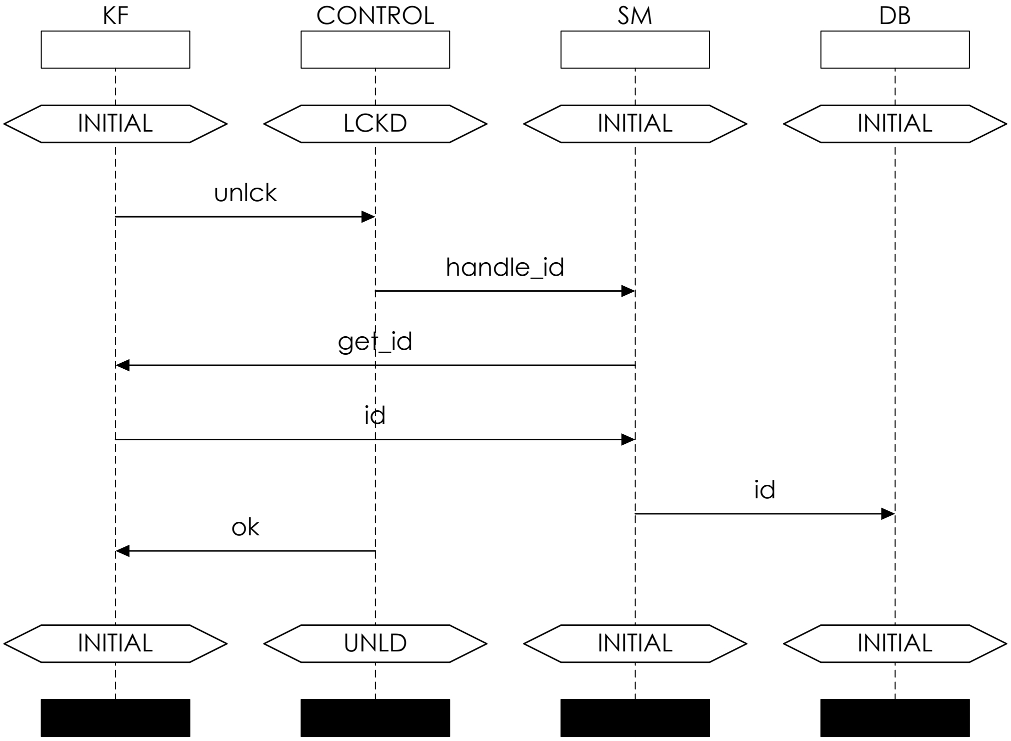

Figure 10. UNLK-2

the unlocking from LM. Out of the many possible failure scenarios, for this example, I chose a scenario based on a deadline. Hence, an additional constraint, identified by the dotted arrow in the diagram, is that the interval between the impact message and the acknowledgement of the door being unlocked must be less than 10 milliseconds.

MODELING THE RIS

The MSCs presented in the previous section define the required behavior of the application. From such models, an automaton accepting the modeled interactions can be derived for each modeled lifeline (service role). Such automata can be used at runtime to identify deviations from the expected behaviors. For brevity, this case study focuses on just one level of the hierarchy of the CLS model. Ref. [28] contains an algorithm to obtain state

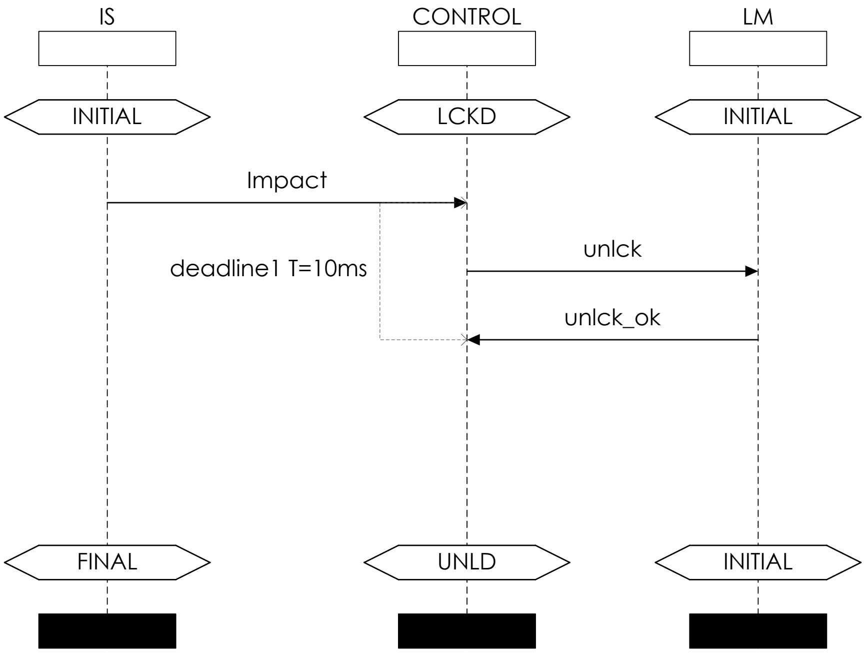

Figure 11. UNLK-3_Managed

machines from MSCs and HMSCs. As demonstrated in the previous chapter, leveraging the Rich Services framework, it is also possible to observe the interactions performed by RASs and implement such state machines in a RIS. Thus, leveraging RISs, it is possible to separate the management of failures in infrastructure services, and enhance the system logic not only to detect but also to recover from errors. For this purpose, a failure hypothesis is needed. The failure hypothesis identifies the type of errors (originating from software or hardware faults) and their combinations, in presence of which the system must still behave correctly.

The process of detecting failures is completely independent from the physical deployment of the system. In fact, errors are identified by the lack of adherence to the interaction pattern defined by a service. However, a failure hypothesis needs to identify dependencies between which component

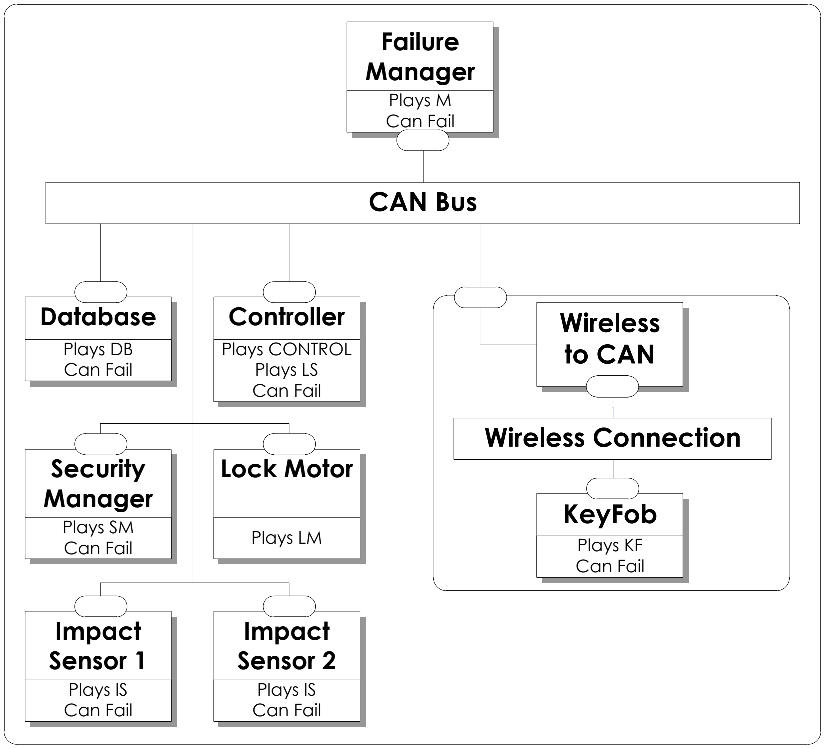

Figure 12. Deployment architecture

performs a given function and how such component can fail. In the CLS case study the assumption is that when an ECU fails, then all its software functions fail together.

DEPLOYMENT AND FAILURE HYPOTHESIS

The failure hypothesis proposed in this approach is closely related with the deployment architecture. In the deployment diagram from Figure 12, a CAN Bus connects the different subsystems: ImpactSensor1, ImpactSensor2, SecurityManager, DataBase, FailManager, LockMotor, and Controller. The KeyFob is connected to the rest of the system via a wireless connection and with an adapter to the CAN Bus. Each component can play one or multiple roles defined as lifelines in the MSCs. The Failure Manager component plays a special role (called M). This role performs the mitigation part of the managed service as depicted in Figure 13. The failure hypothesis is that all components except the lock motor can fail. Also, the wireless channel can fail completely, whereas the CAN Bus can only fail by losing one message per run. This failure hypothesis is completed by a global constraint on the number of concurrent failures allowed – in this case, just one entity per run can fail. I picked this scenario as it covers an important class of failures, i.e., when the number of failures has a finite upper bound. The SADL developed to support the creation of failsafe systems allows specification of all these concerns.

To support fail safety, a RIS, need to be able to specify both how to detect errors, and how to recover from them. Different domains use different techniques and follow different rules. Chapter 2 explored the domain of enterprise applications and proposed a method to make an enterprise system reliable by implementing RIS that support the standard fault tolerance techniques used in this domain. The creation of fail-safe systems in the automotive domain benefits from a domain specific language supporting techniques targeting this domain.

GENERATING THE VERIFICATION MODEL

The Service-Oriented Software & Systems Engineering Laboratory (S3EL) at UCSD has developed a modeling tool called M2Code [49] that is able to generate state machines from interaction descriptions. Additional tools were then added by me and other members of the S3EL team to M2Code. M2Code now supports automatically weaving failure detector / mitigator state machine templates into the generated state machines following the

Rich Services pattern [50]. M2Code takes as input a SADL specification and generates both implementation code and models to support formal verification.

The generation of such implementations is based on the synthesis algorithm presented in Ref. [45]. I briefly outline the main steps of this transformation here; and refer the reader to Ref. [45] for a complete treatment of the subject.

The input for this algorithm is a set of MSCs described in the SADL developed for M2Code. The algorithm uses a closed-world assumption with respect to the interaction sequences that occur in the system under consideration. For each role of the MSC set an automaton is obtained by successive application of the following four transformation steps:

1. projection of the given MSCs onto the role of interest
2. state marker insertion, i.e. adding missing start and end states before and after every interaction pattern
3. transformation into an automaton by identifying the MSCs as transition paths, and by adding intermediate states accordingly
4. optimization of the resulting automata

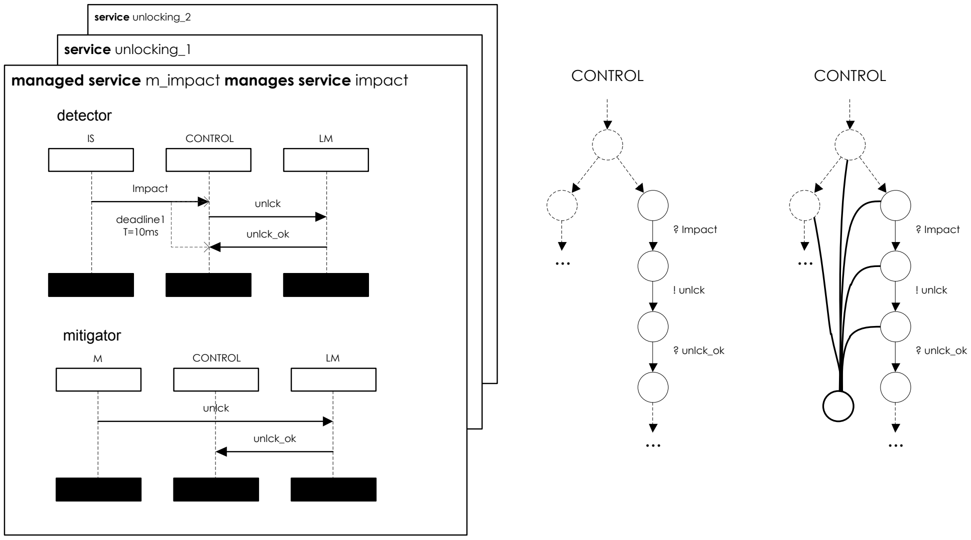

Figure 13. From MSCs to State Machines

This synthesis algorithm works fully automatically for causal MSCs [51], and correctly transforms choice, repetition, concurrency/interleaving and join [28] in MSCs. Because the algorithm is based on syntactic manipulation of the given MSCs it is oblivious to the underlying MSC semantics - as long as the semantics of the target component model matches the one used for the MSCs serving as input to the algorithm.

Figure 13 shows how lifelines in MSCs are converted to state machines. Each transition is marked with a message sent or received by the lifeline. To model failures, the state machine generated by the algorithm described earlier is modified by adding a sink state (lower part of Figure 13). Using this approach, I was able to verify that the CLS models were correct and ensure that the car gets unlocked during an accident even in presence of failures under the given failure hypothesis.

Figure 14 shows a fragment of the Promela code that the tool generates. The Promela code implements the state machines derived via the outlined transformation algorithm. The code from Figure 10 models an automaton for the LS role of our case study. The automaton is simple, there are only two states. State __JS0_26 is the regular state the automaton is in while the LS role is executing correctly. The _Sink state models a failure state in which the LS role can enter. The full Promela model contains one Proctype for each role in the CLS model and has, in addition, models for the mapping of roles to components, as well as the failure hypothesis. Using this Promela model generated by M2Code the SPIN model checker can be used to formally verify properties of the system.

```
Proctype LS (chan _CL3, kill_CL3, _ERR, kill_ERR)

 bool killedChan;
 mtype msg;
/* Set initial state */
 if :: goto end___JS0_26 fi;

/* State transition function */
 end___JS0_26:
  if
  ::d_step CL3?[door_lckd_sig] -> CL3?msg;
          goto end___JS0_26
  ::d_step CL3?[door_unld_sig] -> CL3?msg;
          goto end___JS0_26
  ::d_step _ERR?[kill] -> _ERR?msg;
          goto end_Sink
  fi;

 end_Sink:
 skip;
```

Figure 14. Promela code for the LS role.

SUMMARY

The case study presented in this chapter differs from the one presented in the previous one in two substantial ways. First of all, the approach in this chapter is based on a formal model of the system while the previous case study is based on an implementation. The goal here is to leverage simple models to generate a Promela program. This program can be used to formally verify properties of the system under all considered failure scenarios. Second, in this chapter I focus on an embedded system while in the previous one the case study was an enterprise system. However, I use the same detection and mitigation techniques in both case studies.

The two chapters complement each other. In fact, chapter 5 analyzes the execution performance of different failure detectors and mitigators implemented in an enterprise system, while this one presents a methodology to formally verify the impact of these detectors and mitigators without implementing them but just from their models. More details about the verification approach presented in this chapter and the tool chain supporting it are published in Ref. [50], [52].

ACKNOWLEDGEMENT

This chapter, in part, is a reprint of material as appeared in C. Farcas, E. Farcas, I. H. Krueger, and M. Menarini, "Addressing the Integration Challenge for Avionics and Automotive Systems - From Components to Rich Services," in The Proceedings of the IEEE Special Issue on Aerospace and Automotive Software, K. V. Prasad (Ed.), vol. 98, no. 4. IEEE, Apr. 2010, pp. 562-583. The

dissertation author was the primary investigator and author of the text used in this chapter.

© 2010 IEEE. Reprinted, with permission, from C. Farcas, E. Farcas, I. H. Krueger, and M. Menarini, Addressing the Integration Challenge for Avionics and Automotive Systems - From Components to Rich Services, The Proceedings of the IEEE Special Issue on Aerospace and Automotive Software, and 04/2010

CHAPTER 4

MODELING CROSSCUTTING CONCERNS

In the previous chapters I discussed how SOAs help to successfully integrate complex distributed systems in different domains. The Rich Service pattern simplifies integration further by decoupling crosscutting concerns from the flow of the business logic. I demonstrated the capabilities of the Rich Service pattern to address crosscutting concerns using fault tolerance as an example. In particular, I presented two case studies, one from the embedded systems domain and the other from the enterprise systems domain.

The examples presented in Chapter 2 and 3 cover one specific example of crosscutting concern: failure management. While the Rich Service pattern is not specifically developed for one type of concern, the SADL and the mitigation patterns discussed in the previous chapters are specific to failure management. To fully realize the promises of Rich Services a generic language to specify and compose crosscutting concerns must be devised.

The problem of decoupling the main flow of a program from the specification of crosscutting concerns has also been addressed in the programming world by aspect-oriented programming. To realize systems based on Rich Services I need to provide similar facilities for service models. Thus, in this chapter I analyze how the concept of aspect can be extended to modeling languages. To this end, I leverage the Rich Service architectural blueprint together with Aspect-Oriented Modeling (AOM) techniques [7–9]. In particular, I present work on aspects for MSCs, a simple yet powerful graphical notation to describe interactions that can be used to compose RIS to RAS.

The outline of this chapter is the following: first, I describe how the Rich Service pattern defines the interface between services and how this interface can be modeled. Second, I examine how current MSCs operators can be extended to allow an AOM approach to composition of RASs and RISs. Third, I discuss issues with respect to causality and address how those issues reduce the expressive power of the new operator if the weaving is done at run-time (as it can be done in ESB leveraging the router/interceptor layer). Finally, I discuss how the presented aspect technique is an important element of a Rich Service approach.

SERVICE/DATA CONNECTOR

In the Rich Service pattern each service is connected to the communication infrastructure via a Service/Data Connector (SDC). The SDC encapsulates the internal structure and behavior of a Rich Service and exports an interface that defines the communication patterns that the Rich Service can engage in with the external world. To define the structure and the behavior of such interfaces I can use MSCs. An SDC exports the internal roles (services) that the external world can see as well as the interaction patterns that the exposed roles are allowed to participate in.

Rich Services support two composition approaches. First, a RAS can act as an orchestrator, meaning that it utilizes other RASs to provide the composite service. Second, a RIS can intercept messages produced by RASs and route them as input to other RASs to produce the composite result. These two approaches support the Orchestration [53] and the Choreography [54] composition models respectively.

All interactions between a Rich Service layer and services on other layers happen via messages exchanged through the Service/Data Connector (SDC). The SDC represents both the structural and the behavioral interface for a service. I use MSCs to specify this interface. The structural elements are: the names on the life lines (Roles), the messages names, direction, and their parameter types. The behavioral element is the message sequence captured

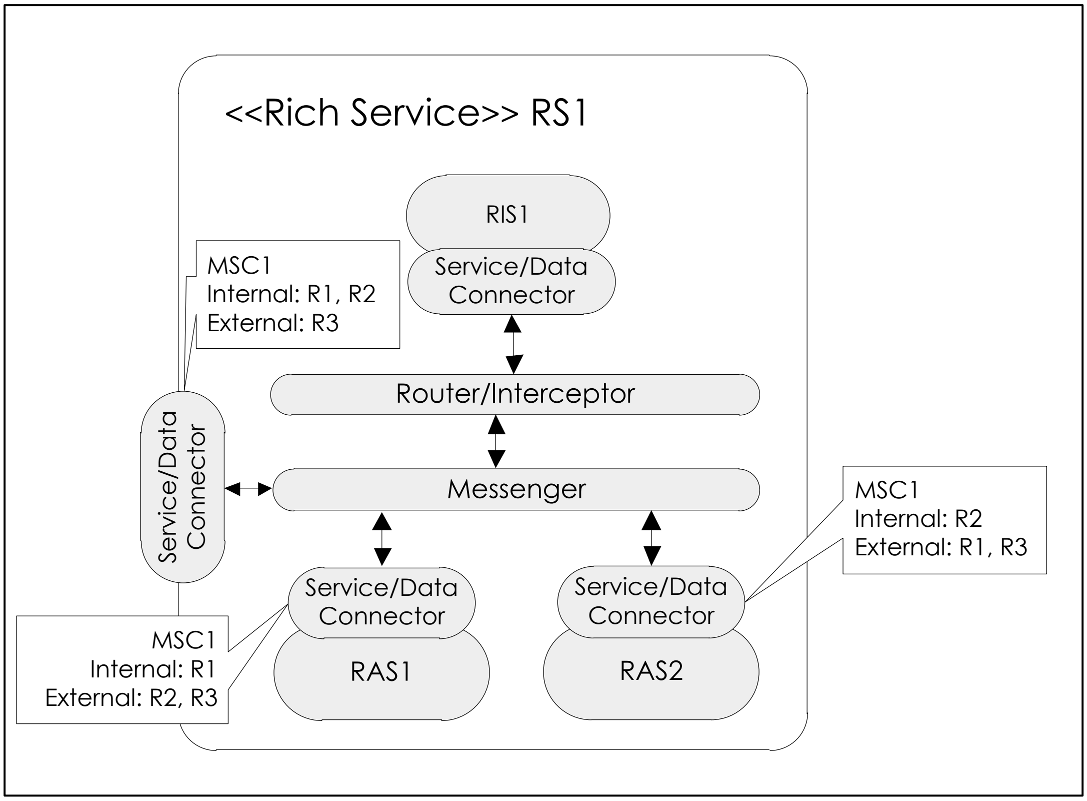

Figure 15. Rich Service with SDC specifications

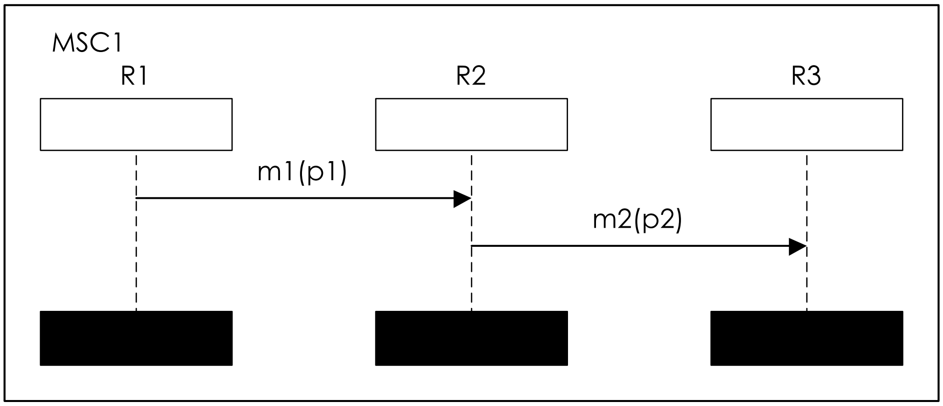

Figure 16. Message Sequence Chart referred in the SDC specifications

by the MSC. Therefore, I associate a set of MSCs with each SDC. To clearly define the interface I also specify which role is internal to the interface and which role is external.

Figure 15 shows a simple example of a Rich Service with the definition of the SDC for two internal RASs and for the SDC to the external world. Each SDC contains references to one or more MSCs that define the interactions allowed through that SDC (in this case the reference of MSC1 from Figure 16). Moreover, the SDC defines which role is internal to the SDC and which one is external. Internal roles are the roles in the interaction that are played by the Rich Service that owns the SDC, while external roles are played by some other RASs. Figure 15 shows that in MSC1, RAS1 plays the role of R1, RAS2 plays role R2, and role R3 is played by some Rich Service outside the scope of RS1.

The use of MSCs enables precise description of the communication behavior of each RAS. For such interaction specifications, under the constraint that MSCs are causal, the algorithm described in [45] can be leveraged to generate one state machine for each role. This state machine accepts the language defined by the complete set of message patterns that the role can engage in. As a result, a set of state machines defines the interface of each SDC, and therefore, the behavior at the interface of any RAS. The MSC of Figure 16 has been created with the M2Code tool [50]. As discussed in the previous chapter, the tool implements the state machine generation algorithm and support MSCs. However, the tool has not been extended to support the Aspect MSC notation and the Match operator introduced here.

The focus of this chapter is on how to model the SDC of RIS. The goal of RIS is to leverage the flexible Router/Interceptor layer of Rich Services to address the crosscutting concerns by modifying the message flow in the ESB. RIS provides a centralized place to define such crosscutting concerns. Because the system behavior is defined by modeling the interaction patterns between services, the message routing and filtering of a RIS should be specified by modeling how the communication patterns change. This goal is very similar to the goal of Aspect-Oriented Modeling (AOM) techniques [2]. To

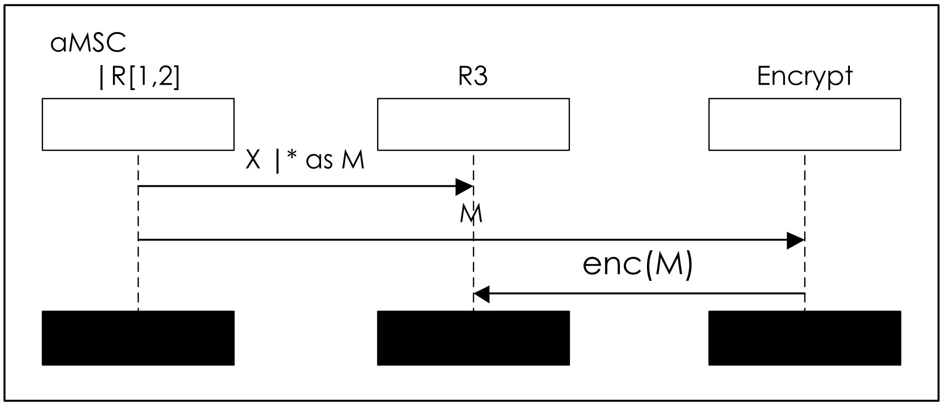

Figure 17. An example of an Aspect MSC

describe the routing capabilities of the RIS, the work presented here leverages the rich literature on AOM. In particular, the language proposed is inspired by Ref. [55].

I describe the SDC of a RIS by means of Aspect MSC. Aspect MSCs represent an extension of normal MSCs where the message name and role name of some elements is replaced by a special expression. I do not describe in detail the syntax of Aspect MSC here instead I introduce them via an example. In fact, the goal of this chapter is to explain how interaction models and aspects can capture in general the Rich Service composition. In the next chapter, I present in detail a more powerful textual language that capture interactions and aspects and can provide a formalization of Rich Services.

Figure 17 contains an example of an Aspect MSC. Roles that start with a "|" symbol represent template roles. The name is given as a regular expression that has to match a role name in the normal MSC. Template messages starts with "|" or with "X|". In the first case, they simply identify the message that matches the following regular expression. In the second case, they indicate that the matched message must be removed during the composition. A regular expression follows the "|" and optionally an "*as*" followed by an identifier name. The identifier is then bound to the matched message. The picture in Figure 17 for example, defines an Aspect MSC that matches whenever Role R1 or R2 send a message to Role R3. The given message is not sent to the role R3, but instead it is sent to the role Encrypt which forwards it encrypted to role R3.

The SDC of a RIS is, therefore, specified by an Aspect MSC. Optionally, it is possible to state in the RIS SDC to which MSCs in the Rich Service the RIS is applied. In case I do not specify any MSC, the RIS applies to all interactions of the Rich Service containing the RIS. To compose an Aspect MSC with normal MSCs in the next section I define an operator that recognizes the new syntax and uses it to weave the aspects.

MATCH, AN EXTENDED JOIN OPERATOR

Here I use the same semantic framework used to define MSCs as a basis for defining the semantics of the composition operator for Aspect MSCs. This is based on the notion of message streams [56] and predicates over such streams. A comprehensive description of the MSC semantics I am referring to can be found in [28].

Two core properties of aspect-oriented programming identified in literature are quantification and obliviousness [57]. Quantification identifies on what elements of the original program the modification is applied. Obliviousness requires that the application programmer does not need to be aware of the aspects that will be applied. The aspect composition operator can be analyzed according to those two properties. I believe, in order to help engineers better understand the resulting system during development and debugging, it is important to define the aspect composition at the level of

MSCs instead of defining it on the state machines translation. This provides the opportunity to view and debug all the resulting composed interactions at the MSC level.

The *Join* operator in the current MSC language is a good candidate to provide a basis for Aspect composition. *Join* is a parallel composition operator that synchronizes on common messages. Common messages, in this case, are defined as messages with the same name and where their sender and receiver roles' names also match. *Join* does not distinguish between messages that are specified to match and messages that are intended to add behavior. Everything that does match is synchronized and everything else adds messages in parallel. The *Join* operator, therefore, composes two MSCs and adds to the behavior specified in the second MSC to the one specified in the first. It quantifies on the common messages and the developer of each MSC does not need to be aware that it will be composed with another MSC. Thus, *Join* is suitable for aspects composition. There are some differences between what Aspect MSCs want to achieve and what *Join* does:

- An aspects specify which messages are in the model to execute the matching and which messages are there to add behavior. An aspect composition operator must understand that difference.
- The new behavior specified must be added only when there is a match.
- The matching strategy is more complicated (regular expressions instead of string equivalence).

- Messages can be specified by variables that are assigned by matching messages.
- Join matches once, while Aspect should be applied every time the given communication pattern is observed.

Given the similarity between the needs of Aspect MSCs for composition and the current *Join* operator I introduce a new operator *Match* for Aspect composition based on *Join*. Given an Aspect MSC ($AMSC$) and a MSC (MSC), I informally define the semantics of *Match* with the following two cases: 1) If there exists an instantiation, $I(AMSC)$, of all patterns defined in $AMSC$ such that all messages defined as patterns (starting with "|" or "X|" matches to messages in MSC according to the Join match rule. Then, the semantics of *Match* is the same of *Join* between the $I(AMSC)$ and MSC except for the following differences: the messages matching instantiated template messages marked for deletion (starting with "X|") are removed from the resulting MSC. Messages defined by an identifier in the $AMSC$ are replaced with the message matching the definition. While *Join* matched only once, the *Match* with $I(AMSC)$ is repeated for all occurrences of the template in the original MSC. Therefore, once the $I(AMSC)$ has been applied for the first time, the operator looks for more occurrences of the template captured by $AMSC$, possibly with a different instantiation $I_1(AMSC)$, and applies the aspect again. 2) If no instantiation $I(AMSC)$ exists, the result of the composition is the original MSC.

Figure 18 shows an example of the *Match* operator. It is the composition of `MSC1` from Figure 16 and `aMSC` from Figure 17. The only template message in `aMSC` is the first one: “X|* as M”. The pattern specifies every message from the source role to the destination role. Because the source role is also a pattern, “|R[1,2]”, matching messages are all messages from *R1* to *R3* and from *R2* to *R3*. The only message in MSC1 that matches the pattern is “m2(p2)” from *R2* to *R3*. The composition in Figure 18, therefore, has the matching message removed. Moreover, the composed MSC has the additional role *Encrypt*, the message “m2(p2)” from *R2* to *Encrypt*, and the encrypted message from *Encrypt* to *R3*. As shown in the figure, the identifier “M” has been replaced with the matching message, “m2(p2)”.

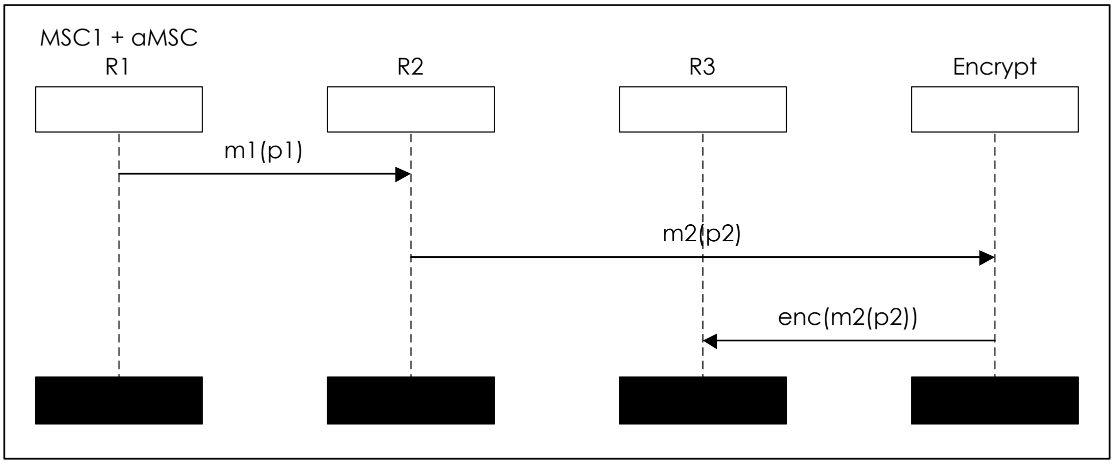

Figure 18. Result of composing MSC1 and aMSC using the Match operator

While this graphical notation is useful for specifying crosscutting concerns and for understanding how to model RISs in general, to formally define *Match* and analyze its properties a formal language defining aspects and their composition is needed. In the later chapters I provide such a language.

RIS AND CAUSALITY

In the MSC dialect I used in this chapter MSCs must be causal. A causal MSC restricts the sequence of messages exchanged such that when a role receive a message it can always locally decide the next step [51]. Causal MSCs, therefore, ensure that the global communication patterns specified by MSCs is the composition of the local communication patterns accepted by the state machines generated for each role by our tools. To have a valid composition, the result of *Match* must be causal. In general, there is no limitation for the set of template messages (starting with "|" or "X|") in an Aspect MSC to be causal, as long as the resulting composition is causal.

Middleware such as ESBs enable an application to intercept messages before they are received or sent by a service and perform arbitrary processing of the messages. For example, a message can be discarded, routed to another destination, or modified before being forwarded. It would be useful to be able to leverage these capabilities to plug RISs in the system at run time. To this end an implementation can insert a message interceptor at the interface of each RAS. This interceptor can use a state machine obtained from an Aspect MSC to observe the communication during run-time and decide if the Aspect MSC is applicable. The definitions of Aspect MSC and Match given in the previous section, however, is not suitable for run-time weaving using the Router / Interceptor layers implemented by ESBs. In fact, this run time weaving is possible only if the pattern that has to be matched is causal. In this case, the monitor can observe messages locally exchanged by RASs. If the template messages that are part of the Aspect MSC have to be identified at run time by a local monitor, the monitor has to choose locally whether the observed communication matches the aspect or not. This implies the local choice that is guaranteed only if the pattern is represented by a causal MSC.

Another restriction to enable the weaving of the aspect at runtime is that a causal dependence exists between detecting that the aspect applies and modifying the communication pattern. In fact, if the weaving is applied by statically analyzing the model, aspects can advise the MSC by inserting behavior before the interaction that identify the match is executed. As an example, consider the Aspect MSC of Figure 17. As identified while discussing Figure 18, there is a match in `MSC1` and it is on the message "m2(p2)" from *R2* to *R3*. In this example, the changes to the communication pattern of `MSC1` follow the detection of the match. However, I could easily add other messages to `aMSC` before the template message. The composition operator, once it has identified the match, can insert the additional messages in the composed MSC before the matching message. However, in case of run time weaving using the message routing and interception facilities of an ESB, there would be a paradox. The aspect would need to send messages before receiving the messages which allows it to identify the need to send them. The following restrictions enable support for run time weaving of Aspect MSCs:

- Normal messages are allowed only after the last template message. Therefore, the run-time monitor can determine that the aspect applies before having to modify the behavior.
- Only one template removing a message is allowed, and it needs to be after all the other template messages. This restriction could be relaxed to let *n* messages to be removed (as long as they are at the end of the sequence). However, this would introduce arbitrary delays in message forwarding.
- Finally, the set of template messages in an Aspect MSC, once instantiated, must form a causal MSC.

With the cited restrictions RISs that can be weaved in an ESB-based application at run time. Therefore, it is possible to keep the crosscutting concerns addressed by RIS separated from the business logic throughout the development phases, from design and modeling phase to deployment. This restricted version of the Aspect MSC language is, thus, suitable for capturing the routing information that the ESB framework uses to transparently modify the execution of deployed services.

SUMMARY

The Aspect MSC language presented in this chapter is a good candidate for providing a general purpose language to model systems according to the Rich Service pattern. Services can be composed describing their interaction patterns. Moreover, crosscutting concerns implemented by RIS can be injected using aspect composition. This approach, therefore, provide an improvement over the techniques described for failure-management in the previous chapters.

An aspect-oriented technique generalizes the failure management service approaches presented in Chapters 2 and 3 if it has the following characteristics. First it has to deal with time in a way that can be used to define deadlines and, in general, timing properties. Second, it has to be grounded in a formal model, which can be leveraged to synthetize code and verification models.

Next chapter introduces an orchestration language that fulfills the properties mentioned above. This language, called Orca, supports the orchestration of services, including sequential and parallel composition, has facilities for synchronizing parallel services, and supports composition of aspects.

ACKNOWLEDGEMENT

This chapter, in part, is a reprint of material as appeared V. Ermagan, I. H. Krüger, and M. Menarini, "Aspect Oriented Modeling Approach to Define Routing in Enterprise Service Bus Architectures," in MiSE '08: Proceedings of the 2008 international workshop on Models in software engineering, Leipzig, Germany. New York, NY, USA: ACM, May 2008, pp. 15-20. The dissertation author was the primary investigator and author of the text used in this chapter.

This work is based on an earlier work: Aspect Oriented Modeling Approach to Define Routing in Enterprise Service Bus Architectures, in

Proceedings of the 2008 international workshop on Models in software engineering (MiSE '08), © ACM, 2008. http://doi.acm.org/10.1145/1370731.1370735

CHAPTER 5

ORCA: A LANGUAGE FOR MODELING CROSSCUTTING CONCERNS

In the previous chapters I discussed different case studies involving integration of complex software systems. In particular, I identified as a key issue the composition of crosscutting concerns and I introduced model-based techniques to address such composition.

In this chapter I propose a simple algebra to capture interactions and address the composition of crosscutting concerns: I call it Orca. Orca extends the Orc language proposed by Misra and Cook [5]. The extension addresses composition of crosscutting concerns. The need for this work arises from the observation that providing a usable service ADL requires the use of graphical modeling languages and the introduction of multiple operators to cope with the different needs of different application. Providing a direct formalization of such language can be challenging. In contrast Orc is very concise (it has only four operators) and the extension I propose here adds just one composition operator for aspects. This simplified the semantic model for the language.

As demonstrated in the previous chapters, Interaction models can be leveraged for different purpose in the development process and used to create various types of artifacts. For example, in this thesis, I used these models for code generation: I generated both implementation of interaction monitors and model checking code to verify formal properties. In other work [58] I leveraged interaction models for instrumenting applications and perform run time verification of various properties. Orca can capture all interaction patterns I used in my case studies. Furthermore, it is also possible to map it to graphical notations. Therefore, it is a good candidate for the creation of an ADL that can address interaction models in different domains.

INTRODUCTION TO ORC

The power of Orc lies in its parsimony and its clearly defined semantics. For this reason, Orc represents an attractive platform to extend with an aspect operator. The result of adding aspect composition to Orc is Orca.

The complete Orc language consists of two parts: a workflow language (formed by the Orc workflow operators) and the Cor functional language. The complete language name is Orc/Cor. This combination allows programmers to create complete workflow-oriented applications. Consistent with [59], for the purposes of defining Orca, I consider only the workflow part of the language and refer to it as Orc. The extension of Orca to Orc/Cor (that is a possible solution for creating complete RIS definitions) is future research.

Figure 19 presents a brief definition of the Orc syntax. Ref. [60] presents a complete definition of Orc; including a formal operational and denotational semantics. Orc programs are made of a goal expression and a set of

$$
\begin{array}{rcl}
f, g, q, h & \in & Expressions \\
e \in Expressions & ::= & S(\bar{p}) | E(\bar{p}) | f > x > g | f \mid g | f < x < g | f ; g \\
p \in Actuals & ::= & x \mid m \\
S(\bar{p}) \in Sites & ::= & M(\bar{p}) \mid X(\bar{p}) \\
E(\bar{p}) \in Definitions & ::= & E(\bar{x}) \triangleq e
\end{array}
$$

Figure 19. Orca abstract syntax BNF

expression definitions. Orc expressions define the operations the Orc program performs. The execution of an Orc program corresponds to the evaluation of the goal expression and of all the other expressions referenced by the goal.

The basic building blocks of Orc expressions are sites and operators. Sites are primitive operations that accept a list of values as parameters, execute some computation, and can return a result. In Orc, a site call can return at most one value. The return of a call is a publication. A site can publish either a tuple value (including a simple scalar) or a *signal*, which represent a publication without content. An extension of the Orc language introduces a new return type (*halt*) and operator (otherwise). With this extension, a site can return a value, a signal, or halt. When a site halts, it reports that it will not return a value. Operators define dependencies between different site calls.

The semantics of Orc defines a set of events and the dependencies between them. Events are site calls, site publications and binding of values to variables. I have already described sites. The site invocation is represented by a site call event. The event representing a site terminating its processing and returning a value is the site publication event. Once a value is published, or if it is defined directly in an Orc expression, it can be bound to a variable and used in the remaining of the Orc expression evaluation. This is represented by a binding event.

A key aspect of Orc is that its operators establish dependencies between events. For example the >> operator in the expression $f >> g$ establishes a relation between the publication event of f and the call event of g. Figure 20 depicts this dependency. The return of the call to site f (publish event) depends on the event representing the call to f. The event call to g depends on the publication of site f because of the Orc operator >>.

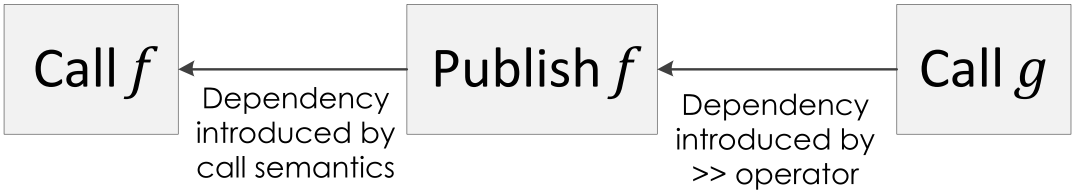
Call f
Publish f
Call g
Dependency introduced by call semantics
Dependency introduced by >> operator

Figure 20. Dependency relations introduced by Orc operators

A call to a site can execute only if all call parameters have value. If an Orc expression refers to a site with a parameter list containing variables, Orc delays the execution of the call until all variables are bound to values. The binding of variables to values can happen in two ways: either a value is assigned to a parameter in an expression call, or the variable is bound to the value of a site publication. An Orc expression can call a site in two ways: it can explicitly name the site (i.e., given a site named M: $M(\quad)$), or it can use a variable that gets bound to the site name in the execution the expression (i.e., given the variable named X: $X(\quad)$).

Orc provides a number of predefined sites, including 0, if, let, and Rtimer. 0 never publishes anything, and can be used to terminate the execution of an expression. $\mathrm{if}(x)$ publishes a signal if x is true, and publishes nothing if x is false. $\mathrm{let}(\mathrm{p}_1, \ldots, \mathrm{p}_\mathrm{n})$ publishes a tuple whose members are the values of call parameters ($\langle \mathrm{p}_1, \ldots, \mathrm{p}_\mathrm{n} \rangle$). $\mathrm{Rtimer}(x)$ publishes a *signal* after x milliseconds. Other sites, including user-defined sites, may perform any operation on the parameters: including transforming parameters, retrieving or persisting results, interfacing with other systems, and performing calculations.

The Orc BNF syntax in Figure 19 has 4 sets. The set $Expressions$ contains all proper Orca expressions, the set $Actuals$ contains the actual parameters of calls, the set $Sites$ contains all valid sites, and the set $Definitions$ contains the definitions of named Orc expressions. Actual parameters are the entities used in sites and expression calls. They can be either variables (x) or values (m). Site calls have two forms. They use the site name ($M(\bar{p})$) or a variable that is assigned to the site name ($X(\bar{p})$). The parameters ($\bar{p}$) used in site and expression calls are tuples of $Actuals$. The syntax of Orc also specifies expression definitions. A definition has a name and a set of formal parameters on the left of the ≜ symbol, and an expression on the right. Expressions are site calls, calls to other expressions (named in a definition), or combinations of other expressions joined by composition operators.

Orc defines four composition operators. $f > x > g$ is serial composition. Values published by expression f are assigned to the variable x, and the value assigned to x is available in expression g. A separate copy of g is executed for each value published by f. If f does not publish a result, x is assigned no value, and g is not executed. Serial composition is right-associative. Note that $f \gg g$ is a convenience notation where f publishes a value which is not used in the rest of the expression. Notice, however, that a new instance of g is executed for each publication of f.

$f \mid g$ is symmetric parallel composition. Expression f is executed in parallel with expression g. If both f and g contain only site calls, the parallel composition can publish 0, 1, or 2 results; otherwise, it can publish a stream of results in the time order they are published by f and g. If f and g publish no results, the parallel composition publishes no results, too. Parallel composition is fully associative.

$g < x < f$ is asymmetric parallel composition. Expression f is executed in parallel with g. When f publishes a value, it is assigned to x, and f stops its execution. If any site call in g depends on the value of x, when one of such calls is reached, the call is delayed until x has a value. If x is never assigned a value (because f never publishes one), the calls in g that depends on x are never executed.

$f; g$ is otherwise composition. The new operator otherwise (;), introduced together with the *halt* return value, runs an alternative expression g if the primary expression f halts without returning any value. An expression halts when all sites called halted and no more sites that can be called (i.e., they depend of variables not assigned to values).

Below I give some simple examples inspired by the copious examples in [5]:

Example 1

$$CNN > x > email(x,\text{"news@news.com"})$$

First calls a CNN news feed site to get a single news story, then passes the news returned to an email site that sends it to a particular mailbox. If no story is available, none is e-mailed.

Example 2

$$CNN \mid BBC > x > email(x,\text{"news@news.com"})$$

This expression calls both a CNN and BBC news feed site to get a single story from each, then passes each story (if any are available) to the email site.

Example 3

$$\big((let(x,y) < x < CNN) < y < BBC\big) > (c,b) > \\ \big(email(c,\text{"news@news.com"}) \mid email(b,\text{"news@news.com"})\big)$$

This is a slightly more complex example. This expression calls a CNN news feed site and publishes the story to variable x. Simultaneously, calls a BBC news feed site and publishes the story to a variable y. The let site waits for both x and y to have values before publishing a tuple containing both stories. The tuple is bound to (c,b) and passed to separate copies of the $email$ site, one mails the CNN story (c), and another mails the BBC story (b). This example demonstrates three composition operators and binding of variables. The compound asymmetric parallel composition is an example of joining two separately executing expressions; a convenience notation for a join would be $\text{let}(CNN, BBC)$.

Example 4

$$let(x) < x < \left(CNN \mid \left(Rtimer(1000) \gg let(\text{"None"})\right)\right)$$

This example introduces the use of a timeout. The expression calls a CNN news feed site and publishes a story if one is available within one second; otherwise “None” is published.

Example 5

$$\begin{aligned}&News(n) \triangleq \left(let(x) < x < \left(n \mid \left(Rtimer(1000) \gg let(\text{"}None\text{"})\right)\right)\right)\\ &\qquad > y > email(y,\text{"news@news.com"}) \gg News(n)\\ &News(CNN)\end{aligned}$$

This is a more complex expression that uses timeouts. The expression defines a $News$ function that fetches a story from a news feed, timing out after one second. Either way, it sends the result as an e-mail, and repeats the process. To call the $News$ function we use a news site name as parameter (i.e., CNN).

Example 6

$$\left(if(flag) \gg News(CNN)\right) \mid \left(if(\neg flag) \gg News(BBC)\right)$$

This expression creates a parallel execution consisting of two expressions. If the flag value is true, the first expression starts a CNN news feed;

otherwise, it does nothing. If the flag is false, the second expression starts a BBC news feed; otherwise, it does nothing.

ORCHESTRATION WITH ASPECTS

Orc is a powerful and elegant language to describe interactions; however, it is missing an important piece needed to define Rich Services: addressing crosscutting concerns. To describe a RIS in Orc an expression describing the orchestration of RAS must be changed to also contain the services required by the RIS. Orca expands the capability of Orc to address the injection of crosscutting concerns in exiting expressions without modifying them.

My goal in creating Orca has been to address the injection of additional computation (or the removal of it) in the middle of an expression. This enables users to keep their crosscutting concerns specified in a modular way as required by the rich service pattern. Orca can be applied beyond modeling rich services. In fact, modularizing computation that cross-cut a workflow has application in other fields such as workflow-evolution and software policy management just to name two.

INTRODUCTION TO ORCA

Orca introduces two new elements to the Orc language. The first element is the concept of expression interface. The second element is the aspect composition operator.

An expression interface extends the expression definition feature of Orc by decoupling the expression implementation from its definition. Expression interfaces are a key element for aspect composition. In fact, aspect-oriented techniques generally can be used to modify the behavior of an application only on a given set of points in the base code (called joinpoint). Each aspect provides an expression that selects a subset of such joinpoints where the aspect must be applied (called pointcut). The role of an interface for aspects it is then to limit how a given expression can be modified by aspects.

An aspect weaver uses interfaces to identify what joinpoints in the pointcut can be advised with the given aspect. This is a key requirement for model based development. In fact, during the development of complex system models inconsistencies arise and a proper modeling language must help the developer in identifying and resolving such inconsistencies. I discuss the consistency issues in model based development in the next chapter.

The aspect composition operator must identify the set of joinpoint which will be modified in the original expression. To this end the aspect composition operator in Orca has two parts: a pointcut and advice. The poincut is an expression that identifies joinpoints. It must identify the expressions it applies to and which elements in the expression. The advice is an Orca expression that can defines interaction patterns between the joinpoints in the pointcut and other sites.

In my approach I chose to use variables in expression definitions as joinpoints. In fact, Orc variables establish communication between the output of and expression and the input of other expressions. They are the perfect point to inject a modification of the interaction pattern.

Let's consider the Orc expression $Site1 > x > Site2(x) > y > Site3(y)$. Figure 21 shows a representation of how variables are used in the expression. When $Site1$ publishes a value, $Site2$ is called (spawning a new instance of $Site2$ for each published value). The input of $Site2$ call is the value published by $Site1$. The same thing happens for values published by $Site2$ and consumed by $Site3$. The figure depicts that the variables can be considered identifying a communication channel between two calls. The difference between a variable and a traditional communication channel is that there are as many different instances $Site2$ and $Site3$ for as many values are published by the preceding sites.

A slightly more complex example expression is the following: $SiteA > j > SiteB(j) > k > SiteC(j, k)$. Figure 22 depicts the communication pattern of this expression. In this case $SiteC$ receives the input of two variables. The control structure is the same of the previous example: a new instance of $SiteB$ is run for each value published by $SiteA$ and similarly a new $SiteC$ is intantiated for

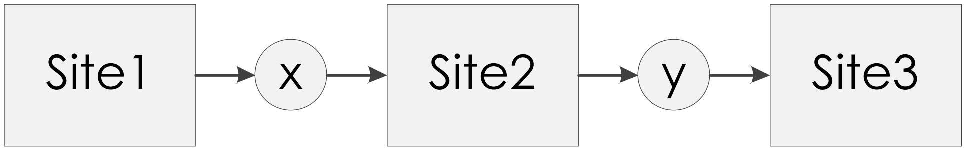

Figure 21. Example of variable used in passing messages

each publication of $SiteB$. The communication between the variable j and $SiteC$ is now different. In fact, each value published by $SiteA$ must be passed not only to one $SiteB$ call, but also to all calls executed by a given instance of $SiteB$.

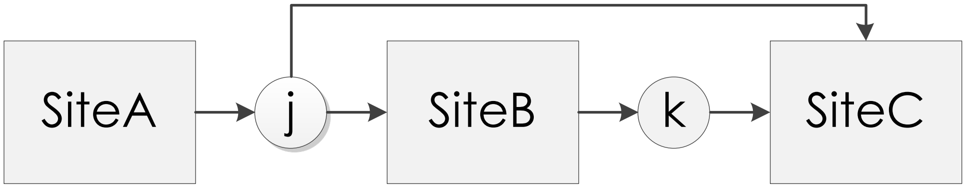

Figure 22. Example of variables used by multiple calls

From the examples in the previous paragraphs, it can be can inferred that variables represent multiple communication channels even an infinite number of them. Each such channel transfer a single value output by a process (publishing site) to the input channel of an unknown number of instances (all sites called by the expression that use the variable as parameters in the call).

In Orca I interpret each variable as a representative for a class of publish/subscribe channels. For each of such channels only one site or expression publishes one value to the channel. All sites and expressions using the variable in their call are subscribers to the channel. Once a value I published to the variable, the variable sends this value to all subscribers.

With this publish/subscribe view each variable can be split in an input part that receives the value published to the variable, and an output part that publish the value to all subscribers. Orca joinpoint is defined between the input and output part of each variable. Therefore, it is possible to intercept all messages being published to a variable and add additional behavior to them. It is also possible to forward messages to the output part of the variable and tap into the interaction following the publication of such variable.

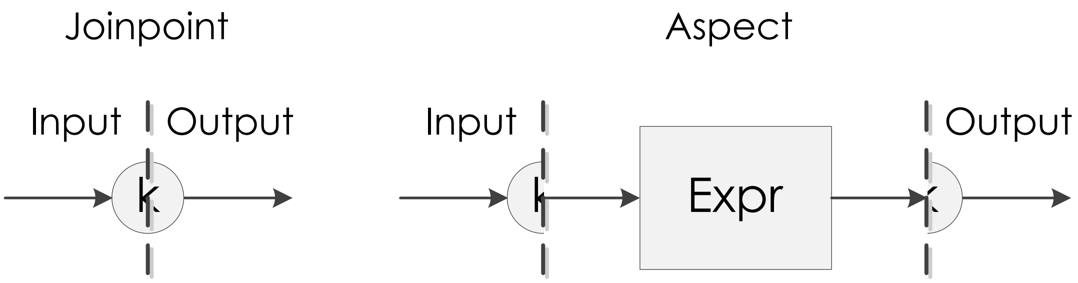

Figure 23. Joinpoints and aspects in Orca

Figure 23 depicts how a variable can act as a joinpoint in Orca. Each variable has an input part, where the value published by some site is published, and an output part, where the value is pushed out of the variable to all its subscribers. An aspect is able to weave an arbitrary expression in between the input and the output of a variable. Thus, the expression can modify the input value before passing it out. However, being an arbitrary expression, an aspect can also ignore the input value or send values to the output even if no input is received. This makes the Orca definition of aspect very powerful.

EXPRESSION INTERFACES

The concept of interface has been introduced, and its usefulness proved, in many modern programming languages. Interfaces are a key concept to modularize code and foster reuse. Because one goal of Orca is to simplify the composition of Orc expressions the first contribution of Orca is an expression interface. An expression interface has two parts. The first one is the interface return signature and the second part is the call signature. By introducing interfaces, Orca forces developers to explicitly define the data types that are exchanged. In an Orc expression definition types are implicitly defined by the sites that use the information or return it. To abstract from the implementation details, interfaces do not depend on sites. Therefore, the information on input and output types must be explicitly provided.

The second line of Figure 24 shows the definition of interfaces in Orca. An interface has a return signature (represented by $(Q\!:t)$ in the figure), a call signature ($I(\overline{x\!:t})$ in the figure), and a pointcut signature ($[\overline{v\!:t}]$ in the figure).

The return signature contains a quantifier Q and a type definition t. Q is a Boolean expression that defines the acceptable values of the natural number $q \in \mathbb{N}$ of values published by the expression interface. Moreover, t represents the type of values published by the interface.

The call signature is similar to a signature in an expression definition. It contains the interface name followed by a comma separated list of parameter names (variables x). The main difference is that for each variable

$$\begin{array}{rcl} f, g, q, h & \in & Expressions \\ I \in Interfaces & ::= & (Q\!:t)I(\overline{x\!:t})[\overline{v\!:t}] \\ E(\bar{p}) \in Definitions & ::= & E\!:I(\bar{x})[\bar{v}] \triangleq e \\ A \in Aspects & ::= & E(\bar{p})[\bar{m}] | @m | @m(p) \end{array}$$

Figure 24. Orca Extensions to Orc, abstract syntax BNF

after the column we have a type definition t.

Finally the pointcut signature is optional and contains a comma separated list of tuples. Each of such tuple contains a variable name and a type. The variable name is used to specify which variable used by an expression implementing the interface can be advised. The type element specifies the type that can be passed to the variable.

I define Orca types using Java types. Therefore, valid interfaces are for example: $(q = 1: Object)IMap(key: String, value: Object)$, which defines a IMap interface which accepts a variable key of type String and a variable value of type Object and publishes exactly one value of type Object, and $(q = \infty: null)IMetronome()$, which defines an IMetronome interface with no parameters and publishes an infinite number of signals (null maps nicely to a value for an Orc signal).

ASPECT COMPOSITION OPERATOR

The aspect composition operator extends Orc by supporting injecting behavior in an existing Orc program. Most of the aspect oriented languages support this by breaking the interfaces of the underlying languages. A set of join points are selected by means of a pointcut definition and they are modified by means of advices. However, the programmer of the base code does not have much control on what join point can be selected and how they can be modified. Orca takes a different approach. By leveraging the expression interface defined in the previous section, Orca clearly defines the input and output requirements for each join point. An Orca aspect can then advise the expression implementing an interface but it is constrained to respect the input and output requirements of each join point in the pointcut. This gives the programmer of the base code the opportunity to clearly define where the program can be modified and what it expects to send and receive at any given point.

Line 4 of Figure 24 presents the syntax used for aspects in Orca. An aspect is a normal Orc expression which calls expressions that have a pointcut signature defined for their interfaces. The main difference from a normal call is that for each tuple in the pointcut signature a name is required. The pointcut matches each name to the corresponding variable inside the called expression. Using these names with the @ opertator it is then possible to receive or send values from the pointcut variables. The syntax for receiving or sending values is similar to a site call in Orc. $@m$, where m is the name used in the pointcut, publishes the values received by the variable. $@m(p)$, on the other hand, does not publish any value but send each value passed to the parameter p in the site call to the output of the pointcut variable.

Example 7

$$News\colon NewsFeed(n)[y] \triangleq \left(let(x) < x < \left(n \mid \left(Rtimer(1000) \gg let(\text{"}None\text{"})\right)\right)\right)$$
$$> y > email(y,\text{"news@news.com"}) \gg News(n)$$
$$(q = 0\colon null)NewsFeed(feed\colon Site)[message\colon String]$$
$$FilterNews(n) \triangleq News(n)[m] \mid @m > msg > Filter(msg) > msg1 > @m(msg1)$$
$$FilterNews(CNN)$$

This example is an evolution of Example 5. It demonstrates the use of aspects in Orca. In this example I changed the definition of the $News$ expression to include the $NewsFeed$ interface. $NewsFeed$ has no return value (meaning that once executed it calls itself forever without returning). It also defines one $Site$ parameter which represents the news feed site called inside the $News$ expression. Finally, the interface defines a pointcut specification. The $message$ parameter in the interface pointcut is of type $String$. This means that all messages received from the pointcut variable or sent to it must be strings.

The expression $FilterNews(n)$ applies an aspect to $News$. This example show how nicely the aspect syntax in Orca merges with the classic Orc language. $@m$ is a site that publishes the values published to the message variable inside the base code (y in $News$). $@m(msg1)$ is a site that publishes the value of $msg1$ to the message variable in the base code (y in $News$). Using aspects enables the expression $FilterNews$ to modify the message that is emailed inside $News$.

A GRAPHICAL MODEL OF ORCA EXPRESSIONS

In this section I present graphical models of Orca expressions. These models are generated by a tool that supports deigning Orca expressions in graphical form. The tool is also able to transform the model and render it in textual form as an Orca expression.

In its graphical form, an Orca expression definition is represented by a box; the name of the expression in the top left corner of the box while the parameters are represented by a list of smaller boxes under the expression name. Figure 25 is an example of an expression definition. In this example, the expression name is Exp1 and it has only one parameter called P1.

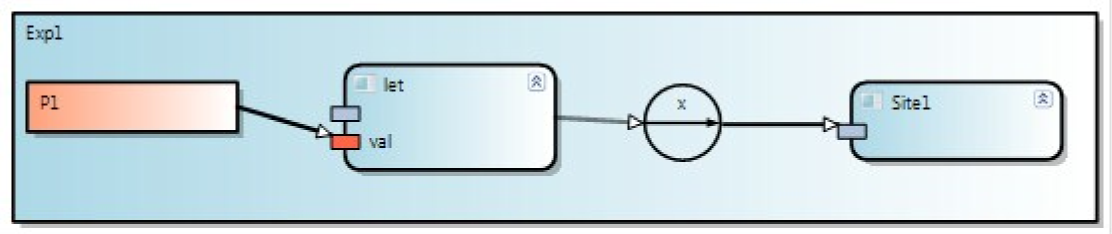

Figure 25. Example of Orc >> in graphical form

Each call in the graphical representation of an expression definition is a rounded box; on the top left part of the box there is the name of the called site or expression. Each call parameter is represented by a small rectangle on the box border. The first of such rectangles (blue) is used to represent execution causality (used in expressions such as << and >>), the other rectangles (red) have the parameter name written next to them.

Variables are represented by a circle. The variable name is written inside the circle. In Orc variables are used with two operators << and >>. The type of operator used is represented in the graphical language as an arrow inside the variable circle. If the arrow is solid, it means that the operator used in Orc is >>, otherwise the operator is <<.

Figure 25 shows an example of using the >> operator in the graphical language. The equivalent Orc expression is $Exp1(P1) \triangleq (let(P1) > x > Site1())$.

In the example, the let box has a val parameter which is assigned the P1 expression parameter. The variable is called x and the solid line indicates the use of the >> operator. Finally the publication of the variable start an instance of the $Site1$ call, however, $Site1$ does not accept any parameter.

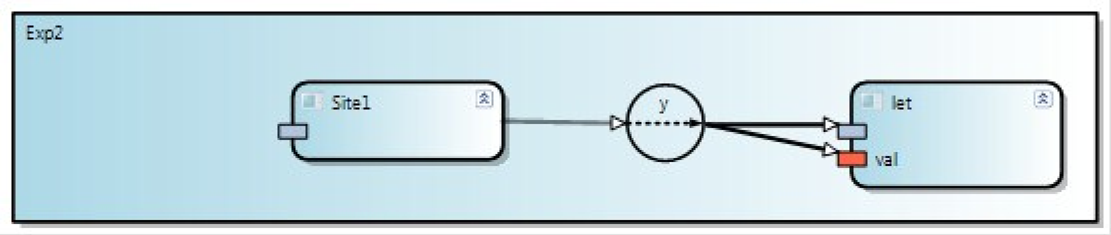

Figure 26. Example of Orc << in graphical form

Figure 26 show an example using the << operator. The expression represented in the figure is the following: $Exp2() \triangleq (let(y) < y < Site1())$. In this case, the expression has no parameter defined. The let site is connected to the variable y with 2 arrows. One is connected to the parameter val and indicate that the value of y must be used in the call (equivalent to the Orc syntax $let(y)$). The second arrow connects y to the blue rectangle and indicates dependency (i.e., the call to let depends on the first value being published to y, in Orc $let < y <$).

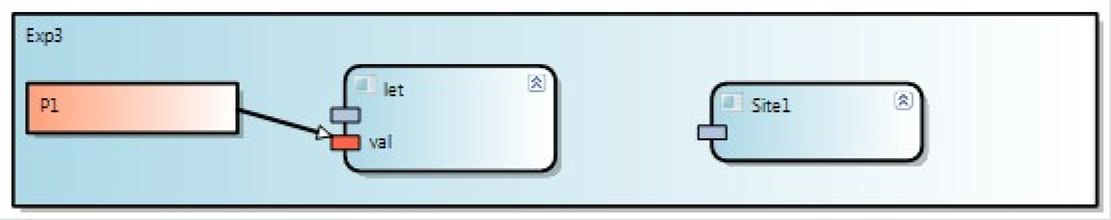

Figure 27. Example of Orc | in graphical form

Figure 27 shows the parallel operator of Orc being rendered in graphical form. In the graphic notation the dependency is explicitly modeled by arrows. Arrows from a call box to a variable indicates that the call must happen before any other call using the variable (arrows from the variable to the parameter rectangles in call boxes) and before any call depending on the variable (blue rectangle in the call boxes). For example, Figure 27 represents the following expression: $Exp3(P1) \triangleq (let(P1) \mid Site1())$. let and $Site$ execute in parallel because there is no dependency defined between them.

Finally, Figure 28 shows how pointcut interfaces are rendered in the graphical representation of Orca. The pictures show an interface definition with two pointcut variables MainService and SecondaryService (green boxes in the figure). These definitions are connected to x and y respectively. The association of an interface pointcut definition to a variable in the definition is represented by splitting the arrow inside the variable in 2 and connecting these arrows to the pointcut parameter.

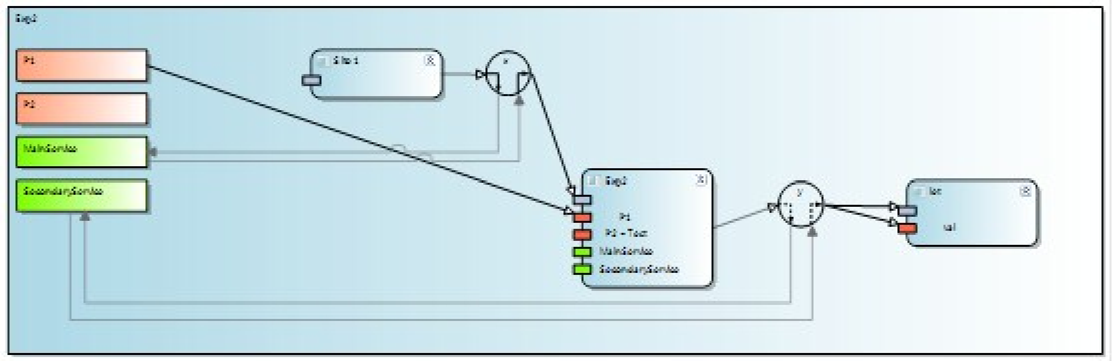

Figure 28. Orca pointcut interfaces in graphical form

The Orca expression definition corresponding to Figure 28 is the following: $Exp2(P1,P2)[x,y] \triangleq (let(y) < y < (Site1() > x > Exp2(P1,Test)))$. The graphical too does not show the types of all variables in the interface nor the return quantifier and type. Types are modeled as attributes of the boxes and now shown in the picture. The return quantifier is still not supported by the tool and left for future work.

ASPECT SEMANTICS

Orca introduces two changes to Orc: interfaces (with types and return quantifier) and aspect sites (the $@name$ used to define the aspects). Interfaces do not change the semantics of the language. They make explicit what types can be produced and consumed by sites. In Orc this information is implicit in the site definition. Also the quantifier does not change the semantics. It is just information that helps the programmer in composing services by knowing how many values they will publish. In Orc this information if implicit, still it can be obtained reading the description of sites.

The only part of an Orca interface that has the potential to change the semantics of Orc is the pointcut definition. To keep the semantics of Orca as close as possible the one of Orc, pointcut just identify variables that can be advised by aspects. However, if an expression is not called in the context of an aspect Orca defines the behavior of the expression call to be identical to Orc's. An expression is considered to be called in the context of an aspect if parameters are defined for the pointcuts. For example, considering the definition of the expression $News$ from Example 7 above, if the expression is called as $News(n)$ the semantics is the same of Orc, if it is called as $News(n)[m]$ it is considered to be in the context of an aspect and the semantics changes.

I give an informal description of how aspects work using the example in Figure 28. The figure shows that the communication inside the 2 variables, x and y, of expression $Exp2$ is split in 2. For example, the value published by the site $Site1$ is sent to the pointcut variable $MainService$. The input of the $MainService$ variable is then sent to the original output of the x variable. When $Exp2$ is used in an aspect, i.e. an expression that calls it assigning a name to the pointcut variables, messages are routed through the pointcut variable.

For example, an aspect that uses $Exp2$ would look like: $(\ ) \triangleq Exp2(t1, t2)[MS, SS] \mid @MS > x > @MS(x) \mid @SS \gg @SS("max")$. This is a regular Orc expression, however, it uses pointcuts (in square brackets) and the aspect operator @. The meaning of $Aspect$ is the following. It calls the expression $Exp2$ regularly, however, it exposes $Exp2$ internal variables, x and y. These variables can be intercepted using the aspect operator @ and are named MS and SS respectively. When $Site1$ called inside $Exp2$ publishes a value, instead of being stored in variable x and passed on to execute the internal call to $Exp2$, the value is captured and re-published by $@MS$. From $@MS$ the value is published to $Aspect$'s x variable. Consequently, $@MS(x)$ is called. $@MS(x)$ publishes the value of x in Aspect to the x variable in $Exp2$. In this example x value is just intercepted and passed back without modification. Different is the situation for $Exp2$ variable y. In this case the publication to the variable is captured by $@SS$. However, its value is forgotten (by using $\gg$ instead of $> y >$) and for each publication to the variable the value is replaced with the string “max”.

MATCH OPERATOR IN ORCA

In Chapter 4 I introduced Aspect MSCs, an aspect-oriented modeling technique for describing interactions and crosscutting concerns. The approach is based on MSCs and uses an operator called Match for aspect composition. After introducing Orca I can present a mapping between the Aspect MSC models an Orca expression and show how the Mach operator functionality can be implemented in Orca.

The first step is to map MSC diagrams to Orc expressions. MSCs have roles and messages while Orca orchestrates site calls via variables. Therefore, Orca uses sites to represent roles and variables to represent messages. Each message sent to a particular role is a call to the corresponding site having the variable containing the message as call parameter. With this mapping the MSC1 from Figure 16 can be represented in Orca as: $MSC1(\quad) \triangleq R1 \gg let(m1, p1) > x > R2(x) \gg let(m2, p2) > y > R3(y)$. The call to $R1$ without parameters is introduced to specify that the first message is produced by $R1$. The let call is used to publish the messages sent by each role into the proper variables.

To represent an Aspect MSC Orca uses interfaces and aspect operators. For example the aMSC in Figure 17 can be represented using the interface $IaMSC(\quad)[M]$. In this interface I have not included the types because the aMSC did not support them. Implementing this interface in $MSC1$, its signature becomes: $MSC1: IaMSC(\quad)[y]$. And the aspect becomes: $aMSC(\quad) \triangleq MSC1(\quad)[M] \mid @M > m > Encrypt(m) > em > @M(em)$.

While the result of the translation proposed returns an interaction equivalent to the composition with the Match operator. There is an important difference. In Orca I chose to force the explicit use of interfaces to define pointcuts. This means that a regular expression approach that such as the one proposed in Aspect MSCs is impossible. In Aspect MSC an aspect includes a pointcut expression which automatically extracts joinpoints from other MSCs. On the other hand, Orca requires that each expression specifies the pointcut by implementing an interface. This architectural choice is a tradeoff. On the one hand Orca requires more manual work to add and modify pointcuts; on the other hand the author of an expression is in control to how the expression can be modified. By choosing this approach I traded convenience for model maintainability.

SUMMARY

This chapter introduced the Orca orchestration language. Orca can model both the orchestration of services (by means of expressions) and crosscutting concerns (by means aspects). Orca can also encode MSCs and Aspect MSC. Thus, it is a perfect candidate for modeling systems according to the Rich Service pattern.

While a prototypical tool exists to model Orca expressions and manipulate them in graphical form, a complete tool chain is needed to support all development activities. In particular, Orca needs a run time execution environment similar to the tools available for Orc. Additional future work includes porting the tools developed for MSCs and part of the M2Code tool chain to Orca.

CHAPTER 6

MANAGING MODEL CONSISTENCY

Systems models are always decomposed according to some dominant concern. However, software systems always need to address multiple concerns. Some concerns are then bound to cross-cut the hierarchies according to which the system has been decomposed. This fact, known as the tyranny of the dominant decomposition [61], is addressed in the rich services pattern using infrastructure services. I introduced examples of aspect-oriented modeling languages that help in specifying RIS and address crosscutting concerns. In particular, Chapter 5 presents Orca, an orchestration language that supports specification of crosscutting concerns as aspects. However, to fulfill my vision of an end-to-end model-based approach, that supports the integration of large scale software systems, two issues stand still in the way.

The first issue is managing the consistency of the models used in my approach. In fact, model-based development of large systems requires composing multiple documents, each capturing part of the system, in a coherent model. Large systems imply that some of the services, and their models, are developed by different organizations. This is the case, for example, in the automotive industry, where OEMs and their multiple suppliers develop different parts of the car. A key requirement for this type of development is to manage inconsistencies and contradictions that always arise when systems are developed by different teams.

The second issue originates from the fact that different languages are used to model different aspects of a system. This is important not only because my approach targets diverse domains, such as enterprise and embedded domains; but also because, even in the same domain, different teams of domain experts are trained in using different languages and notations to model their parts of the system. Thus, a viable approach for large systems in different domains must be able to cope with the diverse notations already existing. The problem of addressing consistency is then even more challenging because it has to assess consistency of different views of the same system that use different representations.

While in this thesis I focus on interaction models, which are key to the development of service-oriented systems, I am well aware that they are only a part of the whole picture. In any real system there is the need for other types of models. For example, in Chapter 3 I use a failure hypothesis model that models how services can fail in relation to their deployment on different hardware devices.

In this chapter I address these two remaining issues. I present a solution for the issue of managing consistency across multiple languages using the UML that is comprise of a rich set of graphical languages. In particular, I focus on embedded systems models by using Modeling and Analysis of Real-Time and Embedded Systems (MARTE) a profile of the UML for embedded systems. My solution uses and approach that I called query and constraints [62], which supports mapping different graphical languages to a common kernel close to the implementation domain. I can use a similar approach for mapping different interaction models to Orca. As a case study I use the Bay Area Rapid Transit System (BART) system. This case study nicely combines elements that are typical of embedded and enterprise systems.

In this chapter I discuss the topic of model consistency as a separate topic from the work presented in previous chapters. Instead of using Orca and MSCs as languages for modeling systems according to the rich service pattern, I present the consistency work using a UML case study. As previously mentioned, this separate contribution is important for supporting model base development in real development scenarios. While the abstract model used as target for integration in this chapter could supports systems modeled in Orca, the necessary integration with the previous work is left for future work.

MULTI-VIEW MODELS AND CONSISTENCY CHALLENGES

When using multiple modeling perspectives, the central question from an engineering point of view is this: is the modeled system realizable? Oddly, the UML is an example of a modeling language that while being used in real development projects, does not provide a complete formal semantics. This fact alone leaves issues such as model consistency unsolved. In discussing the UML consistency problem in detail, I explain how it originates or is worsened by the tradeoffs in the language design, and propose an avenue to solve it.

My first task in presenting the problem of model consistency is to clearly define what kind of consistency I am interested in and how to effectively determine whether or not a model is consistent. The UML is a broad-spectrum language with an informally defined semantics, which serves the goal to be inclusive with respect to modeling styles and domains. However, this creates the first hurdle that must be overcome to define consistency. Any approach aiming at defining consistency needs to explicitly or implicitly define a precise semantics for the UML. A rich body of work exists in the literature on defining multi-view or multi-perspective consistency based on UML semantics definitions. I have presented an extensive analysis of this work in Ref.[63].

Although the consistency problem has been extensively studied in the literature, a solution has been elusive – especially in the context of the UML with its rich set of inter-related description techniques for system structure and behavior. Existing approaches to defining UML model consistency lead to complex definitions of the notion of consistency, or address only a subset of the available modeling notations. My goal is to create a consistency checking approach that is flexible enough to be able to target the full UML language. My approach does not force a developer to fully define the semantics of all UML notations; only the semantics of a subset (profile) of the UML used in the specification must be defined.

The main novelty of the consistency checking approach presented in this chapter is in the comprehensive, yet simple mechanism introduced for specifying consistency rules. Instead of analyzing the semantics of the UML at the metamodel level and extracting consistency rules between different diagram types, I define a simple execution framework (similar to a "virtual machine"), based on a target ontology whose concepts map one-to-one to elements of the system class we are interested in modeling, i.e. distributed, reactive systems. All UML diagram types are then treated as model generators for this virtual machine; each diagram selects entities of the virtual machine and constrains their structure or behavior. Model consistency is then simply defined as the presence of virtual machine behaviors under the specified constraints.

THE BART CASE STUDY

To show the modeling capabilities of the UML, we use a simplified example of the Bay Area Rapid Transit [64] system, particularly the part of the train system that controls speed and acceleration of the trains. BART is the commuter rail train system in the San Francisco Bay area. A full description of

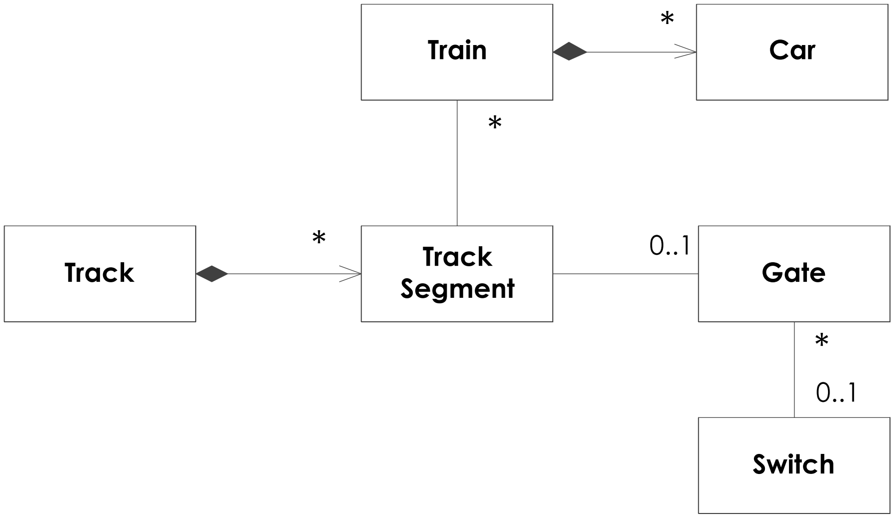

Figure 29. Domain Model for the BART tracks

the case study is beyond the scope of this chapter, so we will exemplify some of the UML diagrams that can be used for modeling such a system – use case, class, sequence, and state diagrams.

The BART system automatically controls over 50 trains, most of them consisting of 10 cars. Tracks are unidirectional and sections of the track network are shared by trains of different lines. A track is partitioned into track segments, which may be bounded by gates. A gate can be viewed as a traffic light, establishing the right-of-way where tracks join at switches. Figure 29 depicts a domain model for the BART track system, showing in a UML class diagram the relationships between physical entities such as train, track, and gate. Such models facilitate establishing a common language for eliciting requirements from domain experts. Typically, specifying relationships and multiplicity constraints on a domain model leads to further discussions with the stakeholders to clarify the domain. For example, gates are not necessarily associated with switches, but can be used just to control the traffic flow.

Other work [64] describes the Advanced Automatic Train Control (AATC) system, which controls the train movement for BART. One important AATC requirement is to optimize train speeds and the spacing between the trains to increase throughput on the congested parts of the network, while constantly ensuring train safety. The specification strictly defines certain safety conditions that must never be violated, such as “a train must never enter a segment closed by a gate”, or “the distance between trains must always exceed the safe stopping distance of the following train under any circumstances".

The system is controlled automatically. Onboard operators have limited responsibility: they signal the system when the platforms are clear so a train can depart a station and they can operate the trains manually when a problem arises. Use case diagrams are useful in identifying the system boundaries (the control system that must be designed) and the external actors that interact with the system. Typically in UML, actors are human actors that use an application, but in embedded systems actors can be external physical resources such as devices and sensors. Nevertheless, actors represent logical roles, so a physical resource could play several roles in UML models. Figure 30 depicts a simple use case diagram for BART. Actors that interact with the AATC system are the Train and the Train Operator and so they are part of the system environment. The use cases depict the high-level goals of the system without details on how these goals are accomplished.

AATC consists of computers at train stations, a radio communications network that links the stations with the trains, and two AATC controllers on board of each train - the two controllers are at the front and back of the train. A track is not a loop. Thus, at the end of the line, the front and back controllers exchange roles, and the train moves in the other direction. Each station controls a local part of the track network. Stations communicate with neighboring stations using land-based network links. Trains receive acceleration and brake commands from the station computers via the radio communication network. The train AATC controller (from the lead car) is responsible for operating the brakes and motors of all cars in the train. The radio network has the capability of providing ranging information (from wayside radios to train radios and back) that allows the system to track train positions.

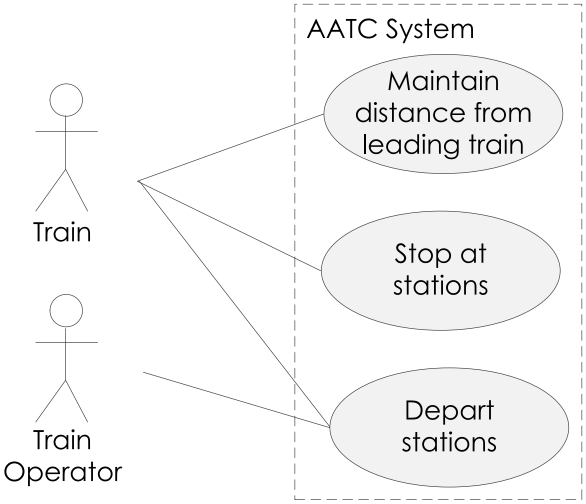

Figure 30. BART AATC system use case

The system operates in half a second cycles. In each cycle, the station control computer receives train information, computes commands for all trains under its control, and forwards these commands to the train controllers. Figure 31 shows a sequence diagram depicting the interactions between three roles called Train, Station AATC, and Train Controller. Note that the Station AATC system obtains the status information directly from the Train by using the radio network, not from the Train Controller.

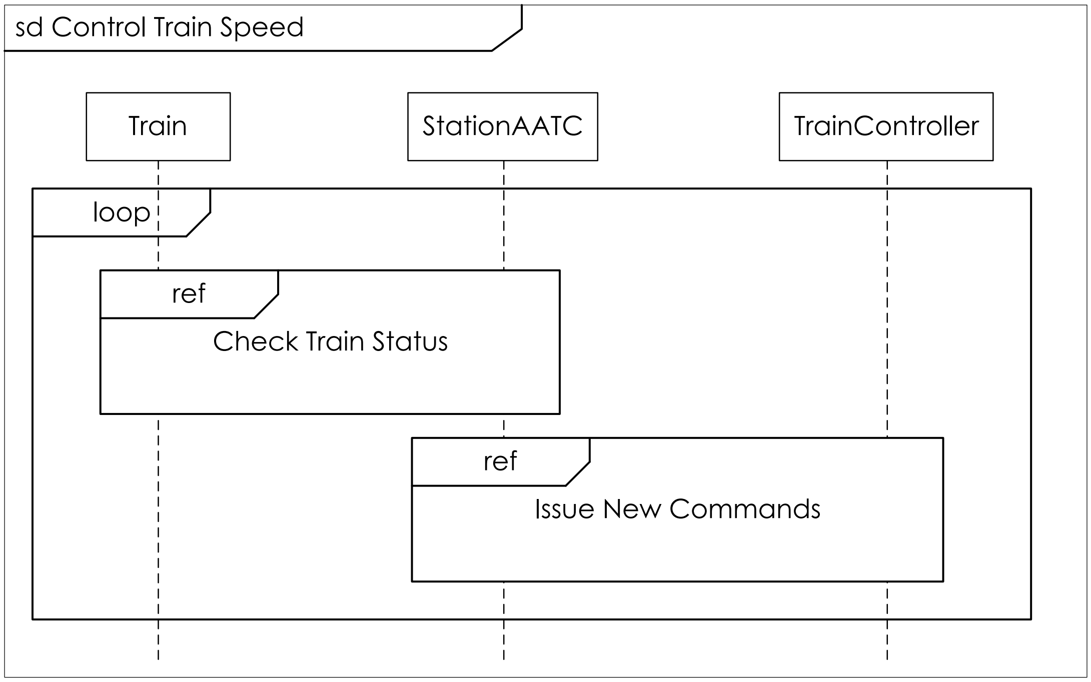

Figure 31. Train speed sequence diagram

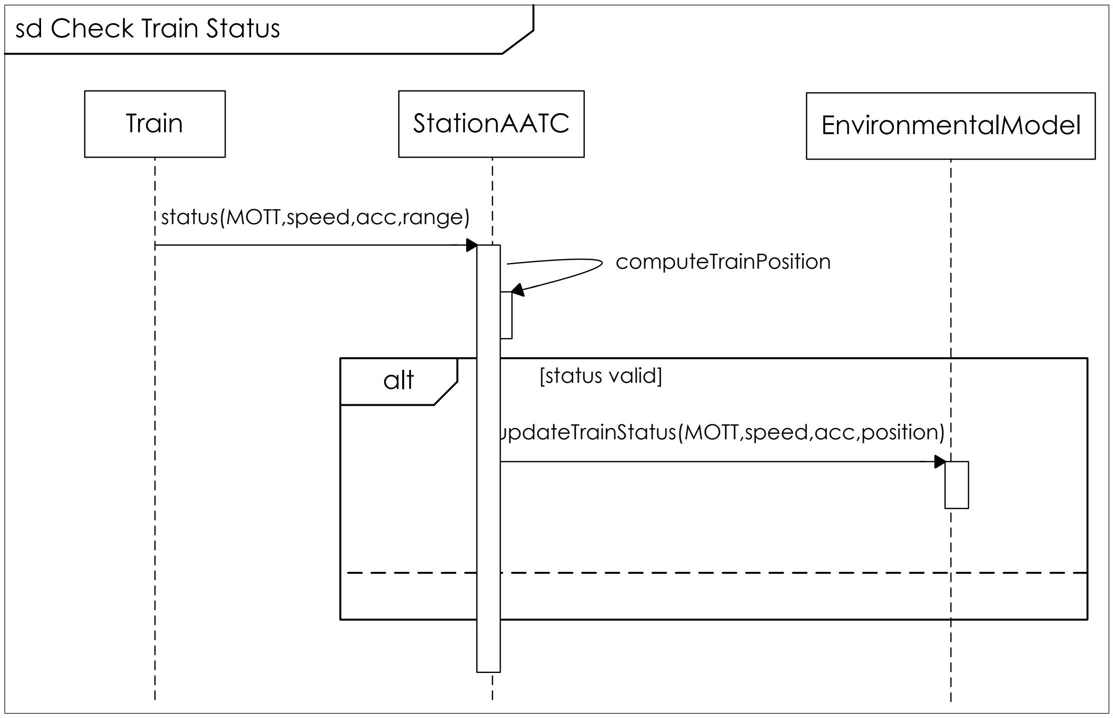

Figure 32. BART Check Train Status sequence diagram

The sequence diagram features interaction frames, introduced in UML 2.0. A frame provides the boundary of a diagram and a place to show the diagram label (e.g., “Control Train Speed” in Figure 31). Frames also allow specifying combined fragments with operators and guards. Common examples of operators are LOOP for repetitive sequences, ALT for mutually exclusive fragments, and PAR for parallel execution of fragments. Figure 31 uses a LOOP operator to show that the system repeats the sequence of checking the train position and issuing new commands. Another operator is REF, which creates a reference to an interaction specified in another diagram. This REF operator allows composing primitive sequence diagrams into complex sequence diagrams. The expressiveness of UML 2 increased with the addition of these operators, which are borrowed from Message Sequence Charts (MSCs) [28], [65].

Figure 32 depicts a simplified Check Train Status sequence diagram as referenced in Figure 31. The Train sends status information regarding its speed, acceleration, and range. The Station AATC system computes the train position from the status information and updates its Environmental Model. Status messages and commands are time-stamped in the so-called Message Origination Time Tag (MOTT). When a Train sends status information to a station, it attaches the time it sends the message as a MOTT. When the Station AATC estimates the train position, it attaches the original MOTT to the estimate. Furthermore, when the Station AATC sends a command, it again attaches the original MOTT, and the Train Controller checks the MOTT before executing the command. The station's control algorithm takes the MOTT, track information, and train status into account to compute new commands that never violate the safety conditions. To ensure this, each station computer is attached to an independent safety control computer that validates all computed commands for conformance with the safety conditions.

The actors in sequence diagrams (e.g., Train, StationAATC, etc) are logical roles – in modeling the interactions, we concentrate on specific use cases and abstract from any concrete deployment architectures. In essence, a role shows *part* of the behavior the system displays during execution. What concrete deployment entity *plays* this role is left for a later modeling stage. The natural modeling entities for roles in the UML are Classifiers – with the understanding that multiple roles may be aggregated into a single Classifier. The roles related to computing commands and safety are omitted from Figure 32, as they are relevant for another sequence diagram, called Issue New Commands, shown later in this chapter. The roles visible in a sequence diagram are a subset of the roles of the entire system.

Figure 33 shows a simplified domain model with the roles mentioned so far. We use the notation of a class diagram without the multiplicities – for a role domain model we are interested in the roles that communicate and the links between them. The same diagram can be seen as a simplified Communication diagram, showing the communication links without the messages being exchanged. The role domain model is part of the logical

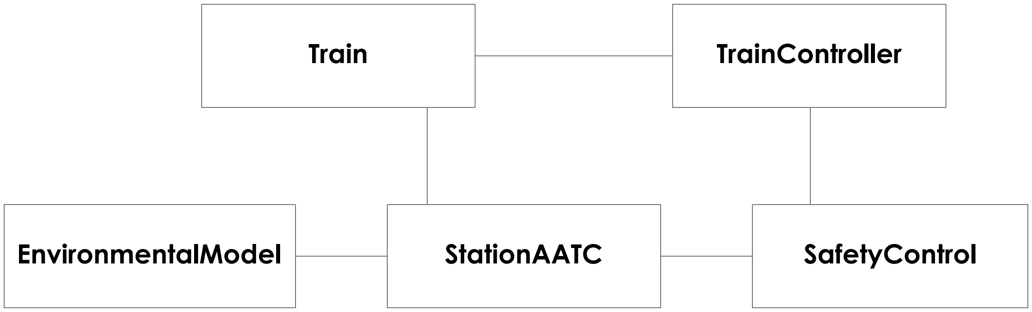

Figure 33. Domain model of BART roles

architecture, as roles are logical entities that are later mapped onto physical components to define the technical architecture. A component can play several logical roles.

If a train does not receive a valid command within two seconds of the timestamp contained in the MOTT accompanying the status, it goes into emergency braking. Figure 34 shows the behavior of the Train Controller as a state-machine diagram with two states for normal operation and emergency mode.

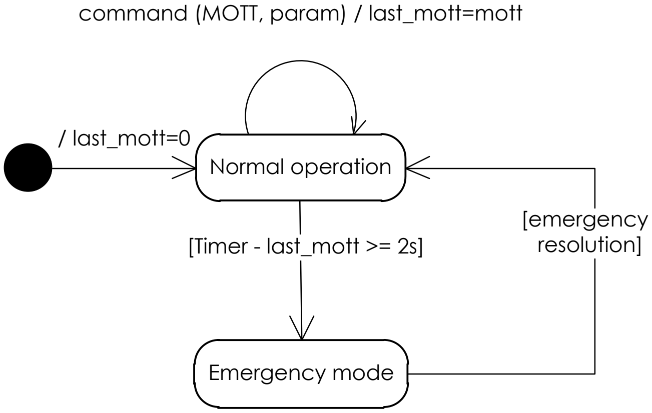

Figure 34. State machine diagram for BART train controller

In state-machine diagrams we show states as boxes with rounded corners. Arrows denote state transitions. Labels on arrows indicate (i) the trigger (such as a message received), (ii) a guard (a condition that must be true for the transition to be taken) in angular brackets, separated from (iii) the action (to be performed when the transition is taken) by a "/". Actions include assignments to state variables and the sending of messages. All three pasts of a transition are optional. A solid circle indicates the initial "pseudo" state.

This example shows a frequently used pattern in modeling time with the basic capabilities of the UML: time is represented as an explicit parameter in messages exchanged among actors and these actors then perform explicit time arithmetic to determine transition triggers.

INCONSISTENCY EXAMPLE

I revisit the example from the Bay Area Rapid Transit (BART) system, introduced in Chapter 3. BART is the commuter rail train system in the San Francisco Bay area. The BART system automatically controls over 50 trains on a large track network with several different lines. Figure 35 shows three modeling perspectives of BART using UML 2.3 and the MARTE profile. Figure 35a shows the component diagram defining the structure of the system. Figure 35b shows a sequence diagram which models the train commands computation and delivery. Finally, Figure 35c and Figure 35d shows state-machine diagrams describing the Emergency Brake system component. In this case study I discuss an inconsistency that can arise when modeling behavior in the different diagrams, namely sequence and state-machine diagrams.

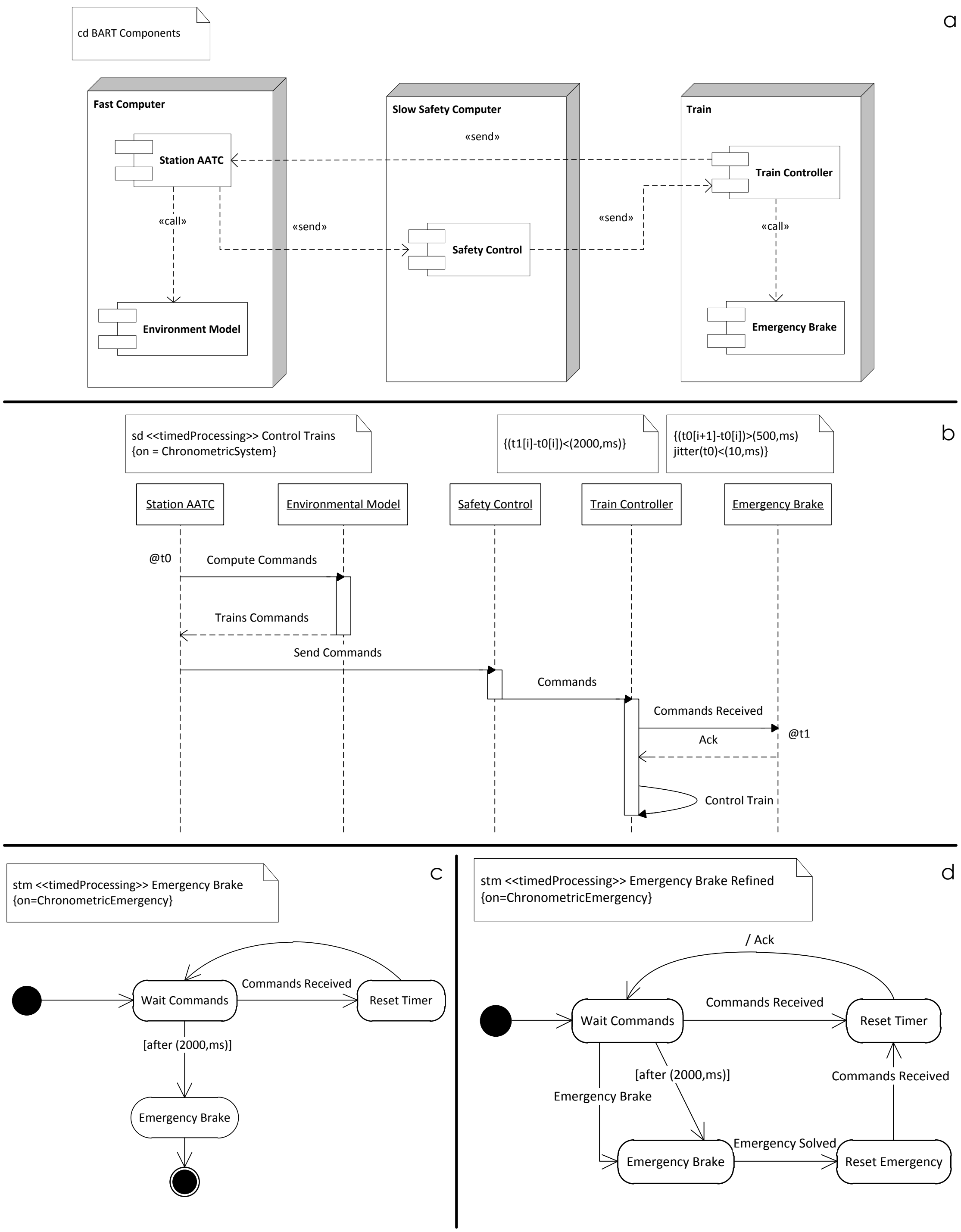

Figure 35. Three different perspectives of the BART case study

The focus of this case study is on the Advanced Automatic Train Control (AATC) system, which controls the train movement for BART. The AATC system consists of computers at train stations, a radio communications network that links the stations with the trains, and AATC controllers on board of each train. Most of the control computation is done at the stations. Each station is responsible for controlling all trains in its area. Trains receive acceleration and brake commands from the station via the radio communication network. The train controller is responsible for operating the brakes and motors of all cars in the train. Controlling the trains must occur efficiently with a high throughput of trains on the congested parts of the network, while ensuring train safety. The station's control algorithm takes the track information, train speed and acceleration, train position estimation, and information from the neighboring stations into account to compute new commands that never violate the safety conditions. To ensure this, each station computer is attached to an independent safety control computer that validates all computed commands for conformance with the safety conditions.

The component diagram for AATC is depicted in Figure 35a. It has three nodes: two for the train station and one for the train. The first node, Fast Computer, represents the station computer that computes the commands to be sent to all trains under the control of that station. It contains two components: one represents the Station AATC control system and the other called Environmental Model, which models the physical environment of a station. The Station AATC uses the Environmental Model to compute commands to send to trains. The second node, Slow Safety Computer, contains the Safety Control component, which checks all commands sent by the Station AATC for safety before forwarding them to each train. The safety computation is based on a simpler model than the one used to compute commands and, therefore, requires less computation resources. However, the Slow Safety Computer is required to have high reliability. The third node in the figure is the Train. It has two components: the Train Controller manages the train accelerations and decelerations, and the Emergency Brake is activated only in case of an emergency and stops the train as quickly as possible.

The AATC system operates in half a second cycles. In each cycle, the station receives train information, computes commands for all trains under its control, and forwards these commands to the train controllers. The Station AATC system obtains the status information regarding train speed, acceleration, and range by using the radio network, which allows the system to track train positions. The Station AATC system computes the train position from the status information and updates its Environmental Model. Then, the Station AATC interacts with the Environmental Model and the Safety Control components to compute and send the new commands, as depicted in the sequence diagram from Figure 35b. The behavior specified in the diagram is the following:

- Station AATC sends a request to Environmental Model to compute the commands for the train.
- Environmental Model computes the commands, taking into account all parameters such as passenger comfort (e.g., not too strong braking

and acceleration changes), train schedule, engine wear, and most importantly safety.

- After receiving the commands from Environmental Model, Station AATC sends the commands to Safety Control to ensure the commands computed are safe.
- Safety Control checks that the commands do not exceed maximum bounds for safety. If the commands are safe, Safety Control forwards them to Train Controller.
- Train Controller informs Emergency Brake that the commands have been received.
- Emergency Brake acknowledges the commands received.
- Finally, Train Controller controls the train engine according to the commands received.

The model in Figure 35b is annotated with MARTE time constraints to specify the real-time requirements of the BART case study. I annotated two time instants t0 and t1 using TimedInstantObservations as defined in MARTE, which is indicated by the graphical representations @t0 and @t1. A TimedInstantObservation denotes an instant in time associated with an event occurrence (e.g., send or receive of a message) and observed on a given clock. T0 is the instant when the message Compute Commands is *sent* by Station AATC whereas t1 is the time instant when the message Commands Received is *received* by Emergency Brake. Because the system operates in cycles, the notation t0[i] and t1[i] represents the generic i^{th} instantiation of the interaction scenario.

Given those two instants, I leverage MARTE to define three time constraints in our system. Commands to trains become invalid after two seconds. If a train does not receive a valid command within two seconds, it goes into emergency braking. Therefore, with the time constraint (t1[i]-t0[i]) < (2000,ms) we limit the duration of each iteration of this scenario to two seconds. The AATC control algorithm needs to take this timing constraint, track information, and train status into account to compute new commands that never violate the train safety. The second constraint, (t0[i+1]-t0[i]) > (500,ms), imposes that between each instantiation of the scenario at least half a second passes. Finally, the last constraint, jitter(t0) < (10,ms), limits the jitter of the t0 event enforcing that between each iteration of the event at t0 there are between 500 and 510 ms.

In normal operations, the AATC system computes the train commands in fixed time cycles. However, in case of a detected emergency condition, the system has to react immediately and take appropriate measures to ensure maximum safety of passengers and equipment. Figure 35c and Figure 35d present state-machine diagrams for the Emergency Brake component. A train will continue to exercise a command until a new one arrives or until that command expires, two seconds after the originating time. The state-machine diagram for the Emergency Brake has states for waiting for commands and entering emergency mode if the timer of two seconds expires. When commands are received, the timer is reset. These state machines are two different versions of the same perspective where the one in Figure 35d is a refined version that enables restarting the system after an emergency brake. If we consider the three graphs from Figure 35a, Figure 35b, and Figure 35c together, we have an inconsistent model: the state-machine diagram Figure 35c does not acknowledge the Commands Received call from Train Controller – contrary to what the sequence diagram from Figure 35b demands. Replacing the diagram from Figure 35c with Figure 35d, we obtain a consistent model.

UML MODEL CONSISTENCY REQUIREMENTS

I have identified 12 important requirements (collected in Table 3) by analyzing the requirements discussed in the literature for current approaches to model consistency. Requirements R1 to R3 in Table 3 originate from the observation that any strategy to manage model consistency should not limit the freedom of developers. This entails that developers should be allowed to modify models even if they introduce some inconsistencies. This idea is introduced in [66], where the authors observe that inconsistency is necessary and often desirable in some phase of the development cycle. For example, in the inception phase of a large project with different stakeholders involved, each stakeholder pursues different goals and, during the collection of requirements, this can lead to inconsistent views that must be identified and reconciled in subsequent iterations. Other arguments in support of Requirements R1-R3 have been documented elsewhere [67–69]. The common

Table 3. Requirements for UML consistency management.

	Requirement Description
R1	Inconsistent models can be introduced and kept in the system specification for a certain amount of time.
R2	Inconsistencies should be discovered automatically and tracked during the evolution of model.
R3	Support should be provided to the developer to resolve inconsistencies when convenient.
R4	Support multiple modeling languages (for example, different UML notations or even non-UML languages).
R5	Support different levels of abstraction.
R6	Support the extension or specialization of languages.
R7	Support Horizontal consistency.
R8	Support Vertical consistency.
R9	Support Static consistency.
R10	Support Dynamic consistency.
R11	Tool support (or translations to available tools).
R12	Scalability to large models.

denominator of all arguments is that effective modeling techniques must support decomposing the problem into independent subproblems. This is the case when in order to solve complex problems; engineers decompose various aspects of the system and reason about each aspect in isolation. Alternatively, this occurs when in order to solve complex problems efficiently, different teams work in parallel on different aspects of the system.

A second observation is that each model caters to different needs that arise during the development process. For example, informal models are used to gather requirements and exchange ideas between stakeholders and developers during requirements gathering [35]. Later in the development process more formal models are used to describe the structure or the behavior of certain parts of the system. In this phase, formal models are used to verify properties of a system or to generate part of the implementation code. This second observation is the source of the additional requirements R4 to R6 in Table 3.

To evaluate consistency management techniques the notion of consistency must be clearly defined. The scientific literature examines different notions of consistency. A distinction can be made between *Horizontal* and *Vertical* consistency [70], [71]. *Horizontal* consistency involves different perspectives on the same system model. For example, on the one hand, to describe the communication between a client and a server, it is possible to use a UML sequence diagram to capture the protocol and a state diagram to capture the server behavior. The two diagrams are different views on the same system and should be horizontally consistent. On the other hand, *Vertical* consistency addresses views of the same aspect of one system, but at different levels of abstraction, often in relation to the evolution of one model during different phases of the development process. For example, an abstract model created during requirements gathering must agree with a more detailed model used for code generation in a later step of the development process. Another important distinction is between *Static* and *Dynamic* consistency [72]. *Static* consistency addresses syntactical and structural model dependencies while *Dynamic* consistency ensures the consistency of executable models. Four requirements (R7 to R10 in Table 3) capture these 4 notions of consistency.

The final two requirements address practical use of consistency management techniques. Requirement R11 recognizes that consistency checking must be supported by a tool chain. Requirement R12 recognizes that industrial systems are large scale and this implies they have large system models. Therefore, scalability of the chosen technique to large models is an important requirement.

SOLVING UML CONSISTENCY

None of the approaches available in the literature fully address all requirements of Table 3. The common challenge of previous work is in losing track of the abstractions implemented in the models that are checked for consistency.

Previous work has taken two routes: either analyzing the semantics of the diagrams at the metamodel level (or defining consistency rules between different notation types from there) or translating the models into an existing formal language leveraged for verification. In contrast, the approach I follow here defines an explicit ontology that captures the target domain of the models. Based on this target ontology I define a simple execution framework (similar to a "virtual machine"). The ontology concepts map one-to-one onto elements of the system class I am interested in modeling.

The main novelty of the consistency checking approach presented here is in the comprehensive, yet simple mechanism introduced for specifying consistency rules. By defining a simple "virtual machine" containing the abstraction used in our models, I can treat all UML diagram types as model generators for this virtual machine. Each diagram selects entities of the virtual machine and constrains their structure or behavior. Model consistency is then simply defined as the presence of virtual machine behaviors under the specified constraints.

I encode constraints as a set of logic propositions over elements of the target ontology, and reduce the verification of virtual machine behaviors to a satisfiability problem. While the work presented here is specific to the UML, the same approach can be leveraged to integrate other modeling languages (such as the Orca language described in the previous chapter) with UML-like models.

For the proposed approach to work, first I tailor the UML to the target domain. I leverage the UML MARTE profile to target embedded real-time systems. For the purposes of this chapter, I limit the scope of the discussion to a subset of the MARTE notations, rich enough to show the value of the consistency verification technique I am proposing. In particular, in this chapter I include State Diagrams, Component Diagrams, and Interaction Diagrams. In the Discussion section below, I analyze avenues for extending this approach to a richer subset of UML 2.0 and to other modeling languages.

QUERIES AND CONSTRAINTS SEMANTICS

To provide the backdrop for my definition of model consistency, I provide a formal semantic framework based on an abstract model of distributed reactive systems, similar to a "virtual machine". I call this model of our target domain the "*abstract semantic space*". In this space, I show how each element of a model can be interpreted as a constraint on the system. The consistency property can then be trivially defined over the "abstract semantic space" as the existence of a system in that domain that satisfies all constraints imposed by the models.

The semantics is based on two elements: queries and constraints. Each model element of a UML specification is interpreted as a set of (query, constraint) tuples. Each query selects some elements in the "abstract semantic space" that we have defined where the corresponding constraint defines a restriction on the structure or behavior of these elements in a system satisfying the specification. The key benefits of this approach are: (i) a mathematically simple, yet comprehensive definition of consistency, (ii) the ability to tie the reasoning about consistency to entities of the target domain – resulting in a non-generic model subclass to which the consistency notion applies, and (iii) the interpretation of model elements as constraints over the target domain.

This consistency checking approach contrasts with other translation-based approaches in the literature in the way I perform the translation. In fact, the target model of my translation abstracts the main components of the target implementation domain. The semantics is then specified by directly mapping each element of the UML model onto some configuration of the target model. The first step is to define an ontology for real-time distributed systems. This ontology is used to assign precise semantics to the UML models used and is formalized with Queries and Constraints. This step allows me to formally reason about the specification (using first order logic). After the formalization, I present the grammar of a language to describe systems based on the target ontology formalism. This step enables the translation of UML models to the new domain. The final steps are the definition of the semantics for the abstract language and, based on such semantics, the definition of consistency.

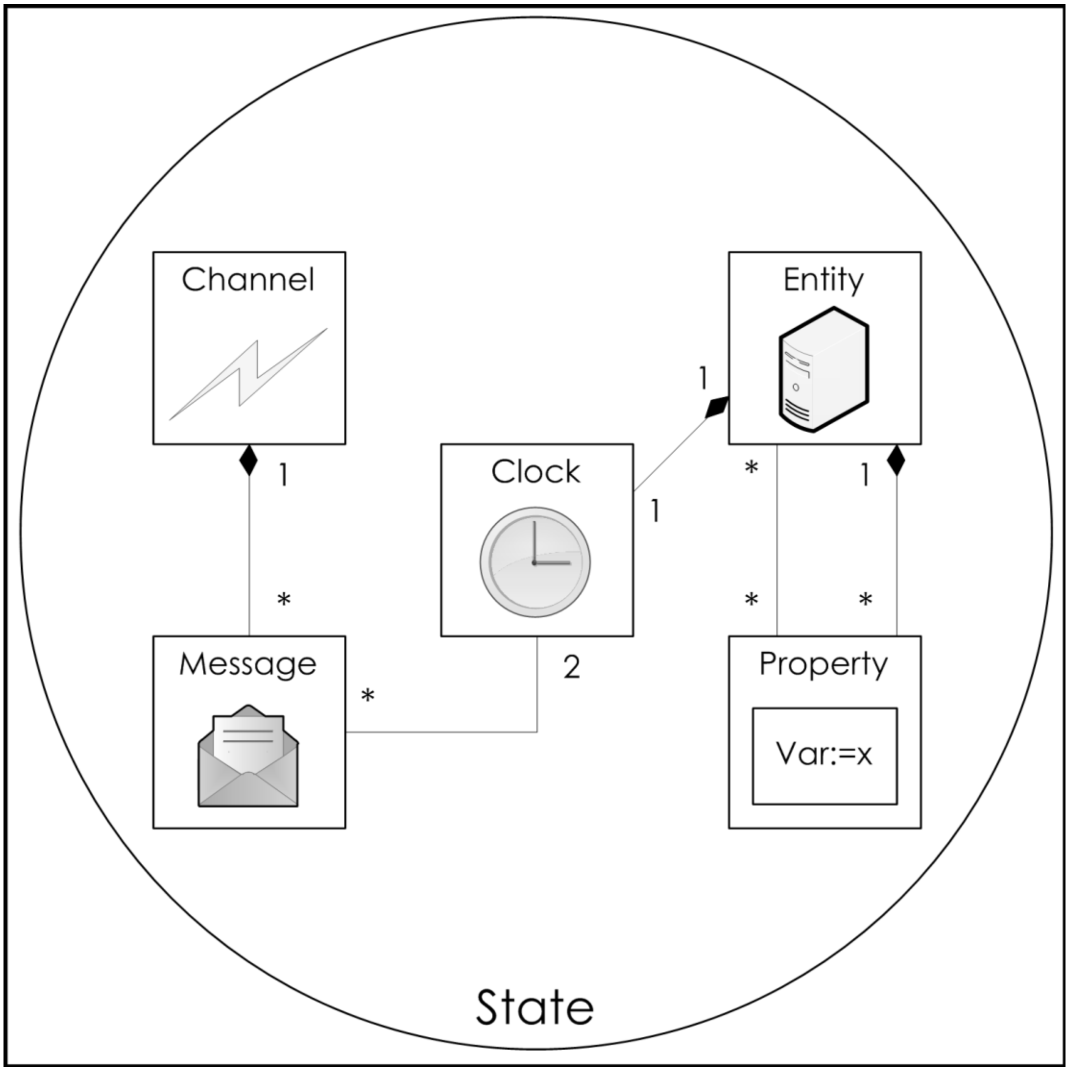

Figure 36. Core elements

Figure 36 captures the core elements of my ontology for distributed systems with real-time constraints. A real-time system in this ontology is described by five types of elements: two elements, Entities and Channels, form the structural configuration of the system; another two, Messages and Properties, define the behavior; and the Clock captures real-time constraints.

An Entity captures the concept of a process in a distributed system. An Entity has local variables, captures state information, has computational capabilities, and can communicate with other Entities by means of sending and receiving messages over a set of channels. Channels are the communication infrastructure. Each entity that must send or receive messages does so leveraging some specific channels. Channels transport Messages. When a message is sent on a channel, all entities that are using there channels eventually receive the message. Properties can be used to capture variables and their state. Each entity has a named set of properties that can be evaluated at run-time. Finally, the Clock captures the time relative to an entity. I could have used different notions of time, the choice depends on the type of system I am modeling and the profile of the UML in use. MARTE supports not only the type of time modeled here, but also other time models, for example, modeling of synchronous reactions.

Figure 36 shows these five core elements forming the *abstract* state of the system. At each instant the structural part of the system state is defined by the existing Entities and Channels. The behavioral part is defined by the Messages exchanged on each Channel and by the internal state of each

Entity defined by the valuation of its Properties. Timing relations are expressed by the collections of all clocks associated to entities. Each Entity has its own reference of time given by the clock. At any given instant, different clocks can have different time values. It is interesting to note that, because the state comprises both a behavioral and a structural part, it is possible to represent a reconfiguration of the system as a change of state.

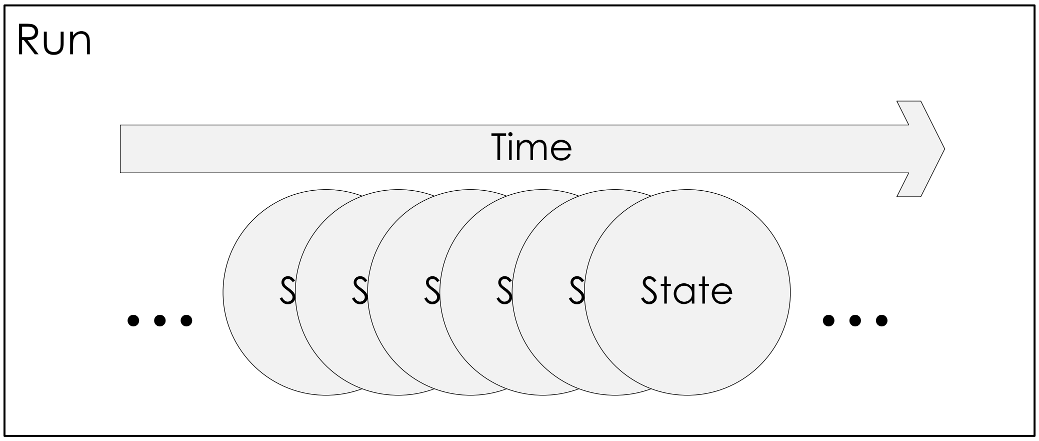

Figure 37. Definition of a run

Based on the concept of state, I can now define a run as an infinite sequence of states (cf.Figure 37). In turn, I now define the semantics of a system based on runs. Figure 38 shows the full ontology that used to assign a semantics to the UML. A system is defined by a set of runs. A specification defines a set of acceptable runs. The specification can constrain the acceptable runs by specifying the initial states and the acceptable transitions.

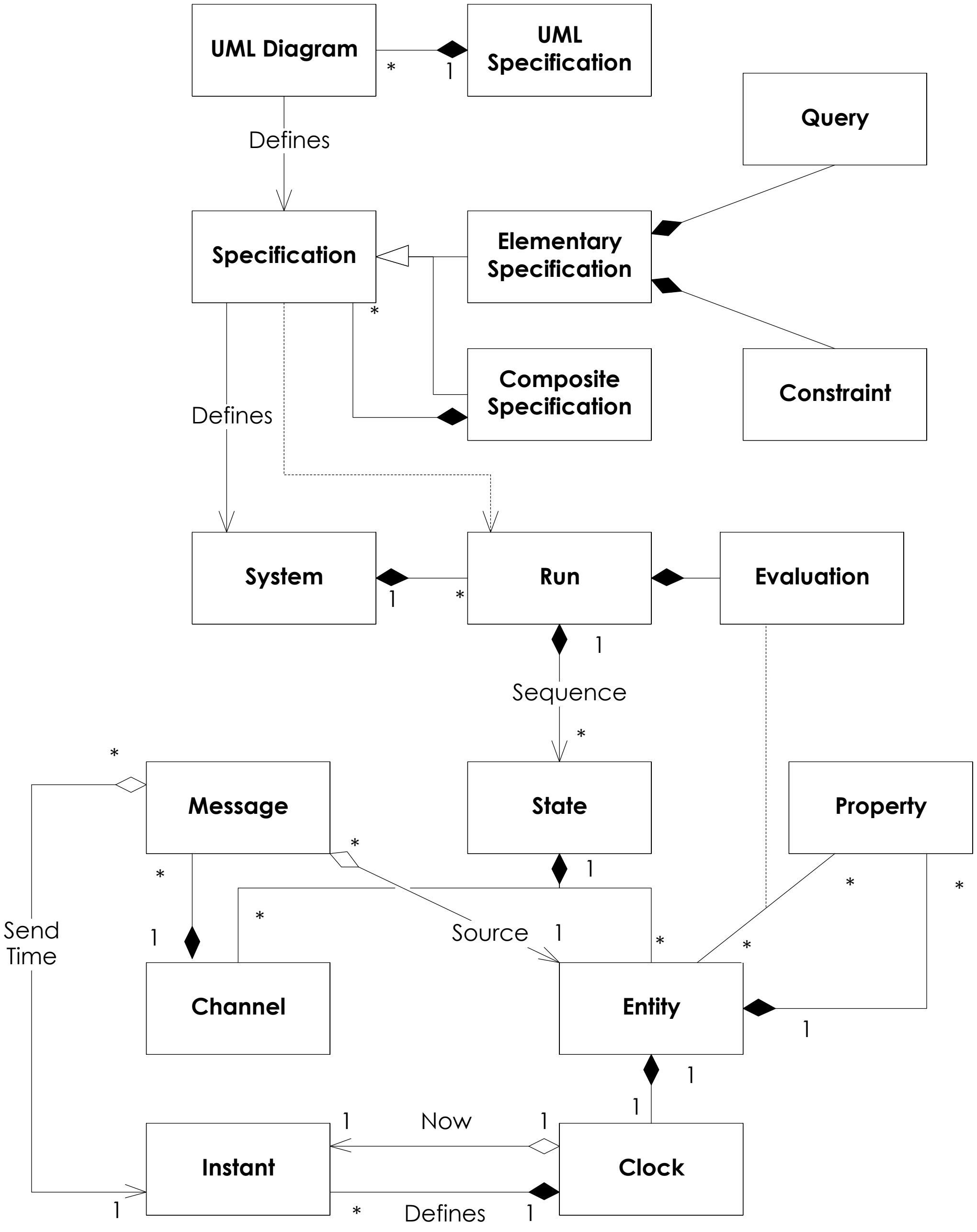

Figure 38. Ontology for distributed real time systems semantics

Another interesting element of Figure 38 is the definition of Specification. A Specification can either be composite or elementary. Every elementary specification is made up of two elements: Query and Constraint. A Query selects states from all possible runs while the corresponding Constraint defines the characteristics for the run to be acceptable. Think of the selection as an operator that is applied to all possible runs. All states selected by the Query are compared with the rules specified in the constraints. If they match, the run is accepted as part of the system whereas if they do not match, the run is discarded.

An important point to notice is how time is treated in the ontology. Each Entity has access to one private Clock. The Clock Defines a series of Instants. At any given time the Clock refers to one of the instants as Now. Each message has one Send Time (an instant on the clock of the entity that sends the message) and one Source entity (the sender of the message). Therefore, it is possible to reason about when each message was sent and by which entity. Messages can be received by different Entities at different times. When an Entity that has subscribed to a channel receives a message, it can identify the local entity time using its clock and obtain the time of the sending entity from the message. Depending on the system and the requirements, it is possible to define synchronization strategies between the clocks so as to be able to reason about times of events across different clocks.

I can now give a formalization of the semantics informally described above. To this end, I first formalize the concepts of state and run as a foundation for the semantics of a distributed system specification. Then I present a simple grammar for a specification based on Queries and Constraints and use the formal definitions introduced before to provide a semantic for it.

NOTATIONAL PRELIMINARIES AND SYSTEM FORMALIZATION

I represent sets with capital Greek letters. For instance, the set of properties will be represented by Ψ. Each element of the set will be represented by the corresponding lowercase letter. For instance, a property in Ψ would be represented by ψ. A function from a domain A to a co-domain B is expressed as $\varphi: A \rightarrow B$. A tuple is defined as $y = (y_1, y_2, \cdots) \in Y_1 \times Y_2 \times \cdots$ and $\pi_i \cdot y = y_i$ is the projection operator returning the i^{th} element of the tuple. Given a set X, $\mathcal{P}(X)$ is the powerset of X where $|X|$ returns the cardinality of X. Furthermore, with $\mathbb{B}$ we indicate the set of Boolean values (true and false), with $\mathbb{N}$ the set of natural numbers, with $\mathbb{N}_+$ the set of natural numbers without 0, and with $\mathbb{N}_\infty$ the set of natural numbers with its supremum ∞.

A stream [56] is a finite or infinite sequence of messages. Given a set of messages M: M^* is the set of finite sequences over M, M^∞ the set of infinite sequences, and with M^ω the union of those two sets. The infix dot operator $x.i$ returns the i^{th} element of a stream x. The notation $x \downarrow i$ returns the prefix stream of length i, whereas $x \uparrow i$ returns the tail stream obtained by removing the first i elements from x. The concatenation of two streams x and x' is denoted as $x \frown x'$. This notation is overloaded to work with sets of streams $X \frown X'$ such that the resulting set contains all streams of the form $x \frown x'$ where $x \in X \wedge x' \in X'$.

I can now give a formal definition of the elements of our ontology. For the two structural elements, Entities and Channels, I define two sets: the set E of Entities and the set X of Channels. A channel valuation relates the channels (elements of the set X) to Messages exchanged over the channels. Because a Channel can be used to send multiple messages at any given moment, for every channel χ we define a set M_χ of messages currently sent over it. Furthermore, for each Entity ε we define a set Ψ_ε of Properties. A special property ν_ε encodes the current time of entity ε's Clock.

State is defined by: (i) a structural configuration formed by Entities and Channels; (ii) a behavioral configuration formed by Messages on each Channel, and valuation of Properties for each Entity; and (iii) the current time value of the Clock property for each Entity.

Properties are intended to encode the state of an Entity. To abstract from the concrete data types used to define the variable space we define a set of functions Φ. Each $\varphi \in \Phi$ is a function defined from the values of a tuple of Properties to a Boolean: $\forall\varphi \in \Phi: (\{\varphi: \Psi \times \Psi \times \cdots \rightarrow \mathbb{B}\})$. This allows for easy translation of UML specifications. For instance, if we want to model a UML Deployment Diagram specifying that a node would run a particular program *P*, we can define a function *run* and have it evaluate to *true* on the entity corresponding to the node (*run*(*P*)=*true*). The evaluation of the function set Φ over an entity ε is defined as $\Phi_\varepsilon \equiv \{(\varphi, \varphi_\varepsilon) : \varphi \in \Phi \wedge \varphi_\varepsilon \in \mathbb{B}\}$.

I can now define structural configuration as:

$$Conf_{Structural} \equiv (\mathrm{E}, \mathrm{X})$$

I define behavioral configuration as:

$$Conf_{Behavioral} \equiv \left(\{\mathrm{M}_{\chi} : \chi \in \mathrm{X}\}, \{\Phi_{\varepsilon} : \varepsilon \in \mathrm{E}\}, \{\nu_{\varepsilon} : \varepsilon \in \mathrm{E}\} \right)$$

The state is then defined as:

$$State \equiv (Conf_{Structural}, Conf_{Behavioral}) \in StateUniverse$$

Where $State$ is an element of the $StateUniverse$ set containing all possible states.

I can now define the concept of a run using streams: $Run \in StateUniverse^{\infty}$. The semantics of a system specification in this framework emerges as the set of admissible runs:

$$System \in \mathcal{P}(StateUniverse^{\infty})$$

ABSTRACT SPECIFICATION LANGUAGE

In this section I define the abstract language used to specify queries and constraints (and, therefore, systems). The benefits of defining this language are twofold. First, it provides an explicit context for mapping specifications (both composite and elementary) to systems in the semantic framework. Second, it provides a target language for the UML translation. The goal of the language is not to introduce a new textual syntax, and, therefore, we keep it simple by ignoring punctuation and other syntactic sugar necessary for a complete textual language definition.

I present the grammar of the language in a Backus-Naur Form (BNF) using production rules of the following form:

$$\langle \mathrm{N} \rangle ::= alt_1^{\langle \mathrm{N} \rangle} \parallel alt_2^{\langle \mathrm{N} \rangle} \parallel \cdots \parallel alt_n^{\langle \mathrm{N} \rangle}$$

Non-terminals are enclosed in angular brackets, the symbol $\parallel$ separates alternative productions, optional terms are enclosed in square brackets, and the notation $\{\mathrm{T}\}^*$ represents the repetition of term $\{\mathrm{T}\}$ for 0 or more times.

$$\begin{array}{ll}
\langle \text{ELEM-SPEC} \rangle & ::= \langle \text{QUERY} \rangle \langle \text{CONSTRAINT} \rangle \\
\langle \text{SPEC} \rangle & ::= \langle \text{SPEC} \rangle \langle \text{SPEC} \rangle \parallel \langle \text{ELEM-SPEC} \rangle \\
\langle \text{QUERY} \rangle & ::= \{\langle \text{MSG} \rangle\}^* \langle \text{ASSERTION} \rangle \\
\langle \text{CONSTRAINT} \rangle & ::= [\exists \parallel \neg\exists]\{[\neg\ ]\langle \text{MSG} \rangle\}^* \langle \text{ASSERTION} \rangle \\
\langle \text{MSG} \rangle & ::= \langle \text{MSGCONTENT} \rangle \langle \text{CHANNEL} \rangle \\
\langle \text{MSGCONTENT} \rangle & ::= \langle \text{MSGNAME} \rangle (\langle \text{SENDER} \rangle \langle \text{TIME} \rangle \{\langle \text{PARAM} \rangle\}^*) \\
\langle \mathit{ASSERTION} \rangle & ::= \langle \text{FUNCTION} \rangle (\{\langle \text{PROPERTY} \rangle\}^*) \parallel \\
 & \qquad \langle \text{UN-OPERATOR} \rangle \langle \text{ASSERTION} \rangle \parallel \\
 & \qquad \langle \text{ASSERTION} \rangle \langle \text{BIN-OPERATOR} \rangle \langle \text{ASSERTION} \rangle
\end{array}$$

Operator definitions are not part of this grammar. Instead, they will be introduced when necessary in the translation of UML. In particular, I express all unary operators with the non-terminal $\langle \text{UN-OPERATOR} \rangle$ and binary operators with $\langle \text{BIN-OPERATOR} \rangle$. $\langle \text{FUNCTION} \rangle$ is a Boolean formula from property names to Boolean. Using this grammar, we can specify a system based on the ontology using Queries and Constraints. In the next section, I define the semantics of such specifications.

Using the $\langle \text{CONSTRAINT} \rangle$ optional operators $\exists$ and $\neg\exists$, it is possible to affect the structure of the system. $\exists$ to creates new entities and channels, $\neg\exists$ removes them.

Time is addressed in this language as a property of entities. In particular, the notation $next(t)$ indicates the value of an entity clock in the first state where the value is greater than t. With $next$ I am able to reason about next states without constraining their occurrence to a particular time value. Moreover, the messages contain the $\langle\mathrm{SENDER}\rangle$ entity and the sending $\langle\mathrm{TIME}\rangle$ of the message in its parameter list.

SPECIFICATION LANGUAGE SEMANTICS

An elementary specification $\langle\mathrm{ELEM\text{-}SPEC}\rangle$ is captured in my abstract language by a tuple $\langle\mathrm{QUERY}\rangle$, $\langle\mathrm{CONSTRAINT}\rangle$. The goal of a specification is to define what runs are part of a system implementing such a specification. The $\langle\mathrm{QUERY}\rangle$ identifies what parts of the run the specification is constraining while the $\langle\mathrm{CONSTRAINT}\rangle$ specifies how those parts are constrained. A run that fulfills a pair of query and constraint is such that in all states following a state where the query is true the constraint is true. Therefore, an $\langle\mathrm{ELEM\text{-}SPEC}\rangle$ encodes a transition function between two states.

I define a $\langle\mathrm{QUERY}\rangle$ as a communication context selecting the states that follow a particular message interaction, and a Boolean formula over properties, which identifies states to constrain. A query thus addresses both the contents of channels (the channel history) and predicates over local data state of the relevant entities. I define the channel configuration Xc as:

$$\mathrm{Xc} \equiv \left(\,\mathrm{X}, \left\{\mathrm{M}_{\chi} \colon \chi \in \mathrm{X}\right\}\right)$$

This definition captures the part of a state S that specifies the channel configuration and the messages being exchanged in the given state. The semantics $[\![q]\!]$ of a $\langle\mathrm{QUERY}\rangle$ q is, therefore,

$$\forall q \in \langle \mathrm{QUERY} \rangle, [\![q]\!] \equiv (h \in \mathcal{P}(\, \mathrm{Xc}^*), a{:}\, \mathcal{P}(\Psi) \rightarrow \mathbb{B})$$

where Xc^* is a finite stream of channel configurations, the channel history $h \in \mathcal{P}(\,\mathrm{Xc}^*)$ is a set of such streams, and the assertion a is a function from a set of properties to Boolean values.

I define a helper function

$$query{:}\, (\mathcal{P}(\,\mathrm{X}^*), \mathcal{P}(\mathit{\Psi}) \rightarrow \mathbb{B}\,) \times StateUniverse^{\infty} \rightarrow \{\, \mathcal{P}(\mathrm{E}) \times \mathbb{N}_{\infty}\}$$

that, given a $\langle \mathrm{QUERY} \rangle$ semantics and a run, returns a set of tuples containing: (i) the indexes of the states where one of the message histories is matched and (ii) the corresponding set of entities for which the evaluation of the function is true. This helper function returns all states in the run where the next state must be constrained. It also returns the specific entities to be constrained in each state.

$\langle \mathrm{CONSTRAINT} \rangle$ is defined as a tuple of channel configurations, Boolean functions over properties, and one of the three quantifiers $\{\exists, \neg\exists, -\}$. Similar to the queries definition, we define the semantics of $\langle \mathrm{CONSTRAINT} \rangle$ as:

$$\forall c \in \langle \mathrm{CONSTRAINT} \rangle, [\![c]\!] \equiv (\mathrm{Xc}, a{:}\, \mathcal{P}(\Psi) \rightarrow \mathbb{B}, \{\exists, \neg\exists, -\})$$

We can define a helper function

$$constr{:}\, (\mathrm{Xc}, a{:}\, \mathcal{P}(\Psi) \rightarrow \mathbb{B}, \{\exists, \neg\exists, -\}) \times \{\, \mathcal{P}(\mathrm{E}) \times \mathbb{N}_{\infty}\} \times StateUniverse^{\infty} \rightarrow \mathbb{B}$$

where $constr$ takes as arguments a run, the result of a query operation, and the semantics of a constraint. This function returns true if the constraint is satisfied. To be satisfied, the channel configuration of the selected states must match the Xc specified by the constraint. Moreover, how the rest of the constraints is satisfied depends on the choice among the three quantifiers $\{\exists, \neg\exists, -\}$. If the chosen quantifier is $-$, the assertion s must evaluate to true in all entities selected. If the quantifier is $\exists$, the assertion s must evaluate to true in some entity not part of the selected ones. Finally, if the quantifier is $\neg\exists$ the selected entities must not be present in the selected states.

Now I can define a $\langle\mathrm{SPEC}\rangle$ in the semantic domain as a set of tuples of the form (query, constraint), and the System corresponding to the specification as the set of all possible runs that fulfill all such tuples (query, constraints) of the set.

Formally:

$$\langle\mathrm{SPEC}\rangle \subseteq \{(\langle\mathrm{QUERY}\rangle, \langle\mathrm{CONSTRAINT}\rangle) \colon \langle\mathrm{QUERY}\rangle, \langle\mathrm{CONSTRAINT}\rangle\}$$

$$[\![\langle\mathrm{SPEC}\rangle]\!] \equiv \{Run \colon Run \in StateUniverse^{\infty} \colon \forall s \in \langle\mathrm{SPEC}\rangle, \forall q \\ \in query(s.0, Run), constr(s.1, q, Run)\}$$

NOTION OF CONSISTENCY

In this chapter I am interested in defining dynamic consistency for real-time distributed systems. This is the reason why I have tailored the semantic framework to this domain rather than staying within the generality of the UML language metamodel. Given the semantic framework presented in the previous section, it is now straightforward to define dynamic consistency for models in this system class. First, I define horizontal consistency and then vertical consistency.

I define horizontal consistency as follows: a specification is horizontally consistent if the system it defines admits at least one run. A specification $\langle \mathrm{SPEC} \rangle$ is made of multiple views at the same level of abstraction (in my formalism this means multiple sets of query and constraint tuples).

Definition 1. A specification $\langle \mathrm{SPEC} \rangle$ such that $[\![\langle \mathrm{SPEC} \rangle]\!] \in \mathcal{P}(StateUniverse^{\infty})$ is horizontally consistent *iff* $[\![\langle \mathrm{SPEC} \rangle]\!] \neq \emptyset$.

This definition captures the idea that the specification is implementable. There are two possibilities for a system to fulfill this property. Either there are no contradictions in the specification, or the admissible runs do not match any query that defines inconsistent constraints. There is nothing wrong in using different perspectives to constrain the system behavior specified by other perspectives. However, if a perspective constrains the behavior of the system such that no run satisfying the specifications of that perspective is allowed in the final system, there can be a consistency problem. A stricter rule for horizontal consistency requires that the system has at least one run admissible for each perspective, meaning that there is at least one run satisfying some queries of each perspective specification.

Definition 2. A specification $\langle \mathrm{SPEC} \rangle$ such that $[\![\langle \mathrm{SPEC} \rangle]\!] \in \mathcal{P}(StateUniverse^{\infty})$ and $\langle \mathrm{SPEC} \rangle$ made of N specifications $\langle \mathrm{PERSP}_i \rangle$ called perspectives such that $[\![\langle \mathrm{SPEC} \rangle]\!] = \bigcap_{i \in N} [\![\langle \mathrm{PERSP}_i \rangle]\!]$ is horizontally consistent *iff*

$\forall \langle \mathrm{PERSP}_i \rangle, \exists Run \in [\![\langle \mathrm{SPEC} \rangle]\!] \wedge \exists s \in \langle \mathrm{PERSP}_i \rangle$ such that $query(s.0, Run) \neq \emptyset$.

A possible problem with my first definition of horizontal consistency is that there could be a system specification with no runs satisfying any query of the general specification. The consistency specification for such system is vacuously satisfied (i.e., runs are possible because selectors never match). The second definition solves this problem requiring that some runs matching the specification queries are present.

The two definitions of horizontal consistency given support two different usage scenarios. In fact, there are two main reasons to create a specification. First, I can be interested in constraining how the system works in a given scenario. The scenario I want to constraint must, therefore, be possible and the corresponding query must select some runs. Definition 2 caters to this type of usage. A different use case is when I want to specify recovery from some failure of the system. For example, I may identify that a given interaction can happen as a result of a failure even if the specification would not allow for it. In this case, the goal is to describe the detection and recovery from a given failure. Consistency Definition 1 caters to this usage scenario.

Vertical consistency is defined between two specifications at different level of abstraction. I define this consistency notion by a containment relation between runs. Given a more abstract specification SPEC_a and a more concrete specification SPEC_c vertical consistency is defined as follows: a concrete specification SPEC_c is consistent with an abstract specification SPEC_a if all runs allowed in the concrete system specification are also allowed in the abstract one. Moreover, the abstract system allows runs that the concrete system does not allow. This definition requires that the concrete systems admit a strict subset of the runs admitted by the abstract one.

> *Definition 3.* Two specifications $\langle \mathrm{SPEC}_a \rangle$ and $\langle \mathrm{SPEC}_c \rangle$, where the first is the abstract and the second the concrete specification, are vertically consistent *iff* $[\![\langle \mathrm{SPEC}_a \rangle]\!] \subset [\![\langle \mathrm{SPEC}_c \rangle]\!]$.

Given the definitions of $\langle \mathrm{SPEC} \rangle$ and $[\![\langle \mathrm{SPEC} \rangle]\!]$ of the previous section, I can now define a modularity theorem. I first observe that each specification has a set of tuples containing one query and one constraint. Therefore, each of these tuples defines a set of runs. From the definition of $[\![\langle \mathrm{SPEC} \rangle]\!]$, I infer a lemma asserting that the semantics of a complex $\langle \mathrm{SPEC} \rangle$ (i.e., formed by multiple tuples of query and constraint) is the intersection of the semantics of all the sub-specifications formed by single query/constraint tuples. The modularity theorem states that for any complex specification $\langle \mathrm{SPEC} \rangle$it is always possible to identify two sub-specifications such that the intersection of the runs permitted by the two contains exactly the runs permitted by the original specification. Moreover, the theorem states that, such sub-specifications can be obtained by taking two subsets of the tuples of the original specification, provided that all tuples of the original specification are in at least one of the two sub-specifications. The lemma and theorem are formally defined as follows.

Lemma 1. Given a specification $\langle \mathrm{SPEC} \rangle$

$$[\![\langle \mathrm{SPEC} \rangle]\!] = \bigcap_{\forall t \in \langle \mathrm{SPEC} \rangle} [\![\{t\}]\!]$$

Proof. Lemma 1 can be proven by observing that the definition of $[\![\langle \mathrm{SPEC} \rangle]\!]$ is such that if a specification contains a single query/constraint tuple t, the $\forall$ quantification in $\forall s \in \langle \mathrm{SPEC} \rangle$ return a single element. Therefore:

$$[\![\{t\}]\!] \equiv \{Run\colon Run \in StateUniverse^{\infty}\colon \forall q \in query(t.0, Run), constr(t.1, q, Run)\}$$

Given the definition of intersection: $\bigcap_{\forall s \in S} s = \{e\colon \forall s \in S, e \in s\}$, and replacing the specification of the semantics of a query/constraint tuple into the definition of intersection we obtain

$$\bigcap_{\forall t \in \langle \mathrm{SPEC} \rangle} [\![\{t\}]\!] = \Big\{e\colon \forall t \in \langle \mathrm{SPEC} \rangle, e \in \{Run\colon Run \in StateUniverse^{\infty}\colon \forall q \in query(t.0, Run), constr(t.1, q, Run)\}\Big\}$$

From this, by replacing e with the definition of Run we obtain

$$\bigcap_{\forall t \in \langle \mathrm{SPEC} \rangle} [\![\{t\}]\!] = \{Run\colon \forall t \in \langle \mathrm{SPEC} \rangle, Run \in StateUniverse^{\infty}\colon \forall q \in query(t.0, Run), constr(t.1, q, Run)\}$$

which is the definition of $[\![\langle \mathrm{SPEC} \rangle]\!]$. This proves the lemma. □

The Modularity theorem asserts that complex query/constraint specifications can be split into two simpler ones without losing information.

Theorem 1. Modularity. Given a specification $\langle \mathrm{SPEC} \rangle$ such that $|\langle \mathrm{SPEC} \rangle| > 1$ (i.e., the specification is complex),

$\forall \langle SPEC_1 \rangle, \langle SPEC_2 \rangle$ such that:

$$\langle SPEC_1 \rangle \subset \langle SPEC \rangle \wedge \langle SPEC_2 \rangle \subset \langle SPEC \rangle \wedge$$

$$|\langle SPEC_1 \rangle| > 0 \wedge |\langle SPEC_2 \rangle| > 0 \wedge$$

$$\langle SPEC_1 \rangle \cup \langle SPEC_2 \rangle = \langle SPEC \rangle$$

$$[\![\langle SPEC \rangle]\!] = [\![\langle SPEC_1 \rangle]\!] \cap [\![\langle SPEC_2 \rangle]\!]$$

The proof of Theorem 1 derives easily from Lemma 1. In fact, because the semantics of a specification is equivalent to the intersection of the semantics of all its constituent query and constraint tuples, we can use the commutative and associative properties of intersection to prove Theorem 1.

CONSISTENCY OF THE BART CASE STUDY

To show how the methodology outlined in this chapter applies to consistency checking in the context of the UML for real-time, I provide a translation from the UML and from its MARTE profile to the abstract language introduced. Translating the entire UML and MARTE metamodels is beyond the scope of this chapter. Instead, I chose a simple subset of the UML and MARTE that uses three graphical notations: Component Diagrams, Sequence Diagrams, and State Diagrams, which are used in the example of Figure 35.

Furthermore, I translate MARTE timed constraints as they are used in the example.

The translation from UML models to the query and constraint language assigns a precise semantics to each model. Several options for assigning semantics to each notation exist. Sequence Diagram, for instance, can be interpreted existentially (at least the specified behavior must be possible) or universally (precisely the specified behavior is required) [28] . The decision of interpreting the diagrams existentially or universally depends on what is the goal of the specification. For example, in a requirements document an Interaction can exemplify one of many possible scenarios and the existential interpretation would be correct. For real-time systems modeling I interpret sequence diagrams universally. All messages exchanged in the system must be represented in diagrams. This interpretation of sequence diagrams is a good choice for the application domain of the case study. In fact, one of the key uses of communication models in real-time systems is to analyze the network traffic and ensure that real-time constraints can be met. Therefore, a complete view of which messages are exchanged over the communication channels is necessary.

My translation strategy interprets every element of a UML graph as a query and constraint tuple. I introduce an operator to compose those elementary specifications – this closes the loop with the introduction of the abstract query/constraint syntax. For demonstration purposes, I introduce the parallel operator. This operator is applied between any two specifications in the translation and returns the specification containing all query and constraint tuples of the operand specifications.

$$\alpha, \beta \in \langle \mathrm{SPEC} \rangle$$

$$[\![\alpha \ \mathrm{PAR}\ \beta]\!] \equiv \{s : s \in \alpha \vee s \in \beta\}$$

Table 4 provides translation rules for some of the interesting model elements used in the example. The entire set of rules is beyond the scope of this chapter. Each rule provides a set of query/constraint tuples that can be composed in a specification using the parallel operator. To support the translations I define a small set of helper functions.

The function $toMSG()$ is used to convert two elements of the UML metamodel, MessageOccurencesSpecification and Triggers, into objects

Table 4. Translation rules for UML Metamodel elements

Name	Metamodel Element	Translation
UML:: BasicComponents:: Component	Figure 5	$Q : \{\}\ true$ $C : \exists\{\}\ EType = Component.name$
UML:: BasicInteractions:: MessageOccurence Specification	Figure 6	$MOS \equiv MessageOccurenceSpecification$ $t \equiv$ time of last message received in history $Q : ExtractHistory\ (\{MOS\})\ Clock > t$ $C : toMSG(\{MOS\})\ true$
UML:: BehaviorStateMachines:: Transition	Figure 7	$TR \equiv Transition, B \equiv TR.effect$ $s_1 = TR.source, s_2 = TR.target$ $Q : ExtractHistory(\{TR.trigger\})\ State$ $= s1 \wedge Clock = t$ $C : toMSG(B)\ State = s2 \wedge Clock = next(t)$
TimedConstraints:: TimedConstraint	Figure 8	$PR \equiv TimedInstantConstraint.specification$ $Q :$ $MsgFromObservations(PR.observation)$ $PropFromObservations(PR.observation)$ $C : \{\}\ evalVSL(PR) = true$

suitable for the abstract language. Informally, MessageOccurencesSpecifications represents on sequence diagram lifelines of the events related to message sending and receiving (plus execution of actions and other details not considered in my simplified model). The function $toMSG()$ expresses the translation from OccurencesSpecification elements of the UML metamodel to messages in the abstract language specification.

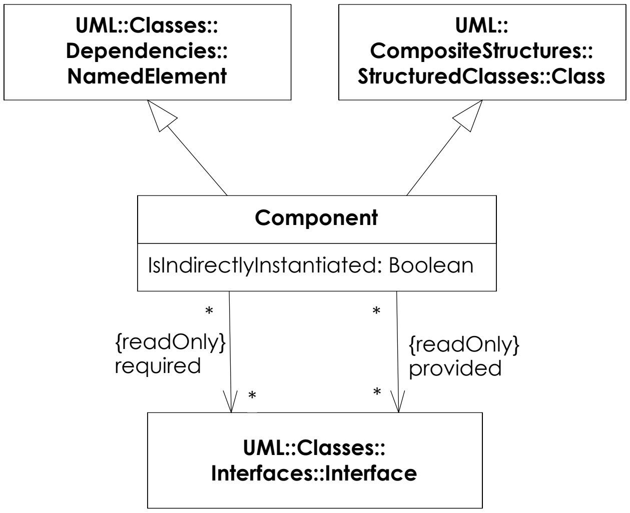

Figure 39. Subset of the UML Component metamodel

Similarly, the $ExtractHistory()$ function applied to a model element of type MessageOccurencesSpecification returns the sequence of messages that maps to the Events in the lifeline before the one defined by the given MessageOccurencesSpecification. Intuitively, this function returns the history necessary for a query to select the correct interactions before applying the constraint to match the message event defined by the MessageOccurencesSpecification model element. I do not describe the details of how this translation is performed because it is beyond the scope of this chapter. In fact, the UML metamodel is very complex. Extracting relations between events and specification elements in different diagrams often requires the exploration of a deep class hierarchy. For example, leveraging the metamodel (shown in Figure 40) to extract the history of events before a given message in a sequence diagram implies several steps. First, identifying the Lifeline the OccurrenceSpecification is covered by. Second, leveraging the fact that the set of events of a lifeline is ordered, extract all the OccurrenceSpecifications that precedes the given one. Third, scroll the ordered list of OccurrenceSpecifications in the history and navigate their event property to obtain the corresponding Events. Finally, using reflection, identify the events that are related to sending and receiving messages and use this information to generate the list of message specifications.

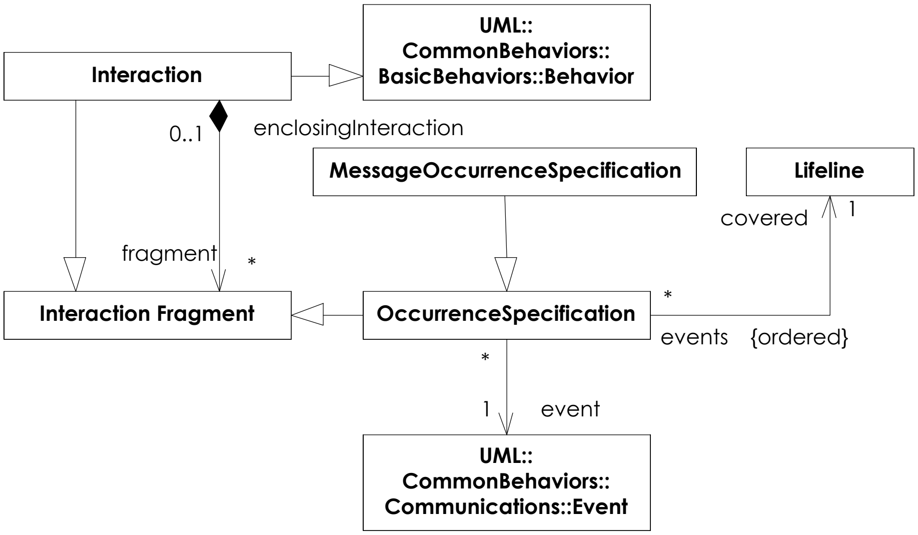

Figure 40. Subset of the UML Message metamodel

The four translations given in Table 4 map the elements of UML and MARTE metamodels depicted in Figure 39, Figure 40, Figure 41, and Figure 42 to query and constraint tuples. The first line of the table gives a translation for

Figure 39. This part of the metamodel defines UML components in a component diagram. The simple model in the figure captures the relation between Components and Interfaces which can be required or provided by the Component. My translation simply asserts that a specification of a component always imposes the existence of an entity with a property called EType and value equal to the component name in the UML diagram.

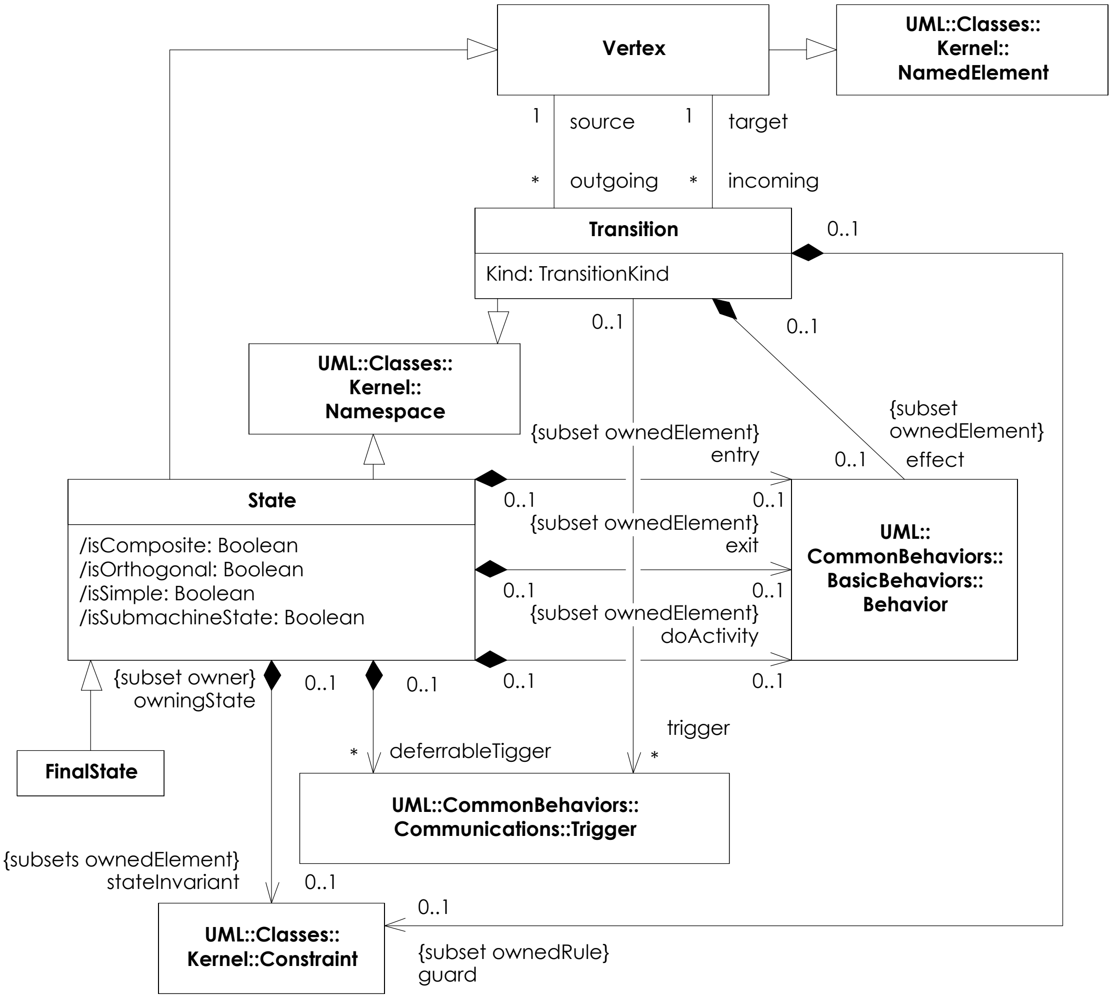

Figure 41. Subset of the UML Transition metamodel

The translation of line 2 of Table 4 defines constraints imposed by a MessageOccurrenceSpecification in a UML sequence diagram. The query extracts the message history before the given MessageOccurrenceSpecification. As already mentioned discussing ExtractHistory, this is not a trivial operation. Figure 40 presents the relevant subset of the UML model for sequence diagrams. Interactions are the type of behavior specified by this type of diagram. In particular, an Interaction is a type of Interaction Fragment that can be composed of other such fragments. Special types of Interaction Fragments are Occurrence Specifications which reference communication Events and Lifelines. An example of such specifications is MessageOccurrenceSpecifications which represents messages exchanged according to the interaction modeled. The constraint in my translation is the existence of the message corresponding to the MessageOccurrenceSpecification. This translation covers only events that are messages. Other types of events cause properties in some entity to be set and are not covered in the example.

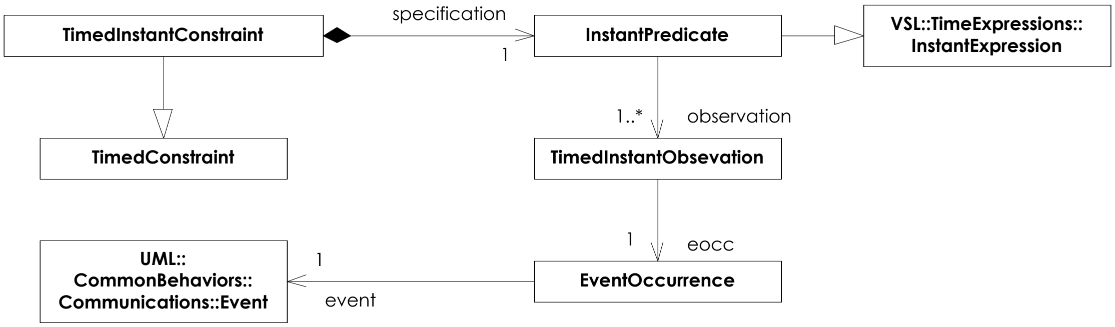

Figure 42. Subset of the MARTE TimedConstraints metamodel

The third line of Table 4 defines a translation for state machines transitions. To support this translation I introduce an entity property named State. Figure 41 depicts the relevant subset of the UML metamodel for state machine diagrams. According to the UML metamodel a Transition has a source and target Vertex, and State is a type of Vertex. A transition can be taken only if the guard constraint is true and, in this case, is taken when a given trigger occurs. Moreover, a transition can have an effect, which is a Behavior. An example of Behavior is the Interaction (depicted in Figure 40). In the case study, the translation is simplified to address just triggers and effects that are messages. The query part of Transition translation selects entities where the State variable coincides with the source state of the model. Other propositions in the query can be used to restrict the selection to only specific entities. In fact, state diagrams define the behavior of particular model elements. For example, the state diagram of Figure 35c only applies to the component Emergency Brake. In this case, the query should also limit the selection to states of the entity Emergency Brake. This can be achieved by adding to the query another clause that selects only entities of the correct type (i.e., $EType = \text{"Emergency Brake"}$). The other part of the query limits the selection to states where the trigger message is present. The constraint simply forces the next state of the selected entities to have the target state in the State property.

Finally the last line of Table 4 defines the translation for MARTE TimedInstantConstraint. Figure 42 (which is adapted from [73]) shows the relevant MARTE metamodel. A TimedInstantConstraint has a specification that is a predicate over a set of observations (TimedInstantObservations). Each observation identifies an event occurrence. EventOccurrences relates MARTE observations to UML Event elements. The translation of InstantPredicates must somehow interpret the Value Specification Language (VSL) instant expression defined in MARTE's VSL language. To this end, the translation uses an $evalVSL$ Boolean function that evaluates a VSL expression. Moreover, it uses two functions, $MsgFromObservations$ and $PropFromObservations$, to obtain the messages and properties that correspond to the events referred to by the observation of a predicate. By observing Figure 42 it is evident that identifying the event associated to a Timed Constraint is complex. Obtaining messages and properties from events requires a good understanding of the UML metamodel and the exploration of many nested relations. While complex, those functions can be implemented in a program. The translation then selects the correct messages and entities in the query part of the specification and asserts that the specification in VSL evaluates to true in the constraint.

I can now show how to detect inconsistency with this query and constraint framework using the example of Figure 35. Thanks to the modularity theorem defined in the previous section, I can split each specification into simpler specifications. In particular, because the intersection of the specifications obtained with the modularity theorem is equivalent to the original specification, inconsistency can be proved by just translating a subset of the model and proving that such subset is inconsistent (no runs allowed).

For example, I translate the model element of Figure 35b that represents the sending of an Ack message from the Emergency Brake to the Train Controller. This translation, according to Table 4, would look like

$$Q : \{\quad\}^{*}\{(\text{Commands Received}(\text{Train Controller},t_1))\}\ EType$$
$$= \text{Emergency Brake} \wedge Clock > t_1$$

$$C : \{(\text{Ack}(\text{Emergency Brake}, \text{t}_{\text{ack}}))\}\ EType = \text{Emergency Brake}$$

The translation of Figure 35c transition triggered by the Commands Received message is

$$Q : \{(\text{Commands Received}(\text{Train Controller},t'))\}\ EType = \text{Emergency Brake} \wedge State$$
$$= \text{Wait Commands} \wedge Clock = t'' \wedge t'' > t'$$

$$C : \{\}\ EType = \text{Emergency Brake} \wedge State = \text{Reset Timer} \wedge Clock = next(t'')$$

and

$$Q : \{\}\ EType = \text{Emergency Brake} \wedge State = \text{Reset Timer} \wedge Clock = t'''$$

$$C : \{\}\ EType = \text{Emergency Brake} \wedge State = \text{Wait Commands} \wedge Clock = next(t''')$$

Let us analyze the type of runs that satisfy the translation of the sequence diagram. We can observe that, for a run to satisfy the specification, if in a state there is a Commands Received message received by the Emergency Brake component, it must send an Ack message. In the sequence diagram translation we do not specify if there is some other action local to the Emergency Brake. In fact, the simplified translation for sequence diagrams deals only with messages sent and received, not local actions. So the message can be returned immediately (next state) or after some local transitions (that is the meaning of {}*, which represents a sequence of zero or more states where the channel is empty). The specification, however, is clear in identifying that no other messages are sent or received by Emergency Brake before returning a message.

The translation of the state diagram of Figure 35c triggers a transition from Wait Commands to Reset Timer when the Commands Received message is received by Emergency Brake. We can identify the inconsistency by observing that all runs that fulfill our translation for Figure 35c never send the Ack message. The intersection of sets of runs identified by the two specifications is, therefore, empty. Thus the two specifications are inconsistent.

In my formalism, I can prove consistency by composing query and constraint tuples and identifying contradictions. In particular, I chose to encode queries and constraints using Propositional Linear Temporal Logic formulae (LTL)[74]. The encoding changes for each definition of consistency. I can then prove that a system is consistent according to the chosen definition by proving that the LTL formula that encodes such definition is satisfiable. This proof can be automated by means of a satisfiability (SAT) solver for LTL formulas. Examples of algorithms for assessing satisfiability of propositional LTL formulas and tools implementing them can be found in [75], [76].

In this chapter I do not give a complete translation for all definitions. Instead, I use the example of inconsistent specification from Figure 35b and Figure 35c and encode the query and constraint specification to prove inconsistency according to Definition 1. For each tuple of query (Q) and constraint (C) I create the implication $Q \Rightarrow \mathbf{X}C$, where $\mathbf{X}$ is the next operator in LTL. If I can find a set of variables that satisfies the disjunction of all these implications the specification is consistent according to Definition 1.

I capture this in the following theorem.

Theorem 2. Consistency D1 Satisfiability. Given a specification $\langle\mathrm{SPEC}\rangle$, $\langle\mathrm{SPEC}\rangle$ is consistent according to consistency Definition 1 if and only if the expression $\bigwedge_{\forall(Q,C)\in\langle\mathrm{SPEC}\rangle} Q \Rightarrow \mathbf{X}C$ is satisfiable.

Proof. In this theorem I assume that messages in the channels history are encoded using appropriate variables and nested temporal operators. The exact discussion of how to encode these messages is beyond the scope of this chapter. The proof of Theorem 2 follows from the definition of $[\![\langle\mathrm{SPEC}\rangle]\!]$. In fact, the semantics of $\langle\mathrm{SPEC}\rangle$ is defined as the set of runs that satisfy all query/constraint tuples. I encode each tuple as an implication in LTL that is true if a run satisfies it. The conjunction of all the LTL implications is true only if a run satisfies all of them. If the formula in Theorem 2 is not satisfiable, there exists no run that can satisfy all implications at the same time, thus $[\![\langle\mathrm{SPEC}\rangle]\!]$ is empty. On the other hand, if the expression is satisfiable, there exists at least one run that can satisfy all queries and constraints, thus $[\![\langle\mathrm{SPEC}\rangle]\!]$ is non-empty. This proves Theorem 2. □

Let's now consider how Theorem 1 and Theorem 2 apply to the example. From Theorem 1 I know that to prove inconsistency I am not required to compose all the queries and constraints. Instead I can split the specification into two subspecifications and the original one will be equivalent to the intersection of the new specifications. Then, if I can prove that one of the two is empty we know that the full specification must be inconsistent. I chose to compose only the specifications of Figure 35b and Figure 35c. I prove that this subspecification is inconsistent (i.e., has an empty set of runs) and from Theorem 1 I obtain that the full specification is also inconsistent.

Consider all runs satisfying the translation of the transition from Wait Commands to Reset Timer in Figure 35c. I identify all runs with a trigger message Commands Received and a transition in the entity Emergency Brake with State changing from “Wait Commands” to “Reset Timer”.

Because the constraint of this specification is the query of the translation for the transition from Reset Timer to Wait Commands in Figure 35c, if I compose the two specifications I obtain all runs where Emergency Brake reacts to a Commands Received by changing two states without sending any message.

I can now compose the current system into the translation of the Ack message specification in Figure 35b and discover that one of the next states of the runs selected must send an Ack message before any other messages is received by Emergency Brake. However, from state Wait Commands the system can exit only if the trigger message Commands Received is received. Therefore, by exploring specification tuples I can argue that because the

Clock time greater than t_1, at which the Ack message must be sent by the sequence diagram constraint, is finite and the specification of the state diagram does not allow any transition that sends messages without receiving anything from the state that it enters after the trigger message, there is a contradiction, and, therefore, the specifications are inconsistent.

The translation of the state diagram specification in Figure 35d limited to the transition from Reset Timer to Wait Commands is:

$$Q : \{\}\ EType = \text{"Emergency Brake"} \wedge State = \text{"Reset Timer"} \wedge Clock = t'''$$

$$C :\ \{(\mathrm{Ack}(\text{Emergency Brake}, \mathrm{t_{ack}}))\}\ EType = \text{"Emergency Brake"} \wedge State \\ = \text{"Wait Commands"} \wedge Clock = next(t''')$$

With this change the composition of the specifications for the state machine identifies a sequence of states initiated by the trigger message Commands Received that ends with the sending of an Ack message. In the composition with the specification from Figure 35b the state where the Ack message is sent must happen at a time $next\big(next(t'')\big)$ that is greater than t_1. Therefore, there is no contradiction between constraints, and, thus, no inconsistency. To prove that the entire specification is consistent all remaining elements must be translated. While this process it long and error prone if performed by hand, the existence of automated tools for solving the satisfiability problem makes it a viable solution.

DISCUSSION

In this chapter, I demonstrated an approach for consistency management based on queries and constraints on a reduced subset of the UML and its MARTE profile. The goal of this work was to demonstrate the feasibility of the approach by providing a case study where I was able to identify inconsistencies in UML models. Thus, the translation I gave assigned a semantics only to a subset of the modeling elements defined in the UML and MARTE. However, even using this reduced subset I was still able to detect and formally verify the inconsistency between models of the BART case study including timing constraints. Because the given translation binds query and constraint tuples to single entities in the UML metamodel, an extension to the full language definition of UML 2.0 and its different profiles is straightforward, albeit complex. Such an extension requires giving a precise semantics for each diagram, and therefore, deciding how each syntactic element of each diagram contributes to its semantics.

Tailoring this consistency notion to a particular target domain (real-time distributed systems in this case) may, at first, seem limiting. However, I believe that a completely general definition of consistency for a general purpose language such as the UML ultimately limits the applicability of consistency checking to very abstract models, or to purely structural notions of consistency (without taking the notion of behavior into account). This claim is supported by a thorough analysis of related work performed in [63].

Different decisions in how to interpret diagrams can lead to different translations. For example, I decided to interpret sequence diagrams universally regarding the messages exchanged. Each message represented in the diagram is exchanged and messages not represented are not. In contrast, state transitions are not part of my translation of sequence diagrams. This is why I set the Clock in the query of row 2 of Table 4 as greater than the time the previous message was sent without setting a specific interval. This is equivalent to a commitment to eventually have a state in which the constraint is true.

The definitions of horizontal and vertical consistency given seem adequate for the domain of real-time systems. However, when a richer subset of the UML or other languages, such as Orca, will be translated and more experience acquired in verifying their consistency, I see potential for reevaluating the definitions. One possible area of concern with the current definition arises when I allow side effects between the queries and constraints of multiple diagrams, in other words, non-local constraints. In this case I could change the definition of horizontal consistency, for instance, to yield inconsistency if the majority of the queries do not match.

The benefit of moving from the abstract domain of UML metamodels to the query and constraint abstract language is that the translation rules define the semantics and implicitly also the consistency rules. I can then avoid enumerating a long list of consistency rules and obtaining a very simple definition of consistency.

With this approach I have converted the problem of detecting the consistency of graphs based on the UML metamodel to verifying emptiness of sets. The sets are defined by logical formulae, each defining the effect of one model element on the system runs. The composition of specifications is defined by set intersection. Additionally I have presented a modularity theorem (Theorem 1) that enables reasoning on separate subsets of the query/constraint specifications. This setup is amenable to translation into propositional linear temporal logic and supports use of many automatic formal verification tools, such as SAT solvers. I have also provided Theorem 2 that affirms the equivalence of proving that an LTL expression is satisfiable with horizontal consistency of the corresponding specification.

I can now evaluate the query and constraint approach proposed by identifying how it addresses the 12 requirements identified in Table 3.

R1. Support inconsistent models. My approach addresses this requirement by not forcing the user to remove inconsistencies. Models that are inconsistent can be identified by identifying the tuples that are in contradiction. More modeling elements can be added and more contradictions detected before the system is made consistent.

R2. Automatic inconsistency discovery. Inconsistencies are discovered by hand in this example. The goal was to show the complexity of the problem and a possible solution. It is possible, however, to automate translation (which leverages the UML metamodel used by all UML modeling tools) and detection to discover inconsistencies automatically. Furthermore, inconsistencies can be tracked by identifying the subset of specifications that are in contradiction.

R3. Support inconsistency resolution. The support to resolve inconsistencies is provided by the ability to identify a small subset of the specification that is sufficient to prove the inconsistency (this property stems from the Modularity Theorem).

R4. Support multiple modeling languages. The query and constraint approach supports multiple languages by creating different translation rules from the UML metamodel to the abstract target language. It could also support languages that are not the UML as long as they are based on a metamodel and a translation is provided.

R5. Support different levels of abstraction. I have identified different consistency rules and translation rules to support different levels of abstraction.

R6. Support extensions. I demonstrated the support for extensions of UML providing a translation rule for the MARTE profile.

R7. Support Horizontal consistency. I provided two horizontal consistency definitions.

R8. Support Vertical consistency. I provided one definition for vertical consistency.

R9. Support Static consistency. This approach supports static consistency by querying entity properties and channel messages and by constraining them.

R10. Support Dynamic consistency. The approach supports dynamic consistency by constraining the properties of different states in admissible runs. Leveraging LTL logic and the Clock it is possible to set constraints on consecutive states or future states.

R11. Provide tool support. While I haven't provided any tool support for this approach, I have demonstrated that a translation of the consistency problem to satisfiability of LTL formulae exists (Theorem 2). The translation from the UML to another domain can be automated and because queries and constraints can be encoded in LTL, existing SAT solvers for this logic can be leveraged to automate the verification.

R12. Address scalability. Thanks to the modularity theorem my approach does not require reasoning about the entire model to identify inconsistencies. This makes it applicable to large models. However, depending on how the different specifications are interconnected, to ensure that no inconsistency exists, it may be necessary to compose a large number of tuples, which could slow down the identification of inconsistencies on some models.

From this requirement analysis I conclude that the query and constraint approach proposed is a step towards a more comprehensive consistency management approach for UML models. This approach can also be extended to incorporate interaction models based on Orca or aspect oriented MSCs. However, more work is required to implement tools to automate the approach and experiment with the effective scalability of such tools by testing them on large industrial-scale system models.

SUMMARY

This chapter covered the last key requirement for a complete model based approach: model consistency. The approach to model consistency presented here addresses not only the modeling techniques introduced in this thesis, but also many different languages. For this reason the case study and models in this chapter are not based on models of interactions and crosscutting concerns according to the Rich Service pattern. Instead the case study uses different graphical languages from the UML profile MARTE. In fact, supporting multiple languages is a key feature for a successful consistency management technique.

ACKNOWLEDGEMENT

This chapter, in part, is a reprint of material as appeared in E. Farcas, I. Krueger, and M. Menarini, “Consistency Management of UML Model,” Real-time Simulation Technologies: Principles, Methodologies, and Applications, K. Popovici and P. J. Mosterman (Eds.), ch. 12, p. 38, CRC Press, 2012. The dissertation author was the primary investigator and author of the text used in this chapter.

CHAPTER 7

RELATED WORK

This chapter presents work related to the different areas relevant to the research presented in this thesis. This survey analyzes prior work related to model-based engineering (MBE) for software-intensive systems both in the enterprise and embedded domains. In particular, the survey covers service-oriented techniques, aspect-oriented modeling, architectures used in embedded and enterprise systems, and techniques for quality assurance and reliability of service-oriented systems.

In summary, the survey shows important advances towards systematic engineering processes for these domains. However, it reveals the lack of comprehensive and seamless integration of requirements, architecture, implementation, and verification and validation models across all development activities. In particular, crosscutting concerns are not satisfactorily addressed in the engineering process.

REQUIREMENTS MODELS

Requirements engineering is arguably one of the most important and least-well understood [77] development activities. Errors made during the activities that pertain to requirements analysis and management are hard to detect and costly to fix as time progresses through the development process. Requirements need to articulate values of the stakeholders of the system under consideration. Stakeholders include (and are not limited to) the customer who commissions and accepts the system, regulatory bodies, marketing and production entities, suppliers, integrators, developers, architects and maintainers, and end-users. There are multiple ways of classifying requirements, such as business, product, and process requirements, or with different criteria: functional and non-functional requirements (e.g., usability [78], performance and efficiency [79–81], reliability [82], [83], and interoperability [84], [85]). Consequently, models, techniques and tools for documenting and managing requirements necessarily need to be able to reflect the various different views that each stakeholder group brings to the table.

Modeling plays an important role in all requirement engineering activities, serving as a common interface to domain analysis, requirements elicitation, specification, assessment, documentation, and evolution. The choice of modeling notations is often a tradeoff between readability and powerful reasoning techniques: natural language is very flexible but it is often an expression of subjective reasoning [86–88]; applied / semi-formal models (e.g., entity-relationship diagrams, UML diagrams, structured analysis) typically have a graphical representation, which is very useful when communicating with stakeholders and for simulations; and formal notations (e.g., KAOS, i*, SCR, RML) capture precise semantics, which supports rich verification techniques. To better support different application domains UML profiles, such as the UML Profile for Schedulability, Performance, and Time [89] have been proposed.

KAOS [88], [90] and i* [91] focus on goal-based hierarchies for system objectives, actors and actions that they are capable of, and iterative refinement of goals using AND/OR decompositions. The resulting models rely on temporal logic for verification of agents' plans, fulfillment of commitments, and other system properties.

There are several similar approaches for structured analysis, including Structured Analysis and Design Technique (SADT) [92], Structured Analysis and System Specification (SASS) [38], Structured System Analysis (SSA) [93], and Structured Requirements Definition (SRD) [94]. For instance, SADT provides a data model linked through consistency rules with a model for operations, supports the formalization of the declarative part of the system (through activity diagrams), but uses natural language for the requirements. SSA adds data access diagrams, whereas SRD introduces the idea of building separate models for each perspective and then merging them.

Software Cost Reduction (SCR) [95], [96] method uses a tabular notation for specifying requirements, a formal Finite State Machine (FSM) based model, and modeling constructs such as modes, terms, conditions, variables, and events to describe the system and its behavior. The Four-Variable Model [97–99] extends the method to entire systems by including critical aspects of timing and accuracy as mathematical relations on monitored and con-trolled variables. CoRE [100] goes further by providing structuring mechanisms for variables (e.g., aggregation or generalization), models (e.g., and/or decomposition), and tables (e.g., refinement relationships).

Requirements State Machine Language (RSML) [101–103] uses both tabular and graphical notations borrowed from Statecharts. The high-level state machine model decouples the specification of requirements from design aspects and enables formal analysis of the entire system for correctness and robustness.

ARCHITECTURES

Component based. An important example of component-based frameworks is Common Object Request Broker Architecture (CORBA) [104]. CORBA is made of a set of specifications, which standardize how to invoke remote objects. It is a rich specification, which covers the infrastructure needed to create robust distributed applications. In particular, a specialization of the CORBA specification targeting embedded systems (CORBA/e) is being currently finalized [105]. Real-time CORBA [106] is designed for applications with real-time requirements; it provides interfaces and policies that allow applications to configure and manage processor, network, and memory resources. Open-source implementations of Real- Time CORBA ORBs (Object Request Broker), such as ZEN [107] and TAO [108], have shown that it is possible to provide QoS guarantees in middleware. RT-CORBA has also been used to evaluate performance in run-time evaluation of inter-action models [36].

Another choice for real-time distributed software is the Honeywell's MetaH [109] specification language. It describes how different elements of a system such as software components, hardware, and communication subsystems are integrated to form the final application. A suite of visual tools help the MetaH developer to add components, edit them, define the scheduling, partition the application, and analyze the timing behavior. The toolset includes formal verification, schedulability analysis, and reliability analysis based on Markov chains. Components may be annotated in the graphical editor with real-time properties, such as execution time and failure modes. MetaH code generator produces glue code that includes such properties.

Initially prototyped in the context of autonomous helicopter flight control, Giotto [22] is a time-triggered high-level programming language that expresses the reactivity of the application related to the external environment. A Giotto program defines several operational modes, each one invoking a set of periodic tasks and allowing mode changes at predefined points in time. Giotto is a real-time extension to traditional programming languages, and has similarities with architecture description languages (ADLs) [110]. In particular, Giotto is similar to MetaH [109], the difference being that Giotto is time-triggered, platform independent, and does not restrict the implementation to a particular scheduling scheme. Giotto introduced the concept of Logical Execution Time (LET), which abstracts from the physical execution time and, thereby, from both the execution platform and the communication topology.

Moreover, for single-processor embedded control systems, the Giotto methodology was integrated with Simulink [19] to allow streamlined operations from the design to the implementation phase [23].

The Timing Definition Language (TDL) [111], a successor of Giotto, is a high-level description language for specifying the explicit timing requirements of an application, which may be constructed out of several components (called modules). A TDL module communicates with the physical environment through sensors and actuators, performs computation in tasks, and defines different operational modes that can be changed at run-time. Similar to Giotto, TDL is based on the Logical Execution Time abstraction. In addition, TDL allows modularization of applications [112], ECU consolidation, and the transparent distribution [113] of multi-mode real-time components. TDL provides a complete tool chain for transparent distribution with a run-time system [114] that enforces LET semantics and automatic generation of communications schedule [115] and glue code [116].

With transparent distribution [113], the observable behavior of a TDL application is exactly the same at run-time, no matter if all components are executed on a single node or if they are distributed across multiple nodes. Thus, TDL components can be developed without having the execution on a potentially distributed platform in mind, as the distribution is visible only for the system integrator who specifies the mapping of components to computation nodes.

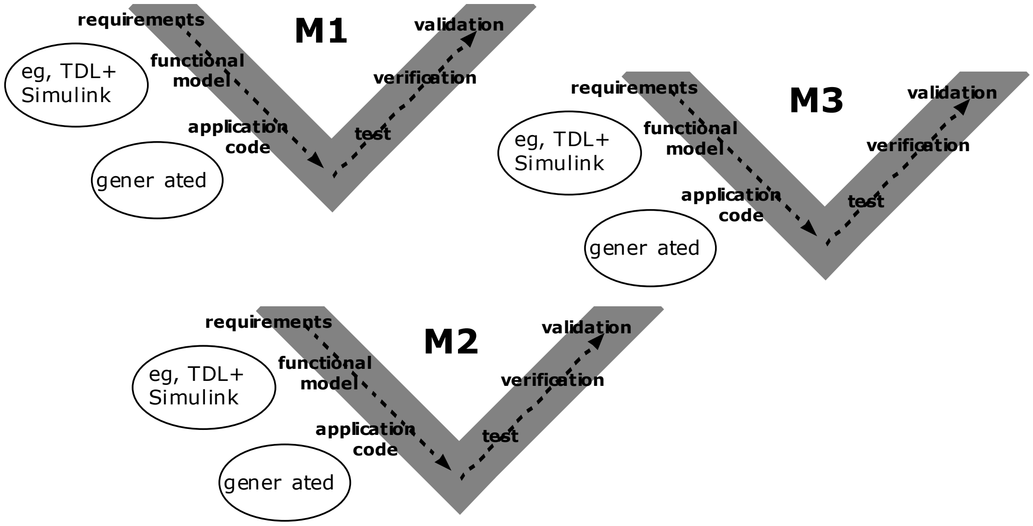

Figure 43. TDL V-Cluster-Life-Cycle

Particularly attractive for the automotive domain, TDL modules can be developed independently of each other, by different suppliers. Each module has its own V-Life-Cycle as seen in Figure 43 for modules M1, M2 and M3; therefore, the system is developed in a V-Cluster-Life-Cycle [112]. In the automotive domain, the functional model is often developed in Simulink. This fact led to proposals to integrate TDL into Simulink (e.g., [117]).

SOA. A different architectural style is the service-oriented architecture. Successfully applied originally in the telecommunications domain a service has become a common term in many application domains, especially in the context of web services [118]. So far, however, services have been used mainly as an implementation concept.

In the telecommunication domain, for example, the notion of service is expressed by the term *feature,* which is used to describe self-contained pieces of functionality and is used to structure components' interfaces [119]. Feature interactions [120] should not lead to inconsistent behaviors. This service notion focuses on the local interface of each component, but the interplay between services is considered only afterwards. Consequently, current definitions of services limit themselves to syntactical definition for operations clients can invoke on a service; however, a behavioral model of allowed service inter-actions is needed. This limited view of the service scope is at the origin of the absence of a service as modeling entity in common modeling languages such as UML [15], [16] and SysML [121].

The aim of service-oriented architectures is to make services first-class elements of the system development process, starting from early models of the requirements all the way to the system design, construction, and verification. To support this approach, comprehensive service theories have been proposed. They provide a semantic framework to interpret the service models [122].

Implementations of service-oriented frameworks exploit the lessons learned from component based frameworks, such as CORBA. Compared to CORBA, SOA frameworks aim to reduce the coupling between components and simplify the implementation of the framework. For example, in the Internet domain, web services architecture [118] leverage Internet standard protocols to provide an interoperable, loosely coupled framework to implement distributed applications. The core Web Service technologies are Web Services Description Language (WSDL) interfaces [123], the Universal Description, Discovery, and Integration (UDDI) [124] standard for service discovery, and the Simple Object Access Protocol (SOAP) [125].

Service-oriented approaches focus on the composition of basic services to provide higher level functions. The main web service based standards to compose services are Web Services Business Process Execution Language (WS-BPEL) [126] and Web Services Choreography Description Language (WS-CDL) [127]. WS-BPEL is an OASIS standard that defines a workflow language to specify a centralized composition of web services. A BPEL engine executes a BPEL XML document by calling web services according to the workflow captured by it. A different approach is the one taken by WS-CDL, which captures a global view of the composition. The specification defines the role of each service involved in the composition; all parties involved are responsible to implement their part of the composition.

Autosar [128], an automotive-specific framework that lever-ages some of the ideas from SOA, tackles the integration problem by specifying appropriate standards for interfaces among different components; thus, software modules provided by different suppliers will be easier to integrate. System functions modeling and function testing is a major concern. Ultimately, Autosar aims at application-centric development of automotive software by decoupling functions from the underlying platform through virtualization. A similar approach, [129] de-scribes execution of SOA applications on a virtual network that is late-bound to a physical network, essentially creating a SOA overlay network.

EMBEDDED SYSTEMS IMPLEMENTATION TECHNIQUES

Traditionally, embedded applications are developed either by using a classical sequential language such as C/C++ and Java, or by using a parallel language, that is, a real-time programming language such as Ada [130], CSP[131], and Real-Time Java[132]. Sequential languages lack concurrency, whereas parallel languages support concurrency and communication as first-class concepts.

Synchronous languages. As reactive systems continuously interact with their environment, the speed of the interaction is dictated by the environment and not by the computing system. The synchronous model is based on the assumption that all computation or communication activities take no time. Thus, synchronous languages define reactions as atomic. The implementation may approximate synchrony by reacting to an event before another event appears. The compiler verifies synchrony, reactivity, and determinism. The compiler checks synchrony based on the maximum input frequency and the worst-case execution times obtained from static code analysis. For reactivity, the compiler must prove the absence of infinite cycles. Determinism is related to the problem of causality, which can be easily verified by requiring dependencies to be acyclic. However, the programmers may specify static cyclic dependencies that lead to deterministic programs if actually there are no cycles at run-time.

Synchronous languages are classified under two categories: imperative and declarative languages. The imperative languages such as Esterel [133], Statecharts [134], Argos [135], and SyncCharts [136] have explicit control flows and are appropriate for control-intensive applications such as bus inter-faces, controllers, supervision of complex systems, and real-time process control. The declarative languages - for example, Lustre [137], [138] and Signal [139] - use a data-flow model and are appropriate for data-intensive applications such as signal processing and steady process-control applications.

The Esterel language is based on the semantics of the finite-state Mealy machine [140], which ensures a deterministic behavior. Esterel is an imperative language that provides high-level, modular constructs that lead to a structure of reactive programs. An Esterel program is defined by a collection of modules and a main module; a module may be instantiated within another module and exports its data declarations to the parent module. Esterel implements communication through signals. Modules must have a defined interface to be able to communicate. A signal is available only at the instant when it was produced, and signals are instantaneously broadcasted - any module can react to the signal. Esterel pro-vides a set of primitives for expressing concurrency, sequencing, communication, and preemption.

Regarding other imperative languages, Statecharts [134] has a graphical formalism and is not fully synchronous. It is used to model complex discrete controllers that need several modes of operation and a switching mechanism; Statecharts extend the concept of finite-state machines with hierarchy, parallel composition, and broadcast communication. Argos [135] simplifies the formalism of Statecharts and provides full synchrony, and SyncCharts [136] extend Argos to yield the power of Esterel.

Lustre [137], [138] is a declarative language that supports only the data-flow systems that can be implemented as bounded automata-like programs in the sense of Esterel. In Lustre, any variable and expression represents a flow, which is a pair of a possible infinite sequence of values and a clock. Lustre has data operators and temporal operators; synchrony in Lustre means that all operators respond instantaneously to their input. Data operators (e.g., arithmetic, relational, conditional opera-tors, and imported functions described for example in C) operate pointwise on the sequences of values of their operands. Temporal operators (e.g., pre, follow by, when, and current) manipulate flows [138]. A Lustre program has a cyclic behavior defined by a basic clock. The clock of any flow may be smaller than the basic clock. In addition, the Signal language [139] allows for creating faster flows.

Tools and Platform. Model-based development techniques are useful only if supported by a powerful set of tools. Tool support is critical to enable the creation and exploitation of models. Models are used in the automotive and avionics domains to capture sys-tem architecture, control loops, electronic components behavior, mechanical characteristics, safety properties, etc. Models are then exploited to generate implementations, automate the generation and execution of test cases, perform formal verifications on important systems properties, and support the con-figuration of product lines, just to mention few applications.

Both in the academic and the industrial world, tools and underlying theories have been developed to fulfill each of those functions. An example of an industrial tool used to create control models in the automotive domain is Matlab/Simulink. This tool is used to generate code implementing the models; moreover, the Simulink Design Verifier tool is able to create test cases to verify the correctness of the models. Matlab/Simulink is targeted to modeling continuous systems, but it can also model discrete controllers by means of Stateflow; furthermore, TTPMatlink models distributed systems by including the time-triggered communication intro the Simulink model.

Other tools provide standard model-checking techniques to verify the correctness of some model. For example, the SPIN model checker [141] provides a modeling language called Promela and is able to verify complex protocols with concurrency by exploiting a partial order reduction algorithm. In the academic domain, SPIN has been used to verify safety properties of interaction models for automotive systems in [142]. Similarly, [143] explores model checking of interaction-based specifications; it uses temporal logic to give a semantics for live sequence charts (LSCs), reducing the verification of LSCs to the model checking of temporal logic formulae. Another ex-ample is [144], which addresses model checking of message sequence charts.

TDL offers a VisualTDL Editor [145], which through a graphical user interface enables the developer to visually model TDL components and their timing requirements. It can be used as a stand-alone tool or as an integrated Simulink editor similar to Stateflow. The VisualTDL Editor translates the visual representation into TDL code, and the functionality code is automatically generated with the Simulink add-on Real-TimeWorkshop Embedded Coder (RTWEC). For each TDL module, the compiler generates the so-called embedded code (E-Code [146]), which describes the timing constraints of the module, and the glue code (E-Code is platform independent [116], but it requires the binding with user-defined functions and a corresponding run-time environment). The run-time system consists of the E-Machine for E-Code execution, a TDL Scheduler, and a TDLComm layer responsible with transferring the information over the network [147]. The run-time system interacts with the underlying real-time operating system via a Run-Time Resource Management (RTRM) layer [148]. This run-time environment ensures that the LET semantics are met in both single-node and distributed systems, and that the execution follows strict hard real-time guarantees.

Without the benefit of LET, other techniques are required to create robust programs from code that is not fail proof. In [149] for example, a technique to obtain failure-tolerant systems by composing intolerant systems is proposed. Systems are com-posed with two types of components – detectors and correctors. The paper proves that that these are sufficient to create fail-safe and non-masking tolerant systems, respectively.

[150] extends this approach to non-fusion-closed systems; it introduces history variables as needed to maintain the required information. Another related approach to automate the implementation of fail-safe systems is presented in [151]. It proposes a technique to synthesize fault-tolerant

programs from computation tree logic (CTL) specifications. It allows generating not only the program behavior but also detectors and correctors to ensure that the system is resilient to failures. The drawback of this technique is that it is subject to the state explosion problem.

The synchronous approach is used in modeling tools such as Scade [152], [153], which supports the development of real-time controllers on non-distributed platforms or distributed platforms like the Timed-Triggered Architecture [154]. The Scade suite supports the design of continuous dataflows (based on Lustre [138]) with discrete parts realized by a state-machine editor (based on Esterel [133]). The computational models are compatible by transforming values and signals [152]. The Scade Suite is used by Airbus for the development of the critical software embedded in several aircrafts [155].

SERVICE-ORIENTED RELIABILITY

One direction of the existing work on fault tolerance for services is to focus on the perspective of the service provider. Corresponding approaches propose techniques for increasing the reliability and availability of a service with respect to its clients. For instance, a number of attempts have been made to apply techniques from Fault Tolerant CORBA (FT-CORBA) [156] to the Web Services domain. Fault Tolerant SOAP (FT-SOAP) [157] follows the service approach in fault tolerant CORBA and provides transparent fault tolerance by upgrading SOAP with additional components to support fault detection and replication management. FT-SOAP extends WSDL to inform the clients of the replica information. FT-Web [40], on the other hand, follows the interception approach from FT-CORBA and proposes an infrastructure where a dispatcher acts as a proxy for client requests, and sends them to service replicas in parallel. [158] applies path monitoring techniques by adding unique identification numbers to requests and performing distributed logging. Failure detection is based on centralized statistical analysis of the logged paths. Ref. [159] uses a probabilistic model to detect the faulty components and attempts to mitigate by restarting the components, or by rebooting the servers hosting them.

Other approaches to fault tolerant services concentrate on centralized service composition or orchestration. Here the goal is to build reliable service composition from unreliable services. Approaches such as [46] and [42] propose techniques to use Business Process Execution Language for Web Services (BPEL4WS) [53] compensation and fault handlers to achieve fault tolerant composition. Due to the nature of BPEL4WS fault handlers, detection is only possible on a single-invocation basis as opposed to more complex interaction based detection. Common forward recovery policies such as ignoring, retrying, substitution, and parallel execution of alternatives can be supported in these techniques. [160] proposes a connector that is used to invoke the composed services, thus acting as the fault-containment element. Assertions based on SOAP exceptions can be declared in the connector, and if not fulfilled, recovery policies can be activated.

A number of transaction-based approaches also exist. In [47], both service providers and the orchestrator explicitly declare their transactional semantics and requirements in an XML-based language. A middleware component acts as an intermediary service and harmonizes these transactional requirements. [161] suggests mining the logs of the service workflow in order to extract a model for the real workflow of the service based on transactions and improves recovery of the transactions where possible.

A number of fault tolerance approaches also exist in the Grid services domain. Ref. [39] suggests a primary-backup mechanism based on notifications for the Grid Services. Grid Workflow [162] proposes a workflow description language that allows users to define recovery strategies for cases where a task fails to complete, in the Grid Services domain.

While all the approaches presented here helps in increasing the reliability of service oriented systems, they consider the service as a single invocation/response pair. Thus their applicability is limited to this simple interaction pattern. Because I consider services as generic interactions that can exhibit more complex pattern I must provide an improved solution. The solution to the reliability problem I present in this thesis is based on the rich service pattern (presented in Chapter 1) and on interceptors that monitor complex interactions. When a failure in the execution of the complex interactions it detected my approach can apply all forward mitigation strategies described above. I present this technique in Chapter 2.

QUALITY ASSURANCE

A key element in both automotive and enterprise systems is the requirement for high reliability. The Reliability of a system measures the ability of it to perform its intended service [163]. A *failure* occurs when the system deviates from its intended behavior. In particular, a system enters in an erroneous state when its state is such that it can lead to a *failure*; the difference between the valid state and the erroneous one is the *error*. A *fault* is the cause that leads to erroneous state of the system [164].

From a theoretical point of view, the resilience to failures of a program can be analyzed by identifying what is the effect of a fault on the program result. Different categories of tolerance have been identified: masking tolerance, non-masking tolerance, and fail-safe tolerance [165].

A key question for assessing the quality of airplanes and other vehicles is how safe they are. In fact, verification only shows that a system performs according to the given specification. However, the question of how safe is the specification cannot be answered by just looking to the quality of the software or of the system in isolation. For example, for a car that is parked in a garage a failure of the Anti-lock Braking System (ABS) braking system is not as unsafe as for a car that is speeding on an icy highway. In [41], for example, it is discussed the importance of embedding the software quality assurance process in a system-wide quality process. To assess the safety of a vehicle, it is not possible to limit the analysis to the quality and reliability of the code.

Both industries engage in extensive system quality processes such as different flavors of Failure Modes and Effects Analysis (FMEA) [166], [167] and Fault Tree Analysis (FTA) [168] to identify and mange possible faults. Extensions of FMEA and FTA for software systems have been proposed; Ref. [169], for instance, discusses software FMEA techniques, and Ref. [170] discusses Software FTA. Nevertheless, these approaches need to be complemented by an end-to-end interaction view.

In the embedded world, different strategies and standards have been proposed to increase the reliability of systems. For example, Fault Tolerant CORBA [156] extends the CORBA framework to provide failover, redundancy, detection and recovery from failures. FT-CORBA approach aims to be transparent to the application level and embed the support for replication, request retry, redirection to alternative servers.

Attempts to support fail-safe computation for web services via replication have been proposed in FT-SOAP [157] and FT-Web [40]. The first approach requires a change in the SOAP standard, whereas the second does not.

In general, we can distinguish two strategies to recover from failures: backward recovery and forward recovery. Backward recovery techniques aim to return the system to a previous consistent state. These include transaction-based approaches such as split transactions [171]. In particular, research has focused in avoiding blocking commits using protocols such as the ones presented in Ref. [172] and the Paxos consensus protocol [173]. Transaction-based approaches for the web services domain are presented in [174] and [175]. Forward recovery, on the other hand, aims to move the system to a new state that is correct. WS-BPEL [126], for example, includes provisions for detecting faults (services that do not respond as expected) and fault handlers to perform recovery activities. In this context, [42] uses the WS-BPEL language to specify the service composition and a rule-based system to recover from errors.

Verifying timing constraints. In the real-time systems encountered in the automotive and avionics domains, a critical constraint is the *deadline*, that is, the time instant before which a computational activity must deliver its results. Deadline requirements are typically verified with schedulability analysis based on analytical theory. An alternative is to provide a model of the real-time application and verify it with formal methods. There is also ongoing research [176], [177] in integrating scheduling theory into formal methods. Schedulability analysis ([178], [179]) checks for a certain scheduling algorithm whether all activities will meet their timing constraints, even in the worst-case behavior of the system; it does not check whether the timing constraints are appropriate for the requirements of the application – instead, formal methods can verify functional and high-level timing requirements such as sampling and actuating times, data avail-ability, or data consistency.

The common approaches for deriving models are to use formalism such as process algebra [131], [180] and Petri nets or to model the system as a state-transition graph [181]. The algebraic approach can be verified by proof theoretic approaches, such as theorem provers [182], [183]. A state-transition model can be verified by model checking [184], or reachability analysis. Modeling timing attributes is done by including clock variables or a tick process in the untimed models, which perform state changes by transitions or by time steps. Timed formalisms include timed transition systems [185], timed automata [186], and real-time temporal logic [187]. There are several tools available for verifying real-time systems (Kronos [188], Uppaal [189], Verus [190] and hybrid systems (HyTech [191]). Nevertheless, it is a challenge to build models that represent complex systems, are compositional, include timing constraints, and model the system scheduler [192]. For example, Ref. [193] addresses the problem of obtaining a timed model from the application software composed with the timing constraints induced by both the environment and the execution platform. The methodology was implemented in the Taxys tool [194], which can be used only for real-time systems programmed in Esterel extended with C functions.

ASPECT-ORIENTED MODELING

The increasing success of Aspect-Oriented Programming techniques (AOP) in software development has led to the idea of extending AOP to the modeling and design levels in the software development lifecycle. Researchers have proposed various Aspect-Oriented Modeling (AOM) techniques and have adopted them to address the need for separation of concerns in complex systems. A large body of work on AOM has been focused on structural models, while fewer approaches also consider behavioral models. Song et al. [195] proposes template class and sequence diagrams to specify aspect models. Additional binding information is needed to instantiate the template aspect models and composition is facilitated by using composition directives. Whittle et al. [55] use Interaction Pattern Specifications (IPS) to capture aspectual scenarios and use binding information provided by the designer to instantiate these models. They propose three special composition operators to compose the aspectual models with the non-aspectual scenarios. In this paper we use template MSCs (Aspect MSC) to capture aspect models. Bindings to other MSCs can be explicit or can be automatically performed by matching regular expressions. The only operator defined is the Match operator.

Jezequel et al. [196] propose a semantic-based composition of aspects in high-level MSCs. They use a basic MSC to capture the pointcut and one to capture the advice of the aspect. Whenever the pointcut is matched the advice MSC replaces the pointcut MSC. By leveraging the semantics of the Match operator, we define the pointcut and the advice in the same template MSC, reducing the overhead of redefining the common messages in multiple places.

A number of approaches have introduced aspects into state machine diagrams. Whittle et al. [55] propose a new aspect composition language for UML state diagrams. They can support complex pointcuts and specify the pointcut and the advice in the same diagram. However, I believe that specifying and composing the scenarios at the sequence diagram level is important since it allows the designer to view and debug all the resulting composed interactions at the MSC level.

In [197], [198], composition patterns have been proposed as a solution to capture crosscutting concerns as patterns. In composition patterns, template classes and sequence diagrams are used to define the crosscutting concerns and pattern binding is used to compose the pattern with concrete model elements. However, pointcut definition for composing the crosscutting concerns with the concrete model elements are less flexible than approaches such as [55] and the one presented in Chapters 4 and 5.

ACKNOWLEDGEMENT

This chapter, in part, is a reprint of material as appeared in 5 papers: 1) in C. Farcas, E. Farcas, I. H. Krueger, and M. Menarini, "Addressing the Integration Challenge for Avionics and Automotive Systems - From Components to Rich Services," in The Proceedings of the IEEE Special Issue on Aerospace and Automotive Software, K. V. Prasad (Ed.), vol. 98, no. 4. IEEE, Apr. 2010, pp. 562-583.

2) V. Ermagan, I. H. Krüger, and M. Menarini, "A Fault Tolerance Approach for Enterprise Applications," in Proceedings of the IEEE International Conference on Services Computing (SCC). Jul. 2008.

3) V. Ermagan, I. H. Krüger, and M. Menarini, "Aspect Oriented Modeling Approach to Define Routing in Enterprise Service Bus Architectures," in MiSE

'08: Proceedings of the 2008 international workshop on Models in software engineering, Leipzig, Germany. New York, NY, USA: ACM, May 2008, pp. 15-20.

4) E. Farcas, I. Krueger, and M. Menarini, "Consistency Management of UML Model," Real-time Simulation Technologies: Principles, Methodologies, and Applications, K. Popovici and P. J. Mosterman (Eds.), ch. 12, p. 38, CRC Press, 2012.

5) E. Farcas, I. Krueger, and M. Menarini, "Modeling with UML and Its Real-Time Profiles," Real-time Simulation Technologies: Principles, Methodologies, and Applications, K. Popovici and P. J. Mosterman (Eds.), ch. 5, p. 36, CRC Press, 2012

The dissertation author was the primary investigator and author of the text used in this chapter.

© 2010 IEEE. Reprinted, with permission, from C. Farcas, E. Farcas, I. H. Krueger, and M. Menarini, Addressing the Integration Challenge for Avionics and Automotive Systems - From Components to Rich Services, The Proceedings of the IEEE Special Issue on Aerospace and Automotive Software, and 04/2010

© 2008 IEEE. Reprinted, with permission, from V. Ermagan, I. H. Krüger, and M. Menarini, A Fault Tolerance Approach for Enterprise Applications, Proceedings of the IEEE International Conference on Services Computing (SCC), and 07/2008

This work is based on an earlier work: Aspect Oriented Modeling Approach to Define Routing in Enterprise Service Bus Architectures, in

Proceedings of the 2008 international workshop on Models in software engineering (MiSE '08), © ACM, 2008. http://doi.acm.org/10.1145/1370731.1370735

Copyright 2012 From Real-time Simulation Technologies: Principles, Methodologies, and Applications by K. Popovici and P. J. Mosterman. Reproduced by permission of Taylor and Francis Group, LLC, a division of Informa plc.

CHAPTER 8

CONCLUSIONS AND OUTLOOK

In this thesis I presented techniques to support model-based development of service-oriented systems. The key differentiator of a service-oriented architecture versus component-oriented ones is focus on managing the interaction patterns between the different parts of the system. Thus, I introduced modeling techniques for specification and composition of interaction patterns.

I applied my model-based approach to two different domains: embedded and enterprise systems. While different in many ways, these domains have to common challenge of integrating different subsystems that are distributed and, often, developed by different teams. I found that the introduction of interaction models, along with techniques for the composition of these models, is beneficial in both domains.

In particular, I identified that the main problem in service composition arise when composing crosscutting concerns. When decomposing a software system, engineers chose the main concerns of the application and usually decompose the system according to such concerns. Unfortunately, additional concerns always exist and they often cross-cut the services created according to the chosen decomposition.

A fundamental problem with current, “flat”, service-oriented architectures is that they do not easily support composition of crosscutting concerns. To address the limitations of current service architectures the Rich Service pattern, presented in Chapter 1, introduces hierarchical decomposition, message routing, and 2 different types of services: Application and Infrastructure services. In my thesis, I proposed aspect-oriented modeling languages for interactions as a tool for modeling Rich Services.

I demonstrated the Rich Service approach and different models of interactions with aspects on three case studies. The central locking system (CLS) case study, which describes the locking and unlocking of a car, covers the embedded system domain. The CoCoME case study, which presents a distributed managements system for a company having multiple stores, covers the enterprise systems domain. Finally the BART case study, which describes the automated control system for an train network, has elements from both the embedded systems domain and the enterprise systems one.

In this thesis I focused my attention of one particular type of crosscutting concern, failure management. For this problem I introduced techniques to model system properties and improve reliability by generating monitors, applying mitigation techniques, and formally verify which system properties are maintained if failures occur.

Two key contributions of this thesis are: a comprehensive interaction language that supports aspects, called Orca, and a technique to assess consistency of different models of the same system developed using different modeling languages. Orca provides a simple language that can be used to formalize the different notations I use to model Rich Services. The consistency management approach, on the other hand, guarantee that my technique can be combined with existing modeling approach, an required feature for using my approach in real development projects.

This thesis advances the state of the art in model-based system development; in particular, in the field of service-oriented architecture. More work is still needed in developing a proper tool chain and in integrating the Orca language with other modeling techniques. However, in this thesis I have collected enough evidence to validate the claim that: the use of Rich Services and aspect-oriented modeling techniques is a viable avenue for improving the development of both embedded and enterprise systems.

REFERENCES

[1] K. V. Prasad, M. Broy, and I. Krüger, "Scanning Advances in Aerospace & Automobile Software Technology," *Proceedings of the IEEE*, vol. 98, no. 4, pp. 510–514, 2010.

[2] *AOM '09: Proceedings of the 13th workshop on Aspect-oriented modeling*. New York, NY, USA: ACM, 2009.

[3] A. Charfi, H. Müller, and M. Mezini, "Aspect-Oriented Business Process Modeling with AO4BPMN," in *Modelling Foundations and Applications*, vol. 6138, T. Kühne, B. Selic, M.-P. Gervais, and F. Terrier, Eds. Springer Berlin / Heidelberg, 2010, pp. 48–61.

[4] A. Charfi and M. Mezini, "Aspect-Oriented Web Service Composition with AO4BPEL," in *Web Services*, vol. 3250, L.-J. Zhang and M. Jeckle, Eds. Springer Berlin / Heidelberg, 2004, pp. 168–182.

[5] J. Misra and W. Cook, "Computation Orchestration," *Software and Systems Modeling*, vol. 6, no. 1, pp. 83–110, 2007.

[6] T. Elrad, R. E. Filman, and A. Bader, "Aspect-oriented programming: Introduction," *Commun. ACM*, vol. 44, no. 10, pp. 29–32, Oct. 2001.

[7] R. Chitchyan, A. Rashid, P. Rayson, and R. Waters, "Semantics-based composition for aspect-oriented requirements engineering," in *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, 2007, pp. 36–48.

[8] R. B. France, I. Ray, G. Georg, and S. Ghosh, "Aspect-oriented approach to early design modelling," *IEE Proceedings - Software*, vol. 151, no. 4, pp. 173–186, Aug. 2004.

[9] D. Stein, S. Hanenberg, and R. Unland, "Expressing different conceptual models of join point selections in aspect-oriented design," in *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, 2006, pp. 15–26.

[10] Object Management Group, *Model Driven Architecture (MDA) v1.0.1, omg/03-06-01*. OMG, 2003.

[11] G. Nicolescu and P. Mosterman, *Model-Based Design for Embedded Systems*. CRC Press.

[12] R. France and B. Rumpe, "Model-driven Development of Complex Software: A Research Roadmap," in *2007 Future of Software Engineering*, Washington, DC, USA, 2007, pp. 37–54.

[13] K. Balasubramanian, A. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema, "Developing applications using model-driven design environments," *Computer*, vol. 39, no. 2, pp. 33–40, 2006.

[14] J. Sztipanovits and G. Karsai, "Model-Integrated Computing," *Computer*, vol. 30, no. 4, pp. 110–111, 1997.

[15] Object Management Group, *Unified Modeling Language (OMG UML), Superstructure, Version 2.3, formal/2010-05-05*. OMG, 2010.

[16] Object Management Group, *Unified Modeling Language (OMG UML), Infrastructure, Version 2.3, formal/2010-05-03*. OMG, 2010.

[17] J. Warmer and A. Kleppe, *The object constraint language: precise modeling with UML*. Addison-Wesley Longman Publishing Co., 1998.

[18] The MathWorks, "MATLAB." [Online]. Available: http://www.mathworks.com/products/matlab/. [Accessed: 08-Jun-2012].

[19] The Mathworks, "Simulink - Simulation and Model-Based Design." [Online]. Available: http://www.mathworks.com/products/simulink/. [Accessed: 06-Apr-2012].

[20] ETAS, "ASCET." [Online]. Available: http://www.etas.com/en/products/ascet_software_products.php. [Accessed: 08-Jun-2012].

[21] S. Gérard, H. Espinoza, F. Terrier, and B. Selic, "Modeling Languages for Real-Time and Embedded Systems - Requirements and Standards-Based Solutions," in *Model-Based Engineering of Embedded Real-Time Systems*, vol. 6100, Springer Berlin / Heidelberg, 2011, pp. 129–154.

[22] B. Horowitz, "Giotto: A Time-Triggered Language for Embedded Programming," University of California, Berkeley, 2003.

[23] T. A. Henzinger, C. M. Kirsch, M. A. A. Sanvido, and W. Pree, "From Control Models to Real-Time Code using Giotto," *IEEE Control Syst.Mag.*, vol. 23, no. 1, pp. 50–64, Feb. 2003.

[24] M. Arrott, B. Demchak, V. Ermagan, C. Farcas, E. Farcas, I. H. Krüger, and M. Menarini, "Rich Services: The Integration Piece of the SOA Puzzle," in *Proceedings of the IEEE International Conference on Web Services (ICWS)*, 2007, pp. 176–183.

[25] MuleSoft, "Mule ESB - Open Source ESB Community." [Online]. Available: http://www.mulesoft.org/. [Accessed: 19-Jun-2012].

[26] I. H. Krüger, M. Meisinger, M. Menarini, and S. Pasco, “Rapid Systems of Systems Integration - Combining an Architecture-Centric Approach with Enterprise Service Bus Infrastructure,” in *Proceedings of the 2006 IEEE International Conference on Information Reuse and Integration (IRI’06)*, 2006, p. 51—56.

[27] V. Ermagan, C. Farcas, E. Farcas, I. H. Krüger, and M. Menarini, “A Service-Oriented Blueprint for COTS Integration: the Hidden Part of the Iceberg,” in *Proceedings of the ICSE Second International Workshop on Incorporating COTS Software into Software Systems: Tools and Techniques (IWICSS’07)*, 2007, p. 10.

[28] I. H. Krüger, “Distributed System Design with Message Sequence Charts,” Fakultät für Informatik, Technischen Universität München, 2000.

[29] M. Broy and I. Krüger, “Interaction Interfaces - Towards a scientific foundation of a methodological usage of Message Sequence Charts,” in *Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM’98)*, 1998, pp. 2–13.

[30] B. Demchak, C. Farcas, E. Farcas, and I. H. Krüger, “The Treasure Map for Rich Services,” in *Proceedings of the 2007 IEEE International Conference on Information Reuse and Integration (IRI)*, 2007, p. 400—405.

[31] B. W. Boehm, “A spiral model of software development and enhancement,” *Computer*, vol. 21, no. 5, pp. 61–72, 1988.

[32] National Institute of Standards and Technology (NIST), “National Information Assurance Partnership (NIAP) - The Common Criteria Evaluation and Validation Scheme.” [Online]. Available: http://www.niap-ccevs.org/cc-scheme/. [Accessed: 19-Jun-2012].

[33] T. G. K. Venkatesh Prasad, “The Case for Modeling Security, Privacy, Usability and Reliability (SPUR),” in *Automotive Software*, 2006.

[34] I. H. Krüger and R. Mathew, “Component Synthesis from Service Specifications,” in *Scenarios: Models, Transformations and Tools, International Workshop, Dagstuhl Castle, Germany, September 7-12, 2003, Revised Selected Papers*, vol. 3466, S. Leue and T. J. Systä, Eds. Springer Berlin / Heidelberg, 2005, p. 255—277.

[35] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003.

[36] J. Ahluwalia, I. H. Krüger, M. Meisinger, and W. Phillips, “Model-Based Run-Time Monitoring of End-to-End Deadlines,” in *Proceedings of the 5th*

ACM international conference on Embedded Software (EMSOFT'05), New York, NY, USA, 2005, p. 100—109.

[37] V. Ermagan, I. Krüger, and M. Menarini, "Model-Based Failure Management for Distributed Reactive Systems," in *Composition of Embedded Systems. Scientific and Industrial Issues*, 2007, vol. 4888/2007, pp. 53–74.

[38] T. DeMarco, "Structured analysis and system specification," in *Classics in software engineering*, Upper Saddle River, NJ, USA: Yourdon Press, 1979, pp. 409–424.

[39] X. Zhang, D. Zagorodnov, M. Hiltunen, K. Marzullo, and R. D. Schlichting, "Fault-tolerant grid services using primary-backup: feasibility and performance," in *Cluster Computing, 2004 IEEE International Conference on*, 2004, pp. 105 – 114.

[40] G. T. Santos, L. C. Lung, and C. Montez, "FTWeb: a fault tolerant infrastructure for Web services," in *Ninth IEEE International EDOC Enterprise Computing Conference*, 2005, pp. 95–105.

[41] N. Leveson, *SafeWare : system safety and computers*. Reading, Mass.: Addison-Wesley, 1995.

[42] A. Liu, Q. Li, L. Huang, and M. Xiao, "A Declarative Approach to Enhancing the Reliability of BPEL Processes," in *Web Services, 2007. ICWS 2007. IEEE International Conference on*, 2007, pp. 272–279.

[43] B. Demchak, V. Ermagan, E. Farcas, T.-J. Huang, I. Krüger, and M. Menarini, "A Rich Services Approach to CoCoME," in *The Common Component Modeling Example: Comparing Software Component Models*, A. Rausch, R. Reussner, R. Mirandola, and F. Plasil, Eds. Berlin/Heidelberg: Springer-Verlag, 2008, p. 85—115.

[44] A. Erradi, P. Maheshwari, and V. Tosic, "Recovery Policies for Enhancing Web Services Reliability," in *Web Services, 2006. ICWS '06. International Conference on*, 2006, pp. 189–196.

[45] I. Krüger, R. Grosu, P. Scholz, and M. Broy, "From MSCs to Statecharts," in *Proceedings of the IFIP WG10.3/WG10.5 international workshop on Distributed and Parallel Embedded Systems (DIPES'98)*, 1999, pp. 61–71.

[46] G. Dobson, "Using WS-BPEL to Implement Software Fault Tolerance for Web Services," in *Software Engineering and Advanced Applications, 2006. SEAA '06. 32nd EUROMICRO Conference on*, 2006, pp. 126 –133.

[47] T. Mikalsen, S. Tai, and I. Rouvellou, “Transactional attitudes: Reliable composition of autonomous Web services,” in *Workshop on Dependable Middleware-based Systems*, 2002.

[48] “Apache ServiceMix ESB.” [Online]. Available: http://servicemix.apache.org. [Accessed: 26-Jan-2009].

[49] I. H. Krüger, D. Gupta, R. Mathew, P. Moorthy, W. Phillips, S. Rittmann, and J. Ahluwalia, “Towards a Process and Tool-Chain for Service-Oriented Automotive Software Engineering,” in *Proceedings of the ICSE 2004 Workshop on Software Engineering for Automotive Systems (SEAS)*, 2004.

[50] V. Ermagan, T.-J. Huang, I. Krüger, M. Meisinger, M. Menarini, and P. Moorthy, “Towards Tool Support for Service-Oriented Development of Embedded Automotive Systems,” in *Proceedings of the Dagstuhl Workshop on Model-Based Development of Embedded Systems (MBEES’07), Informatik-Bericht 2007-01*, 2007.

[51] B. Finkbeiner and I. Krüger, “Using Message Sequence Charts for Component-Based Formal Verification,” in *Specification and Verification of Component-Based Systems Workshop at OOPSLA 2001*, 2001.

[52] V. Ermagan, I. Krueger, M. Menarini, J.-I. Mizutani, K. Oguchi, and D. Weir, “Towards Model-Based Failure-Management for Automotive Software,” in *Proceedings of the ICSE Fourth International Workshop on Software Engineering for Automotive Systems (SEAS’07)*, 2007, p. 8.

[53] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, and S. Thatte, *Business Process Execution Language for Web Services (BPEL4WS) 1.1*. 2003.

[54] A. Banerji, C. Bartolini, D. Beringer, V. Chopella, K. Govindarajan, A. Karp, H. Kuno, M. Lemon, G. Pogossiants, S. Sharma, and S. Williams, *Web Services Conversation Language (WSCL) 1.0*. W3C, 2002.

[55] J. Whittle, A. Moreira, J. Araújo, P. K. Jayaraman, A. M. Elkhodary, and R. Rabbi, “An Expressive Aspect Composition Language for UML State Diagrams,” in *MoDELS*, 2007, pp. 514–528.

[56] M. Broy and K. Stolen, *Specification and Development of Interactive Systems Focus on Streams, Interfaces, and Refinement*. Springer, 2001.

[57] R. E. Filman and D. P. Friedman, “Aspect-Oriented Programming is Quantification and Obliviousness,” *Workshop on Advanced Separation of Concerns*, vol. 2000, 2000.

[58] I. H. Krüger, M. Meisinger, and M. Menarini, “Interaction-Based Runtime Verification for Systems of Systems Integration,” *J Logic Computation*, vol. 20, no. 3, pp. 725–742, Jun. 2010.

[59] D. Kitchin, A. Quark, W. Cook, and J. Misra, “The Orc Programming Language,” in *Formal Techniques for Distributed Systems*, vol. 5522, D. Lee, A. Lopes, and A. Poetzsch-Heffter, Eds. Springer Berlin / Heidelberg, 2009, pp. 1–25.

[60] I. Wehrman, D. Kitchin, W. R. Cook, and J. Misra, “Properties of the Timed Operational and Denotational Semantics of Orc,” University of Texas at Austin, Department of Computer Sciences, Technical Report TR-07-65, 2007.

[61] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton,Jr., “N degrees of separation: multi-dimensional separation of concerns,” in *Proceedings of the 21st international conference on Software engineering*, New York, NY, USA, 1999, pp. 107–119.

[62] I. H. Krüger and M. Menarini, “Queries and Constraints: A Comprehensive Semantic Model for UML2,” in *Models in Software Engineering*, vol. 4364, T. Kühne, Ed. Springer-Verlag Berlin Heidelberg, 2007, p. 327—328.

[63] E. Farcas, I. Krueger, and M. Menarini, “Chapter 12: Consistency Management of UML Model,” in *Real-time Simulation Technologies: Principles, Methodologies, and Applications*, K. Popovici and P. J. Mosterman, Eds. CRC Press, 2012, pp. 289–328.

[64] V. Winter, F. Kordon, and M. Lemoine, “The BART Case Study,” in *Formal Methods for Embedded Distributed Systems*, Springer US, 2004, pp. 3–22.

[65] ITU, *Message sequence charts (MSC). ITU-TS Recommendation Z.120*. 1996.

[66] A. C. W. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh, “Inconsistency handling in multiperspective specifications,” *Software Engineering, IEEE Transactions on*, vol. 20, no. 8, pp. 569–578, 1994.

[67] S. Easterbrook and B. Nuseibeh, “Using ViewPoints for inconsistency management,” *Software Engineering Journal*, vol. 11, no. 1, pp. 31–43, 1996.

[68] S. Easterbrook, “Learning from inconsistency,” in *Proceedings of the 8th International Workshop on Software Specification and Design (IWSSD’96)*, 1996, pp. 136–140.

[69] S. Easterbrook, J. Callahan, and V. Wiels, "V&V through inconsistency tracking and analysis," in *Software Specification and Design, 1998. Proceedings. Ninth International Workshop on*, 1998, pp. 43–49.

[70] W. Hongyuan, F. Tie, Z. Jiachen, and Z. Ke, "Consistency check between behaviour models," in *Communications and Information Technology, 2005. ISCIT 2005. IEEE International Symposium on*, 2005, vol. 1, pp. 486–489.

[71] G. Engels, J. M. Küster, R. Heckel, and L. Groenewegen, "A methodology for specifying and analyzing consistency of object-oriented behavioral models," *SIGSOFT Softw. Eng. Notes*, vol. 26, no. 5, pp. 186–195, 2001.

[72] H. Malgouyres and G. Motet, "A UML model consistency verification approach based on meta-modeling formalization," in *Proceedings of the 2006 ACM symposium on Applied computing*, Dijon, France, 2006, pp. 1804–1809.

[73] Object Management Group, *UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) Version 1.0, formal/2009-11-02*. OMG, 2009.

[74] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (FOCS 1977)*, 1977, pp. 46–57.

[75] V. Goranko, A. Kyrilov, and D. Shkatov, "Tableau Tool for Testing Satisfiability in LTL: Implementation and Experimental Analysis," *Electron.Notes Theor.Comput.Sci.*, vol. 262, pp. 113–125, 2010.

[76] K. Y. Rozier and M. Y. Vardi, "LTL satisfiability checking," in *Proceedings of the 14th international SPIN conference on Model checking software*, Berlin, Heidelberg, 2007, vol. Berlin, Germany, pp. 149–167.

[77] M. Shaw, "Prospects for an engineering discipline of software," *IEEE Software*, vol. 7, no. 6, pp. 15–24, Nov. 1990.

[78] J. L. Bennett, "Managing to meet usability requirements: Establishing and meeting software development goals," *Visual Display Terminals: Usability Issues and Health Concerns*, pp. 161–184, 1984.

[79] B. Nixon, "Representing and using performance requirements during the development of information systems," in *Advances in Database Technology — EDBT '94*, vol. 779, Springer Berlin / Heidelberg, 1994, pp. 187–200.

[80] P. J. Guinan, J. G. Cooprider, and S. Faraj, "Enabling software development team performance during requirements definition: a

behavioral versus technical approach," *Information Systems Research*, vol. 9, no. 2, pp. 101–125, Jun. 1998.

[81] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: a survey," *IEEE Transactions on Software Engineering*, vol. 30, no. 5, pp. 295– 310, May 2004.

[82] D. Del Gobbo, M. Napolitano, J. Callahan, and B. Cukic, "Experience in developing system requirements specification for a sensor failure detection and identification scheme," in *High-Assurance Systems Engineering Symposium, 1998. Proceedings. Third IEEE International*, 1998, pp. 209–212.

[83] C. Smidts, M. Stutzke, and R. W. Stoddard, "Software reliability modeling: an approach to early reliability prediction," *IEEE Transactions on Reliability*, vol. 47, no. 3, pp. 268–278, Sep. 1998.

[84] L. Chung and J. do Prado Leite, "On non-functional requirements in software engineering," in *Conceptual Modeling: Foundations and Applications*, vol. 5600, Springer Berlin / Heidelberg, 2009, pp. 363–379.

[85] S. Lauesen, *Software Requirements: Styles & Techniques*, 1st ed. Addison-Wesley Professional, 2002.

[86] P. Zave, "Classification of research efforts in requirements engineering," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 315–321, Dec. 1997.

[87] P. Zave and M. Jackson, "Four dark corners of requirements engineering," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 1, pp. 1–30, Jan. 1997.

[88] A. Dardenne, S. Fickas, and A. van Lamsweerde, "Goal-directed concept acquisition in requirements elicitation," in *Proceedings of the 6th international workshop on Software specification and design*, Los Alamitos, CA, USA, 1991, pp. 14–21.

[89] Object Management Group, *UML Profile for Schedulability, Performance, and Time, v1.1, formal/05-01-02*. OMG, 2005.

[90] A. Dardenne, A. van Lamsweerde, and S. Fickas, "Goal-directed requirements acquisition," in *Selected Papers of the Sixth International Workshop on Software Specification and Design*, Amsterdam, The Netherlands, 1993, pp. 3–50.

[91] E. S.-K. Yu, "Towards modelling and reasoning support for early-phase requirements engineering," in *Proceedings of the Third IEEE International*

Symposium on Requirements Engineering, Washington, DC, USA, 1997, pp. 226–235.

[92] D. T. Ross and K. E. Schoman Jr, "Structured analysis for requirements definition," in *Classics in software engineering*, Upper Saddle River, NJ, USA: Yourdon Press, 1979, pp. 363–386.

[93] C. P. Gane and T. Sarson, *Structured Systems Analysis: Tools and Techniques*. Prentice Hall Professional Technical Reference, 1979.

[94] K. Orr, *Structured requirements definition*. Topeka, KS: Ken Orr and Associates Inc, 1981.

[95] K. L. Heninger, J. W. Kallander, D. L. Parnas, and J. E. Shore, "Software requirements for the A-7 E aircraft," Naval Research Lab., Washington D.C., 1978.

[96] K. L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Application," *IEEE Transactions on Software Engineering*, vol. 6, no. 1, pp. 2–13, 1980.

[97] D. L. Parnas and J. Madey, "Functional Documentation for Computer Systems Engineering," McMaster University, CRL, Telecommunications Research Institute of Ontario (TRIO), Hamilton, Ont., Canada, CRL Report 237, 1991.

[98] C. Heitmeyer, B. Labaw, and D. Kiskis, "Consistency checking of SCR-style requirements specifications," in *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, Washington, DC, USA, 1995, pp. 56–63.

[99] C. Heitmeyer and D. Mandrioli, *Formal Methods for Real-Time Computig*. New York, NY, USA: John Wiley and Sons, 1996.

[100] S. Faulk, J. Brackett, P. Ward, and J. Kirby, "The Core method for real-time requirements," *IEEE Software*, vol. 9, no. 5, pp. 22–33, 1992.

[101] M. S. Jaffe, N. G. Leveson, M. P. E. Heimdahl, and B. E. Melhart, "Software requirements analysis for real-time process-control systems," *IEEE Transactions on Software Engineering*, vol. 17, no. 3, pp. 241–258, 1991.

[102] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese, "Requirements specification for process-control system," *IEEE Transactions on Software Engineering*, vol. 20, no. 9, pp. 684–707, 1994.

[103] M. P. E. Heimdahl and N. G. Leveson, "Completeness and consistency in hierarchical state-based requirements," *IEEE Transactions on Software*

Engineering, Special issue: best papers of the 17th International Conference on Software Engineering (ICSE-17), vol. 22, no. 6, pp. 363–377, 1996.

[104] Object Management Group, *Common Object Request Broker Architecture (CORBA) 3.1*. OMG, 2008.

[105] Object Management Group, *Common Object Request Broker Architecture (CORBA) For Embedded (CORBAe), Version 1.0*, vol. ptc/2008–02–02. OMG, 2008.

[106] Object Management Group, *Real-time CORBA, Version 1.2*. 2002.

[107] R. Klefstad, D. C. Schmidt, and C. O'Ryan, "Towards highly configurable real-time object request brokers," in *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, Los Alamitos, CA, USA, 2002, pp. 437–447.

[108] D. C. Schmidt, D. L. Levine, and S. Mungee, "The design of the TAO real-time object request broker," *Comput.Commun.*, vol. 21, no. 4, pp. 294–324, 1998.

[109] S. Vestal, *MetaH Users Manual, Version 1.27*. 3660 Technology Drive, Minneapolis, MN 55418: Honeywell Technology Center, 1998.

[110] P. Clements, "A survey of architecture description languages," in *Proceedings of the 8th International Workshop on Software Specification and Design*, 1996, pp. 16–25.

[111] Chrona, "Scientific foundations - technology - chrona." [Online]. Available: http://www.chrona.com/en/technology/scientific-foundations/. [Accessed: 06-Apr-2012].

[112] E. Farcas, C. Farcas, W. Pree, and J. Templ, "Real-Time Component Integration Based on Transparent Distribution," in *Proceedings of the 2nd workshop on Software Engineering for Automotive Systems (SEAS) at 27th ACM International Conference on Software Engineering (ICSE)*, St. Louis, 2005.

[113] E. Farcas, C. Farcas, W. Pree, and J. Templ, "Transparent Distribution of Real-Time Components Based on Logical Execution Time," in *Proceedings of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Chicago, Illinois, USA, 2005, pp. 31–39.

[114] C. Farcas and W. Pree, "A Deterministic Infrastructure for Real-Time Distributed Systems," in *Proceedings of the ECRTS Workshop on*

Operating Systems Platforms for Embedded Real-Time applications (OSPERT), 2007.

[115] E. Farcas and W. Pree, "Hyperperiod Bus Scheduling and Optimizations for TDL Components," in *Proceedings of the 12th IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, 2007, p. 1262—1269.

[116] C. Farcas and W. Pree, "Virtual Execution Environment for Real-Time TDL Components," in *Proceedings of the 12th IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, 2007, p. 93—100.

[117] W. Pree, G. Stieglbauer, and J. Templ, "Simulink Integration of Giotto/TDL," in *Automotive Software – Connected Services in Mobile Networks*, vol. 4147, Springer, 2006, pp. 137–154.

[118] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard, Eds., *Web Services Architecture*. W3C, 2004.

[119] C. Prehofer, "Plug-and-play composition of features and feature interactions with statechart diagrams," *Software and Systems Modeling*, vol. 3, no. 3, pp. 221–234, 2004.

[120] P. Zave, "Feature interactions and formal specifications in telecommunications," *Computer*, vol. 26, no. 8, pp. 20–28, 30, 1993.

[121] Object Management Group, *OMG Systems Modeling Language (OMG SysML™), Version 1.2*, vol. formal/2010–06–01. OMG, 2010.

[122] M. Broy, I. H. Krüger, and M. Meisinger, "A formal model of services," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 1, Feb. 2007.

[123] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, *Web Services Description Language (WSDL) 1.1*. W3C, 2001.

[124] *UDDI Version 3.0.2*. OASIS, 2004.

[125] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon, Eds., *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. W3C, 2007.

[126] OASIS, *Web Services Business Process Execution Language Version 2.0*, vol. OASIS Standard. 2007.

[127] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, and C. Barreto, Eds., *Web Services Choreography Description Language Version 1.0*. W3C, 2005.

[128] AUTOSAR, "Automotive Open System Architecture." [Online]. Available: http://www.autosar.org. [Accessed: 19-Jun-2012].

[129] D. Skvorc, S. Srbljic, and M. Podravec, "Virtual Network for Development and Execution of Service-Oriented Applications," in *International conference on Networking and Services (ICNS '06)*, 2006, p. 96.

[130] S. T. Taft and R. A. Duff, *Ada 95 Reference Manual: Language and Standard Libraries*, vol. 1246. Springer, 1995.

[131] C. A. R. Hoare, *CSP--Communicating Sequential Processes*. Prentice Hall, 1985.

[132] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull, *The Real-Time Specification for Java*. Addison Wesley, 2000.

[133] G. Berry, *The Esterel v5 Language Primer - Version v5 91*. 06902 Sophia-Antipolis CDX, France: INRIA, 2000.

[134] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, p. 231—274, Jun. 1987.

[135] F. Maraninchi, "The Argos language: Graphical representation of automata and description of reactive systems," in *IEEE Workshop on Visual Languages*, Kobe, Japan, 1991.

[136] C. André, "Representation and analysis of reactive behaviors: a synchronous approach," in *Proceedings of CESA'96*, Lille, France, 1996.

[137] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice, "Lustre: A declarative language for programming synchronous systems," in *14th ACM Symposium on Principles of Programming Languages*, Munich, Germany, 1987, pp. 178–188.

[138] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data-flow programming language Lustre," *Proceedings of the IEEE*, vol. 79, no. 9, p. 1305—1320, Sep. 1991.

[139] P. L. Guernic, T. Gautier, M. L. Borgne, and C. de Marie, "Programming real-time applications with Signal," *Proceedings of the IEEE*, vol. 79, no. 9, p. 1321—1335, Sep. 1991.

[140] F. C. Hennie, *Finite-state Models for Logical Machines*. John Wiley & Sons, 1968.

[141] G. J. Holzmann, *The Spin Model Checker*. Addison-Wesley Professional, 2003.

[142] V. Ermagan, C. Farcas, E. Farcas, I. H. Krüger, and M. Menarini, "A Service-Oriented Approach to Failure Management," in *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV*, 2008.

[143] H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps, "Temporal Logic for Scenario-Based Specifications," in *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin / Heidelberg, 2005, pp. 445–460.

[144] R. Alur and M. Yannakakis, "Model Checking of Message Sequence Charts," in *Proceedings of the 10th International Conference on Concurrency Theory*, London, UK, 1999, pp. 114–129.

[145] A. Werner, "Visual TDL - The Timing Description Language Integrated in Simulink," Department of Computer Science, University of Salzburg, Austria, 2005.

[146] C. M. Kirsch, M. A. A. Sanvido, and T. A. Henzinger, "A Programmable Microkernel for Real-Time Systems," in *Proc. ACM/USENIX Conference on Virtual Execution Environments (VEE)*, 2005, pp. 35–45.

[147] G. Menkhaus, S. Fischmeister, M. Holzmann, and C. Farcas, "Towards Efficient Use of Shared Communication Media in the Timed Model," in *11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS 2005)*, 2005, pp. 342–351.

[148] C. Farcas, "Towards Portable Real-Time Software Components," Department of Computer Science, University of Salzburg, Austria, 2006.

[149] A. Arora and S. S. Kulkarni, "Component based design of multitolerant systems," *IEEE Transactions on Software Engineering*, vol. 24, no. 1, pp. 63–78, 1998.

[150] F. C. Gärtner and A. Jhumka, "Automating the Addition of Fail-Safe Fault-Tolerance: Beyond Fusion-Closed Specifications," in *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, 2004, pp. 183–198.

[151] P. C. Attie, A. Arora, and E. A. Emerson, "Synthesis of fault-tolerant concurrent programs," *ACM Trans.Program.Lang.Syst.*, vol. 26, no. 1, pp. 125–185, 2004.

[152] J.-L. Camus and B. Dion, *Efficient Development of Airborne Software with Scade Suite*. Esterel Technologies, 2003.

[153] P. Caspi and P. Raymond, "From Control system design to embedded code: the synchronous data-flow approach," in *40th IEEE Conference on Decision and Control*, 2001, pp. 3278–3283.

[154] H. Kopetz and G. Bauer, "The Time Triggered Architecture," *Proceedings of the IEEE - Special Issue on Modeling and Design of Embedded Software*, vol. 91, no. 1, pp. 112–126, 2002.

[155] G. Durrieu, O. Laurent, C. Seguin, and V. Wiels, "Formal Proof and Test Case Generation for Critical Embedded Systems Using Scade," in *Building the Information Society*, vol. 156/2004, Springer, 2004, pp. 499–504.

[156] Object Management Group, *Fault Tolerant CORBA*, vol. formal/04–03–21 chapter 23. OMG, 2009.

[157] D. Liang, C.-L. Fang, C. Chen, and F. Lin, "Fault tolerant Web service," in *Software Engineering Conference, 2003. Tenth Asia-Pacific*, 2003, pp. 310–319.

[158] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer, "Path-based failure and evolution management," in *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, 2004, pp. 309–322.

[159] K. R. Joshi, M. A. Hiltunen, W. H. Sanders, and R. D. Schlichting, "Automatic model-driven recovery in distributed systems," in *Reliable Distributed Systems, 2005. SRDS 2005. 24th IEEE Symposium on*, 2005, pp. 25 – 36.

[160] N. Salatge and J.-C. Fabre, "Fault Tolerance Connectors for Unreliable Web Services," in *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, 2007, pp. 51 –60.

[161] S. Bhiri, W. Gaaloul, and C. Godart, "Discovering and Improving Recovery Mechanisms of Composite Web Services," in *Web Services, 2006. ICWS '06. International Conference on*, 2006, pp. 99 –110.

[162] S. Hwang and C. Kesselman, "Grid workflow: a flexible failure handling framework for the grid," in *High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on*, 2003, pp. 126 – 137.

[163] P. A. Lee and T. Anderson, *Fault tolerance, principles and practice*, vol. 2nd. Springer-Verlag, 1990.

[164] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, no. 1, pp. 11–33, Jan. 2004.

[165] S. S. Kulkarni and A. Ebnenasir, "Automated synthesis of multitolerance," in *Dependable Systems and Networks, 2004 International Conference on*, 2004, pp. 209–218.

[166] US Gov., *Procedures for Performing a Failure Mode, Effects and Criticality Analysis*, vol. MIL-STD-1629A. Military Standard, 1980.

[167] Committee Automotive Quality And Process Improvement, *Potential Failure Mode and Effects Analysis in Design (Design FMEA) and Potential Failure Mode and Effects Analysis in Manufacturing and Assembly Processes (Process FMEA) and Effects Analysis for Machinery (Machinery FMEA)*, vol. J1739. SAE, 2002.

[168] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl, *Fault Tree Handbook*. Washington, DC: U.S. Nuclear Regulatory Commission, 1981.

[169] P. L. Goddard, "Software FMEA techniques," in *Reliability and Maintainability Symposium, 2000. Proceedi gs. Annual*, 2000, pp. 118–123.

[170] N. G. Leveson and P. R. Harvey, "Analyzing Software Safety," *IEEE Transactions on Software Engineering*, vol. 9, no. 5, pp. 569–579, 1983.

[171] C. Pu, G. E. Kaiser, and N. C. Hutchinson, "Split-Transactions for Open-Ended Activities," in *Proceedings of the 14th International Conference on Very Large Data Bases*, 1988, pp. 26–37.

[172] D. Skeen, "Nonblocking commit protocols," in *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, Ann Arbor, Michigan, 1981, pp. 133–142.

[173] J. Gray and L. Lamport, "Consensus on transaction commit," *ACM Trans. Database Syst.*, vol. 31, no. 1, pp. 133–160, 2006.

[174] P. Pires, M. Benevides, and M. Mattoso, "Building Reliable Web Services Compositions," in *Web, Web-Services, and Database Systems*, vol. 2593, Springer, 2003, pp. 59–72.

[175] S. Tai, T. Mikalsen, E. Wohlstadter, N. Desai, and I. Rouvellou, "Transaction policies for service-oriented computing," *Data Knowl.Eng.*, vol. 51, no. 1, pp. 59–79, Oct. 2004.

[176] E. Fersman, "A Generic Approach to Schedulability Analysis of Real-Time Systems," Faculty of Science and Technology, UPPSALA University, 2003.

[177] V. A. Braberman, "Modeling and Checking Real-Time System Designs," Departamento de Computación, Universidad de Buenos Aires, 2000.

[178] J. W. S. Liu, *Real-Time Systems*. Prentice-Hall, 2000.

[179] G. C. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*, vol. 1st. Kluwer Academic Publishers, 1997.

[180] R. Milner, *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press, 1999.

[181] D. Harel, *Modeling Reactive Systems With Statecharts : The Statemate Approach*. McGraw-Hill, 1998.

[182] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A prototype verification system," in *11th International Conference on Automated Deduction (CADE)*, London, UK, 1992, vol. 607, pp. 748–752.

[183] L. C. Paulson, *Isabelle: A Generic Theorem Prover*. Springer-Verlag Berlin / Heidelberg, 1994.

[184] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 2000.

[185] T. A. Henzinger, Z. Manna, and A. Pnueli, "Temporal Proof Methodologies for Timed Transition-Systems," *Information and Computation*, vol. 112, no. 2, pp. 273–337, Aug. 1994.

[186] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, Apr. 1994.

[187] R. Alur and T. A. Henzinger, "Logics and models of real time: a survey," in *Real Time: Theory in Practice*, vol. 600/1992, Springer, 1992, pp. 74–106.

[188] S. Yovine, "Kronos: A verification tool for real-time systems," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 1, no. 1–2, pp. 123–133, 1997.

[189] K. Larsen, P. Petterson, and W. Yi, "Uppaal in a nutshell," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 1, no. 1–2, pp. 134–152, 1997.

[190] S. Campos, E. Clarke, W. Marrero, and M. Minea, "Verus: A Tool for Quantitative Analysis of Finite-State Real-Time Systems," in *Proc.*

Workshop on Languages, Compilers and Tools for Real-Time Systems, 1995, pp. 70–78.

[191] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi, "HyTech: A model checker for hybrid systems," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 1, no. 1–2, pp. 110–122, 1997.

[192] J. Sifakis, "Modeling Real-Time Systems - Challenges and Work Directions," in *Proceedings of the International Workshop on Embedded Software (EMSOFT)*, 2001, vol. 2211, pp. 373–389.

[193] J. Sifakis, S. Tripakis, and S. Yovine, "Building Models of Real-Time Systems from Application Software," in *Proceedings of the IEEE - Special Issue on Modeling and Design of Embedded Software*, 2003, vol. 91, pp. 100–111.

[194] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine, "Taxys: a tool for the development and verification real-time embedded systems," in *Lecture Notes in Computer Science*, Paris, France, 2001, vol. 2102, pp. 391–395.

[195] E. Song, R. Reddy, R. France, I. Ray, G. Georg, and R. Alexander, "Verifiable composition of access control and application features," in *SACMAT '05: Proceedings of the tenth ACM symposium on Access control models and technologies*, 2005, pp. 120–129.

[196] J. Klein, L. Hélouët, and J.-M. Jézéquel, "Semantic-based weaving of scenarios," in *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, 2006, pp. 27–38.

[197] S. Clarke and R. J. Walker, "Towards a standard design language for AOSD," in *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, 2002, pp. 113–119.

[198] S. Clarke and R. J. Walker, "Composition patterns: an approach to designing reusable aspects," in *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, 2001, pp. 5–14.