



Traceability of Concerns and Observer-Based Verification for Railway Safety-Critical Software

Marc Sango

► To cite this version:

Marc Sango. Traceability of Concerns and Observer-Based Verification for Railway Safety-Critical Software. Software Engineering [cs.SE]. Université de Lille 1, 2015. English. NNT : . tel-01208083

HAL Id: tel-01208083

<https://theses.hal.science/tel-01208083>

Submitted on 1 Oct 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Traceability of Concerns and Observer-Based Verification for Railway Safety-Critical Software

THÈSE

présentée et soutenue publiquement le 18 September 2015

pour l'obtention du

Doctorat de l'Université Lille I
(spécialité informatique)

par

MARC SANGO

Composition du jury

Rapporteurs : Pr. Jean-Charles Fabre, *INP-ENSEEIH*T - France
Pr. Philippe Dhaussy, *ENSTA-Bretagne* - France

Examineurs : Pr. Franck Barbier, *Université de Pau* - France
DR. Simon Collart-Dutilleul, *IFSTTAR/COSYS/ESTAS* - France

Invité : Dr. Yannick Moy, *AdaCore* - France

Directeurs : Pr. Laurence Duchien, *Université Lille 1* - France
CR. Christophe Gransart, *IFSTTAR/COSYS/LEOST* - France

Mis en page avec la classe thloria.

Acknowledgement

Foremost, I would like to express my gratitude to my supervisors, Prof. Laurence Duchien, and Dr. Christophe Gransart, for their continuous support during my Ph.D. study. Their advices, knowledge and support have been invaluable on both academic and personal levels.

Next, I would like to thank the members of my thesis committee. A special thanks to Prof. Jean-Charles Fabre and Prof. Philippe Dhaussy for having accepted to review my manuscript. They gave me valuable comments, and I really appreciated their feedback.

Besides my advisors, I would like to thank Dr. Charles Tatkeu and Prof. Lionel Seinturier for welcoming me in the IFSTTAR/COSYS/LEOST and INRIA/SPIRAL teams, and for their continuous advices throughout my thesis years.

In addition, I thank all my colleagues at the both research teams, and my colleagues at IUT'A and at University Lille 1, for the inspiring discussions about research, teaching and the life of every day. A big thank for my friends for their help and tips during my years in Lille and before Lille.

My biggest gratitude goes to all my family, for their enduring support and understanding. This thesis would not have been possible without your encouragements.

Finally, as we say in my native tongue, Bissa¹, one word can often replace a long speech. So BARKA, which merely means thanks in Bissa.

¹[https://fr.wikipedia.org/wiki/Bissa_\(langue\)](https://fr.wikipedia.org/wiki/Bissa_(langue))

Abstract

In recent years, the development of critical systems demands more and more software. This growth has generated many strategic and organizational impacts among critical software development and verification actors. In order to reduce their costs of development and verification, actors in critical domains, such as avionics and automotive domains, are moving more and more towards model-driven engineering. In contrast, in the railway domain, for strategic and organizational reasons, actors remain faithful to traditional methods that allow them to take advantage of their knowledge. At the same time, they use ad-hoc reuse approaches for capitalizing on the reuse of their accumulated software achievements. However, these conventional approaches supplemented by ad-hoc reuse techniques suffer from a lack of abstraction and do not provide an automated and formal support for reasoning about rules related to the reuse of software components. On the other hand, railway software is designed to be sustainable. The lifetime of a train and, thus, its software is over thirty years. It is therefore necessary to design software in order to be able to evolve it in time and space.

To address these shortcomings, we present in this thesis a systematic approach based on model driven engineering and basic models of components, in order to better manage software complexity and traceability of functional and non-functional requirements. In this dissertation, we provide in particular three major contributions. First, we provide an integrated set of meta-models for describing the concerns of software requirements, software components, and traceability between the concerns and software components. By providing an abstract model, we are independent of any implementation and thus allow existing approaches relying on that model to expand their support. With the second contribution, we propose a formal support of our model to allow formal verification. We focus on temporal property verification. For this, our design model is translated into timed automata for which we can apply a timed model checker. Instead of using temporal logic, which is difficult to handle by formal verification non-experts, we use patterns of temporal properties. For each pattern identified in the railway case studies, we propose timed automata that can be applied directly into a timed model checking tool. These timed automata are seen as observers or watch dogs that check the system under observation. Finally, with the last contribution, we propose a software component-based development and verification approach, called SARA, and included in V-lifecycle widely used in the railway domain.

Experiments we conducted to validate our approach through a few case studies of the new European train control system ERTMS/ETCS, show that by using component model that explicitly include requirement traceability, we are able to provide a practical, scalable and reliable approach. Empirical experiments also show that this approach can reduce the development cost for the users who are looking at reusability for long-term benefits, even in the presence of initial overhead cost introduced by component-based development.

Résumé

Ces dernières années, le monde des systèmes critiques a connu un véritable essor en matière de demande de logiciels. Cet essor a généré de très nombreux impacts stratégiques et organisationnels chez les acteurs de développement du logiciel critique. Dans une optique majeure de réduction des coûts de développement et de réutilisation de leur réalisations accumulées, les grands acteurs du monde critique comme ceux de l'avionique et de l'automobile s'orientent de plus en plus vers l'ingénierie dirigée par les modèles. Par contre les acteurs du domaine ferroviaire, pour des raisons stratégiques et organisationnelles restent encore fidèles à des méthodes conventionnelles qui leur permettent de tirer au maximum profit de leurs compétences. Dans le même temps, ils utilisent des approches ad-hoc qui permettent de capitaliser sur la réutilisation de leurs réalisations logicielles. Cependant, ces approches conventionnelles complétées par des techniques ad-hoc de réutilisation souffrent d'un manque d'abstraction et ne fournissent pas un support automatisé et formel pour raisonner sur des règles liées à la réutilisation de composants logiciels. D'autre part, les logiciels du ferroviaire ont pour vocation à être pérennes. La durée de vie d'un train et, donc, de son logiciel est de plus de trente ans. Il est donc nécessaire de concevoir ces logiciels de manière à pouvoir les faire évoluer dans le temps et dans l'espace.

Pour faire face à ces limitations, nous présentons dans cette thèse une approche systématique basée sur l'ingénierie dirigée par les modèles et les modèles à base de composants, de façon à maîtriser au mieux la complexité des logiciels et la traçabilité des exigences fonctionnelles et non-fonctionnelles. Pour atteindre cet objectif de traçabilité des exigences, nous nous fondons également sur un des principes fondamentaux de l'ingénierie logicielle qui est la séparation des préoccupations. Dans cette dissertation, nous proposons notamment trois contributions essentielles. En premier lieu, nous fournissons un ensemble uniformisé de méta-modèles permettant de décrire les préoccupations des exigences logicielles, les composants logiciels, et la traçabilité entre les préoccupations et ces composants logiciels. L'avantage de cette traçabilité est de permettre une traçabilité explicite des préoccupations à l'intérieur ou entre les différents niveaux d'abstraction du modèle et, ceci, indépendamment de tout mécanisme de transformation de modèles. En fournissant un modèle abstrait, nous sommes donc indépendants de toute implémentation et permettons ainsi aux approches existantes de s'appuyer sur ce modèle pour étendre leur support. Avec la deuxième contribution, nous proposons un support formel de notre modèle pour en permettre la vérification formelle. Le modèle est défini afin de faciliter la vérification mixte des propriétés fonctionnelles et non-fonctionnelles. Ainsi les propriétés fonctionnelles d'entrée/sorties des composants peuvent être vérifiées par les outils de preuve formelle. Pour les propriétés temporelles, le modèle est traduit en automates temporisés sur lesquels on peut appliquer les outils de «model checking» temporisé. Au lieu d'utiliser la logique temporelle, difficile à manipuler par des non-experts, nous utilisons des motifs de propriétés temporelles.

Pour quelques motifs identifiés dans le domaine ferroviaire, nous proposons des automates temporisés qui peuvent être appliqués directement dans un outil de « model checking » temporisé. Ces automates temporisés sont vus comme des observateurs ou des chiens de garde du système vérifié. Cette technique de « model checking » est appelée la vérification par observateurs. Finalement, la dernière contribution propose une approche de développement et de vérification à base de composants logiciels, nommée SARA pour « SAFety-critical RAilway control applications », qui s'intègre dans le cycle de développement en V très largement utilisé dans le domaine ferroviaire.

Les expérimentations conduites pour évaluer notre approche à partir de quelques cas d'études du nouveau système européen de contrôle de train ERTMS/ETCS, montrent qu'en utilisant des modèles à base de composants qui intègrent explicitement la traçabilité des exigences, nous sommes capables de fournir une approche pratique, extensible et fiable. Les expérimentations empiriques montrent aussi que cette approche peut réduire le coût de développement pour l'utilisateur qui se penche sur la réutilisation des composants à long terme, et ce même en présence du surcoût initial introduit par le développement, la vérification et le stockage d'un nouveau composant avec ses liens de traçabilité ou de la recherche et l'adaptation d'un composant existant pour un nouveau projet.

Contents

List of Tables	xiii
Part I Motivation and Context	1
Chapter 1 Introduction	3
1.1 Problem Statements	5
1.2 Research Goals	6
1.3 Contributions	7
1.4 Dissertation Roadmap	8
1.5 Publications	10
Part II State of the Art	13
Chapter 2 Component-Based Modeling and Observer-Based Verification	15
2.1 Introduction	15
2.2 CBSE Background	16
2.3 Focus on CBSE for Embedded System Design	22
2.4 V&V Background	24
2.5 Focus on Observer-Based Verification	30
2.6 Comparative Analysis and Discussion	33
2.7 Summary	37

Chapter 3 Traceability of Concerns	39
3.1 Introduction	39
3.2 Traceability Background	40
3.3 Traceability Approaches in MDE	46
3.4 Comparative Analysis and Discussion	48
3.5 Summary	56
 Part III Contribution	 59
 Chapter 4 Component-Based Modeling with Traceability of Concerns	 61
4.1 Introduction	61
4.2 Motivation and Challenges	63
4.3 SARA Meta-Model	66
4.4 Process to Use the Meta-Model	73
4.5 Challenges Revisited and Lessons Learned	78
4.6 Summary	81
 Chapter 5 Observer-Based Verification with Patterns of Properties	 83
5.1 Introduction	83
5.2 Motivation and Challenges	84
5.3 SARA to TAIO Formal Model	88
5.4 A 3-Layer Approach for OBV	108
5.5 Challenges Revisited and Lessons Learned	112
5.6 Summary	114

Part IV Validation	115
Chapter 6 Validation Through Railway Safety-Critical Software	117
6.1 Introduction	118
6.2 Overview of SARA Process	118
6.3 A Brief Presentation of ERTMS/ETCS	120
6.4 Rail-Road Level Crossing Case Study	122
6.5 RBC Handover Case Study	130
6.6 Metrics for Model Transformation and Component Reuse	137
6.7 Threats to Validity and Discussion	145
6.8 Summary	147
Part V Conclusion	149
Chapter 7 Conclusion and Perspectives	151
7.1 Summary of the Dissertation	151
7.2 Review of Research Questions	152
7.3 Perspectives	154
Appendices	157
Appendix A SARA Model Implementation in Ada Language	159
Bibliography	163

List of Figures

2.1	Components and component Composition in UML	17
2.2	An idealized component life cycle	20
2.3	Idealised component and system life cycles	21
2.4	Different levels of control in a train control system	23
2.5	Verification and validation in software development	25
2.6	Observer of time-bounded response pattern	32
3.1	Backward and forward traceability	41
3.2	Pre-RS and Post-RS traceability	42
3.3	Inter and extra-requirement traceability	43
3.4	Traceability of concerns	44
4.1	An example of model driven view	64
4.2	Concern meta-model	67
4.3	Component meta-model	69
4.4	Traceability meta-model	71
4.5	A process to apply the metamodel and tracing concerns	74
5.1	LC-APS motivating example	85
5.2	An execution trace of Ada implementation for LC-APS	87
5.3	Component instance purposes : (1) deployment and (2) binding	91

5.4	Example of black-box component: a buffer component which implements connection type DC_1 of Figure 5.1	95
5.5	Examples of passive composition: (a) plugging composition, (b) hiding after plugging, (c) feedback, (d) hiding after feedback	99
5.6	Examples of active composition: (c) coordination of two one-place buffers to define a three-place buffer, (d) coordination of two one-place buffers to define a two-place buffer	99
5.7	(a) Gate component UML SM, (b) The corresponding UPPAAL TAIO	104
5.8	Example of TAIO connections for Figure 5.1	105
5.9	UPPAAL model declaration of Figure 5.1	106
5.10	The overview of our OBV approach	109
5.11	(a) Absence before observer, (b) Response max delay observer	110
6.1	SARA component-based development and verification process included in CENELEC prescribed V-Lifecycle	119
6.2	System architecture of the ERTMS/ETCS and its interfaces	120
6.3	3-Layer Temporal QoS Ontology for ERTMS/ETCS	122
6.4	A level crossing topography	123
6.5	UPPAAL TAIO model of LC-APS components	126
6.6	The forward trace query definition	129
6.7	A result of trace query	130
6.8	A RBC-RBC handover topography	131
6.9	RBC-RBC handover scenario	133
6.10	UPPAAL TAIO model of RBC handover	134
6.11	Composition of RBC TAIO model with response delay observer	135
6.12	RBC handover simulation with ERTSA Simulator	136
6.13	Effect of increasing the size of input model in model transformation	138
6.14	Effect of increasing the complexity of input model in model transformation	139
6.15	Cost benefit analysis with C&P approach	142
6.16	Cost benefit analysis with CBD approach	143
6.17	Component by component cost saving in both C&P and CBD approaches	144
6.18	C&P versus CBD cost benefit analysis	145
6.19	Use cases and its actors	146
6.20	Exploration time	146

List of Tables

2.1	The main industrial needs and the concerns they impact on	23
2.2	Synthesis of comparison for some CBD-V approaches	35
3.1	Summary of some traceability approaches.	48
3.2	Comparison of some traceability approaches	50
3.3	Synthesis of comparison for some traceability approaches	57
4.1	Synthesis of comparison for some traceability approaches including ours . . .	79
5.1	Synthesis of comparison for some CBD-V approaches including ours	113
6.1	Validation results of requirements identified in Section 6.4.1	128
6.2	Unacceptable output observed	135
6.3	Cost estimation for LC-APS V1 development with reuse C&P approach	142

Part I

Motivation and Context

Introduction

Contents

1.1 Problem Statements	5
1.2 Research Goals	6
1.3 Contributions	7
1.4 Dissertation Roadmap	8
1.5 Publications	10

Software in safety-critical embedded systems, particularly safety-critical transportation systems, such as aeronautic, railway and automotive unmanned or control systems, is becoming more and more complex, requiring increasing functional and dependability requirements. This growth has generated many strategic and organizational impacts among critical software development and verification actors. In order to reduce their costs of development and verification, actors in avionics and automotive domains, are moving more and more towards model-driven engineering [Peleska, 2013]. For example, the new avionic standard DO-178C, for software considerations in airborne systems, is complemented by several supplements, such as DO-331 for model-based development and verification supplement [DO-331, 2011]. In the same way, in automotive standard ISO 26262, Appendix B of Part 6 is dedicated to model-based development [ISO-26262, 2009]. In contrast, in the new version of railway standard [EN-50128, 2011] for safety-critical software of control and protection systems, the model-based development is not particularly discussed. For example, the word “model-driven” is used only once in the informative Annex D.

In addition to these standard observations, in practice, for strategic and organizational reasons, software actors in railway domain remain faithful to traditional methods that allow them to take advantage of their knowledge. At the same time, they use ad-hoc reusability approaches for capitalizing on the reuse of their accumulated software achievements

[Riaz, 2012]. However, these conventional approaches supplemented by ad-hoc reuse techniques suffer from a lack of abstraction and do not provide an automated and formal support for reasoning about rules related to the reuse of software components. On the other hand, railway software is designed to be sustainable. The lifetime of a train and, thus, its software is over thirty years. It is therefore necessary to design software in order to be able to evolve it in time and space.

From standards and practice observations, we can say that software engineering for safety-critical systems, such as railway control and protection systems, is facing three major challenges: (1) the increasing complexity of systems to be developed, (2) the assurance of maintenance, and (3) the higher demand of quality in terms of Safety Integrity Level (SIL) [EN-50128, 2011]. One effective means to handle the complexity challenge is the *separation of concerns*, which is one of the fundamental principles of computer science, first advocated by [Dijkstra, 1982b]. On the other hand, the assurance of maintenance and high-integrity can be enhanced by the application of *traceability of concerns* and *formal modeling and verification* at each stage of the software life-cycle in order to ensure that requirements have been properly implemented.

Informally, the *separation of concerns* is to divide and conquer. At any stage of the development of a system, the system is divided according to the system concerns, which include functional concerns and non-functional concerns, such as safety, security, concurrency and so on. These concerns are thus modeled separately and their integration forms a model of the whole system. However, experience shows that it is not easy to practice the separation of concerns if we do not have a rigorous unified semantic theory and development process to separately define and process relations among different concerns [Chen et al., 2007b].

Nevertheless, there exist some rigorous unified semantic theories, such as a theory of contracts with separation of concerns [He et al., 2006, Chen et al., 2009]. On the other hand, there exist also some unified development approaches, such as a component-based process with separation of concerns [Panunzio and Vardanega, 2014], based on the correctness by construction principle [Chapman, 2006], and Model-Driven Engineering (MDE) [Schmidt, 2006].

At the same time, in safety-critical domains in general and particularly in railway control systems, which has lagged behind in the adoption of MDE approaches [Favaro and Sartori, 2014], compared to other safety-critical domains, such as aeronautical and automotive domains, there is a strong need to link methods and theories that highlight separation of concerns. This bridge can help to deal with the other two challenges: *traceability of concerns* and *formal modeling and verification*.

In this context, a unified meta-model, which includes a traceability of concerns in its high-level language, and an underlining formal model, which establishes the semantics of model rules for formal verification, is one of the bridge solutions. However, in this solution case, one issue is the transformation of the high-level model into a low-level model for which a formal verification can be realized. Another issue is to provide an intuitive way for

non-experts of formal theory to annotate its model with properties to be verified without requiring a significant knowledge of higher order logic and theorem proving.

As a solution to these issues, we thus explore in this dissertation a component-based software engineering with traceability of concerns and observer-based verification with patterns of properties.

The remainder of this introductory chapter is organized as follows. In Section 1.1, we identify the problems that motivate this research. Next, in Section 1.2, we present our research goals. Then, we summarize the contributions of our thesis in Section 1.3. In Section 1.4, we give a brief introduction to each of the chapters of the document.

1.1 Problem Statements

In this dissertation, we deal with component-based modeling with traceability of concerns and observer-based verification with patterns of properties. Existing Component-Based Development and Verification (CBD-V) approaches aiming at achieving the same objectives are subject to a number of open problems or limitations hampering the efforts for building a well-suited approach in a specific domain, such as the railway domain. In particular, we have identified that those approaches are confronted to the following issues.

Lack of Explicit Traceability of Concerns in CBD-V. Numerous component models or frameworks are available. The first problem faced by developers in a specific domain, such as the railway domain, is an effective reusability of components. For example, the train control domain is a new domain and there are not enough software components available in the market. So it is still not guaranteed to be able to purchase trusted and certified Commercial Off The Shelf (COTS) components and build software from them. As a consequence, two options are available: whether components should be developed from scratch or identify the reusable software components from existing previous projects. Identifying and modifying reusable software component from existing projects seems to be one of the more promising ways to obtain reusable assets. In this context, the new project specific requirements are a big challenge to cope with, because changing requirements will force us to move on a new version of components. This will raise further problem such as *traceability of concerns*, including intra- and extra-requirement traceability.

This problem statement raises the following research question:

Research Question 1. What is a suitable component model to build safety-critical control software, specially railway control and protection software, with traceability of concerns?

Lack of Safety Interoperability Guarantee. The second problem faced by component-based developers in a specific domain, such as the railway domain, is an effective *interoperability* of components. The sinew of war in current component models is *Black-box composability*, *substitutability* and *reusability*, i.e., there is no need to know the design and the

implementation when composing a component with other parts of the system, substituting a component with another one or reusing it in another application. However, this is not sufficient in a specific domain, where domain knowledge, such as data format and protocol of communication, is required for an effective interoperability. In addition, formal modeling of interfaces that enables such interoperability is necessary to reason and predict the application of composition, substitution and reusability mechanisms. For example, since the European Commission's decision to establish international standardisation of ATC systems with ERTMS/ETCS, interoperability between ATC components has become the key challenge in railway domain.

This problem statement raises the following research question:

Research Question 2. How can safety-critical control software, specially railway control and protection software, be built in an efficient way by using CBD-V rules, such as interoperability and model verification?

Concrete Applications in Railway domain. Another important characteristic of CBD-V is the separation of the development process for individual components, named *component life cycle*, from the development process for the overall system, named *system life cycle*. In the literature, there is an idealized component life cycle entailed in an idealized system lifecycle. However, due to the structure of domain market and established development process, it is difficult to develop the complete train control applications by using idealized component lifecycle without integrating the traditional development lifecycle, such as V-Lifecycle prescribed in the CENELEC standard [EN-50128, 2011] of railway safety-critical software for control and protection systems.

This problem statement raises the following set of research questions:

Research Question 3. Will CBD-V processes replace the need for traditional software development and verification processes? Particularly, what is the suitable development and verification process for railway control and protection software?

1.2 Research Goals

As explained in the previous Section 1.1, building safety-critical control software in an efficient way by using a CBD-V approach implies considering several issues related to traceability of concerns, interoperability and a concrete application by considering the domain concerns. The main objective of this dissertation is thus to provide a solution facing these issues. With such an approach, we want to introduce a unified high-level abstraction of component model, as well as underlining low-level formal model, for which we can use property specification patterns for formal verification. To achieve this, we decompose this objective in the following goals.

Define requirements, components and traceability of concerns meta-models. First of all, our approach has to provide a unified means to describe different levels of software artifacts, *i.e.*, requirement concerns, component concerns and traceability links among these concerns, by using the same formalism, *i.e.*, a set of meta-models. Thus, a unified representation with separation of concerns would be used among all a CBD-V process, to facilitate the traceability of concerns, whatever the functional or non-functional requirement concerns.

Define an underlining formal model of CBD-V rules, such as interoperability. Our approach for modeling software concerns has to provide a means to describe CBD-V, rules, such as interoperability of component interfaces, which provides a baseline for other rules, such as component composition. Thus, we can reason about the application of composition rules. This formal model would be used in the model transformation into a verification model for which we can use formal verification tool.

Moving towards CBD-V in railway safety-critical applications. Another goal is to provide an approach relying on simplicity, flexibility, applicability and reusability in railway safety-critical applications. Regarding simplicity, it must be simple for any developer including V-based developers, to handle and apply. Regarding flexibility, the approach must provide a means to be extended and maintained over time without difficulties. The XML-based implementation of the model helps us in those purposes. Regarding applicability, the approach must be applied in railway concrete case studies. Regarding reusability, the approach must provide cost benefit compared to other software reuse strategies. The cost estimation of the development and verification of our case studies through our systematic approach and the ad-hoc copy paste approach helps us to analyze the cost benefit of our approach.

1.3 Contributions

In this section, we present an overview of the contributions described in this dissertation. As stated before in Section 1.2, the goal of our research is to provide our approach conceptual meta-model, an underlining formal model, and its application facilities. As a consequence, the main contributions of our work are summarized in three parts:

A Component-Based Modeling with Traceability of Concerns. Our first contribution is an integrated meta-model for requirement concerns, component concerns and traceability of concerns. The benefit of *concern meta-model* is the representation of functional and non-functional requirements as scenarios with nominal and degraded modes in order to deal with specific dysfunctions affecting temporal and safety concerns. The benefit of *component meta-model* is that it supports the integration of non-functional concerns, specially temporal safety concerns coming from a system concern modeling. The model elements are clearly presented with separation of concerns in order to facilitate the traceability of concerns. The benefit of *traceability meta-model* is to enable traceability of concerns within or across model

abstraction levels independently of any model transformation mechanism. By providing abstract meta-models, we are thus implementation-independent.

An Observer-based Verification with Patterns of Concerns. As a second contribution, we provide a formal support for our conceptual meta-model. The model is defined to handle mix formal verification, *i.e.*, *formal proof* of functional input and output constraints and *model checking* of non-functional real-time constraints in order to take advantage of both formal methods. The formal model is thus translated into the timed automata model for which we use a timed model checker tool to verify temporal requirements. The behavioral equivalence of each source model and its corresponding generated model is guaranteed by bisimulation relation with respect to reachability. Instead of using temporal logic specifications, which are difficult to handle by non-experts, we use pattern-based specifications, which propose user-friendly syntax. For each pattern, *observer automata*, which can be applied directly in a timed automata model checker, are constructed. We call this model checking technique *observer-based verification with patterns of concerns*. We thus demonstrate that the defined observers have no impact on the behavior of the system under observation, meaning that any trace of the observed system is preserved in the composition of the system and the observers.

An Evaluation Process. Finally, besides the theoretical framework, we provide a concrete evaluation of our approach, named SARA, dedicated to SAFETY-critical RAILway control applications. We have included our SARA process in railway CENELEC prescribed V-Lifecycle. The evaluation is realized through different railway case studies derived from ERTMS/ETCS specification for which we have provided an ontology. We have implemented our meta-model in the XML-based tool and use XQuery queries to infer trace information. Then, we translated XML-language into the Ada programming language for which we use the SPARK annotation tool for formal proof of component functional input and output constraints *i.e.*, classical pre- and post-conditions. However, this tool does not currently support real-time constraints, such as bounded interval time inherent in our case studies. For this, we have translated the model into the TAIO model for which we use UPPAAL model checker to verify real-time constraints. The XML-based implementation helps us for simplicity and rapid prototype development for our case studies. It is flexible for extension and maintenance. For reusability evaluation of our component-based approach, we realize the reuse cost estimation of components developed in order to analyze the cost benefit of our approach compared to ad-hoc reuse copy paste strategy used in certain railway companies.

1.4 Dissertation Roadmap

The dissertation is divided into five parts. While this introductory chapter is part of the first part, the second one encloses the State of Art. The third part presents the contribution of this dissertation, and the fourth one the validation of our proposal. Finally, the last part includes the conclusions and perspectives of this dissertation. Below, we present an overview of the chapters that compose the different parts.

Part II: State of the Art

Chapter 2: Component-Based Modeling and Observer-Based Verification. In this chapter, we give a brief introduction to the component-based development and verification. Since it is used throughout the dissertation, the idea of the chapter is to provide a better understanding of this background in which our work takes place, as well as the terminology and concepts presented in the later chapters.

Chapter 3: Traceability of Concerns. In this chapter, we list and describe some of the most relevant works related to traceability of concerns. We compare these related works using different criteria and highlight the benefits for component based modeling.

Part III: Contribution

Chapter 4: Component-Based Modeling with Traceability of Concerns. In this chapter, we present our approach for component-based modeling with traceability of concerns. In particular, we describe our approach meta-model, which is composed of a concern meta-model, a component meta-model and a traceability meta-model. We also propose a generic application process which can be used to instantiate the defined meta-models to assess change impact analysis.

Chapter 5: Observer-Based Verification with Patterns of Properties. In this chapter, we introduce a formal model of our component meta-model. In particular, we define component interface interoperability by using the theory of component contract. This formal model also facilitates the transformation into the timed automata model, for which we can use a timed model checker to apply observer-based verification, where property specification patterns are presented as observer automata. Finally, by using the composition theory of the timed input-output labeled transition system, we demonstrate that the defined observers have no impact on the behavior of the system under observation.

Part IV: Validation

Chapter 6: Validation. In this chapter, we describe the application details of our SARA approach through railway safety-critical use cases. We also report on some experiments we conducted to evaluate the case studies. This evaluation investigates in particular the soundness, the scalability and the practicality of our approach when dealing with component-based modeling and verification. Overall, as our cost empirical evaluation shows, we observe that SARA is well-suited to handle the modeling with traceability concerns and observer-based verification, while saving the cost for users who are looking at reusability for long term benefits.

Part V: Conclusion and Perspectives

Chapter 7: Conclusion and Perspectives. This chapter concludes the work presented in this dissertation. We summarize the overall approach and discuss some limitations that motivate new ideas and future directions as short-term and long-term perspectives.

1.5 Publications

We present below the list of research publications related to the work done while developing the approach described in this dissertation.

International Journal

- Marc Sango, Olimpia Hoinaru, Christophe Gransart, and Laurence Duchien. *A Temporal QoS Ontology for ERTMS/ETCS*. In International Journal of Computer, Control, Quantum and Information Engineering, 9(1):95 - 101, 2015.

Under Submission

- Marc Sango, Laurence Duchien, and Christophe Gransart. *Component-Based Modeling and Observer-Based Verification for Railway Safety-Critical Applications*. Extended version of FACS 2014 conference paper submitted to the special issue of FACS 2014, to be reviewed in Elsevier's Science of Computer Programming Journal. *Submission: February 2015*.
- Marc Sango, Christophe Gransart, and Laurence Duchien. *A Traceability Model Based on a Component-Based Model-Driven Approach: Application to Railway Real-Time Control Systems*. Major Revised version of article, submitted to the Journal of Systems and Software. *Revised version submission: April 2015*.

International Conferences

- Marc Sango, Laurence Duchien, and Christophe Gransart. *Component-Based Modeling and Observer-Based Verification for Railway Safety-Critical Applications*. In 11th International Symposium on Formal Aspects of Component Software (FACS'2014), pages 248-266, Bertinoro, Italy, September 2014.
- Marc Sango, Christophe Gransart, and Laurence Duchien. *Safety Component-Based Approach and its Application to ERTMS/ETCS On-Board Train Control Systems*. In Transport Research Arena (TRA'2014), pages 648-653, Paris, France, April 2014.

- Marc Sango, Olimpia Hoinaru, Christophe Gransart, and Laurence Duchien. *A Temporal QoS Ontology for ERTMS/ETCS*. In International Conference on Knowledge Engineering and Ontological Engineering (ICKEOE'2015), London, United Kingdom, January 2015.

Presentations and other Publications

- Marc Sango. *A Component-Based Model-Driven Approach with Traceability of Concerns: Railway RBC Handover Case Study*. In Young Transport Researchers Seminar (YRS'2015), Rome, Italy, June 2015. *Paper and presentation are online on the European Conference of Transport Research Institutes (ECTRI) website: <http://www.ectri.org/YRS15/Papers15.htm>.*
- Marc Sango, Laurence Duchien, and Christophe Gransart. *SARA Component Approach for the Development of Railway Safety-Critical Applications*. In 17th International ACM Sigsoft Symposium on Component-Based Software Engineering (CBSE'2014) Lille, France, July 2014, June 2014. *Poster presentation.*
- Marc Sango. *Application of SARA Approach to ERTMS/ETCS On-Board Train Speed Control Software*. December 2014, *Technical Report, IFSTTAR.*
- Marc Sango, Laurence Duchien, and Christophe Gransart. *Modèle de Défaillance lié à la Sécurité pour des Applications Ferroviaires Critiques - Développement à Base de Composants*. Journée GDR GPL, April 2013. *Poster presentation.*

Part II

State of the Art

Chapter 2

Component-Based Modeling and Observer-Based Verification

Contents

2.1 Introduction	15
2.2 CBSE Background	16
2.2.1 CBSE Definitions	16
2.2.2 CBSE Key Concepts	17
2.3 Focus on CBSE for Embedded System Design	22
2.3.1 Embedded System characteristics	22
2.3.2 Focus on Railway Embedded Real-Time Control Systems	23
2.4 V&V Background	24
2.4.1 V&V Definitions	25
2.4.2 V&V Applicability	25
2.5 Focus on Observer-Based Verification	30
2.5.1 Property specification patterns	30
2.5.2 Railway domain-specific property specification patterns	32
2.6 Comparative Analysis and Discussion	33
2.6.1 Comparative Analysis	33
2.6.2 Discussion	35
2.7 Summary	37

2.1 Introduction

In this chapter, we give a brief introduction to the component-based modeling and verification. The objective of this chapter is not to present an in-depth description of all the existing

concepts, approaches and technologies surrounding the Component-Based Software Engineering (CBSE), but to give a brief background in which our work takes place. Particular importance is given to the survey of CBSE approaches in the design of Embedded Safety-Critical Software (ESCS), specially railway ESCS, and to the observer-based verification of software model.

The chapter is structured as follows. Section 2.2 introduces the CBSE background. In Section 2.3, we focus on CBSE approaches for the design ESCS. Section 2.4 introduce the V&V background. In Section 2.5, we focus on a background of observer based verification. Section 2.6 describes the comparative analysis and discussion of state of the art work. Finally, Section 2.7 summarizes the ideas presented in this chapter.

2.2 CBSE Background

In software engineering, the CBSE [Bachmann et al., 2000, Heineman and Councill, 2001, Szyperski et al., 2002] is an important emerged topic. The CBSE primary aim is to compose systems from pre-built software units or components. Such an approach is a systematic approach that enables software reuse throughout the software development and management process with the ultimate aim, reduce production cost. To realize these aims, it is crucial to have software component model, which is the cornerstone of any CBSE approach [Lau and Wang, 2007]. Before comparing in Section 2.6 the approach category in which our work takes place, let introduce some definitions and concepts that are essential to understand the CBSE terminology.

2.2.1 CBSE Definitions

In the literature, there are several terminologies related to the CBSE. Here, we give the widely accepted definitions.

- **Szyperski et al. Definition:** "A software component is a *unit of composition* with contractually specified *interfaces* and explicit *context dependencies* only. A software component can be deployed independently and is subject to composition by third parties." [Szyperski et al., 2002].

The definition of Szyperski et al. is a widely accepted definition. It introduces the key concepts of software component, such as interface, composition and context dependencies defined in Section 2.2.2. However, this definition do not include a *component model* as in the definition of Heineman and Councill [Heineman and Councill, 2001].

- **Heineman and Councill Definition:** "A component is a software element that conforms to a *component model* and can be independently deployed and composed without modification according to a composition standard." [Heineman and Councill, 2001].

Lau and Wang defines the software component model as follow:

- **Software Component Model Definition:** "A software component model is a definition of (i) the *semantics* of components, that is, what components are meant to be, (ii) the *syntax* of components, that is, how they are defined, constructed, and represented, and (iii) the *composition* of components, that is, how they are composed or assembled." [Lau and Wang, 2007].

It is important to distinguish *component model* from *component framework*. They are quite often confused because the same name is used for both. For example, Fractal is some time used for component model and for its framework built in java or C language [Bruneton et al., 2006].

- **Component Framework Definition:** A framework, specially component-oriented framework, can be viewed as a generic structure and sometimes a standard that will cater a skeleton for developing software in a certain application domain [Pop et al., 2014].

For example, as the name AUTOSAR (AUTomotive Open System Architecture) indicates, AUTOSAR is an open and standardized automotive software architecture [AUTOSAR, 2006]. The AUTOSAR standard enables the use of a *component based software design model* for the design of a vehicular software. The design model uses application software components which are linked through an abstract component, named the virtual function bus. This virtual function bus, which is the conceptualization of all hardware and system services offered by the vehicular system, makes it possible for the designers to focus on the application instead of the infrastructure software.

2.2.2 CBSE Key Concepts

As emphasized in Section 2.2.1, the key concepts related to CBSE are *component*, *interface*, *connector* and *composition*. To illustrate graphically the definitions of these concepts, we use the UML2.0 component specification [UML2.0, 2005], which is a well-known graphical representation among the other graphical component representations synthesized in the Appendix of [Lau and Wang, 2007]. The key argument over the use of UML specification is the universality of this specification.

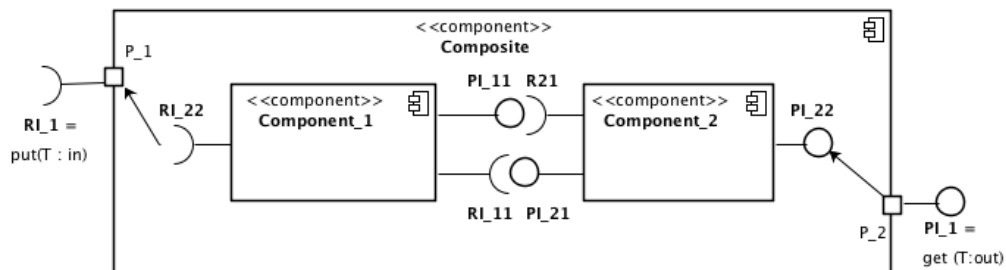


Figure 2.1: Components and component Composition in UML

Component

As shown in Section 2.2.1, a generally accepted definition of a software component is that it is a modular and reusable software unit (of a software system) that encapsulates implementation and exposes a set of *provided services* and *required services*. The provided services are services performed by a component. The required services are services needed by a component to produce its provided services. As illustrated in Figure 2.1, a component specifies its behavior by one or more required services, represented by sockets (e.g., RI_11) and provided services represented by lollipops (e.g., PI_21).

Current component models fall into two main categories [Lau and Wang, 2007]: (1) models where components are collections of objects, as in Object-Oriented Programming (OOP) and (ii) models where components are architectural units, as in software Architectures Description Language (ADL). A standard example of OOP category is Enterprise JavaBeans (EJB) [Rubinger and Burke, 2010], while a standard example of ADL category is CORBA Component Model (CCM) [CCM, 2006].

In current component models where components are objects in the sense of OOP, the *operations* of these objects are the provided services. Since they cannot specify their required services, these objects are usually hosted in a container, which handles interactions between components. A recent example is a component model used in [Panunzio and Vardanega, 2014]. As a result, the semantics of these components is an enhanced version of that of the corresponding objects. In particular, they can interact with one another via mechanisms provided by the container.

In current component models where components are architectural units, services can also be specified as *ports*, represented by square in Figure 2.1 (e.g., P_1). The port of one unit represents not only the provided service of that unit but also the required service of the other unit and vice versa. In some models, for example UML2.0 and CCM, ports for provided services are distinguished from those for required services.

Interface

An interface, either an operation-based or a port-based interface [Crnkovic et al., 2011], provides a syntactic information for an interaction point of a component [Chen et al., 2009]. In this way, an interface I implements its required and provided services. It consists of two parts: the data declaration section, $I.D$, that introduces a set of variables with their types, and the method declaration section, $I.M$, that defines a set of method signatures. Each signature is of the form $M(T1\ in; T2\ out)$, e.g., $Put(T1 : in)$ and $Put(T2 : out)$, where $T1$ and $T2$ are type names, *in* stands for an input parameter, and *out* stands for an output parameter.

This syntactic type information is obviously not enough for rigorous verification and validation. For this, some component models, such as rCOS component model [Liu et al., 2009], define the notion of *contracts of interfaces*. A contract of an interface is a

specification of the semantic for the interface. For example, if the component is to be used in a real-time application, the contract of its interface must specify real-time constraints, such as the lower and upper bounds of the execution time of a method.

Connector

A connector specifies a relationship that enables communication between two or more components. In most of component model, connectors are used for the composition of component at *design time*, where components have to be constructed, composed, cataloged, and stored in a repository. For example, in UML 2.0 component model [UML2.0, 2005], there are two kinds of connectors: (1) An *assembly connector* (lollipop in socket, see Figure 2.1) is used to connect the required interface of a component to the provided interface of another component; and a *delegation connector* (arrow, see Figure 2.1) is used to forward requested and provided services from the inside of a composite component to the outside. In Fractal component model [Bruneton et al., 2006], a connector is a *binding* component, *i.e.*, a component dedicated to the communication between other components.

Composition

Composition is a fundamental issue in component-based development since components are supposed to be used as building blocks from a repository and assembled or plugged together into larger blocks or systems. Theoretically, composition can take place during three stages of the *life cycle* of components [Crnkovic, 2002]: *design phase*, *deployment phase* and *runtime phase*. Figure 2.2 shows an idealized *component life cycle* with composition operators [Lau and Wang, 2007].

However, in practice, composition should be possible in both the design and the deployment phases of the component life cycle while the system is being constructed [Lau and Wang, 2007]. In the design phase, components have to be constructed, composed by using composition operators, and stored in a repository. In the deployment phase, components have to be retrieved from the repository and compiled to binary code, and then assemble them into a system by using composition operators.

Current component models have two kinds of composition mechanisms: *endogenous composition* and *exogenous composition* [Crnkovic et al., 2011]. The use of intermediary connectors corresponds to the concept of exogenous composition because the interaction between components is handled outside of the components themselves. In contrast to exogenous composition, endogenous composition refers to a binding without any intermediary connector. In this case, the handling of binding and interaction protocols is part of the components themselves, including its interfaces.

One important requirement for the application of composition in safety-critical system is to provide the ability to reason about composition. For this we need a composition theory as discussed in [Crnkovic et al., 2003]. However, as explained in [Lau and Wang, 2007],

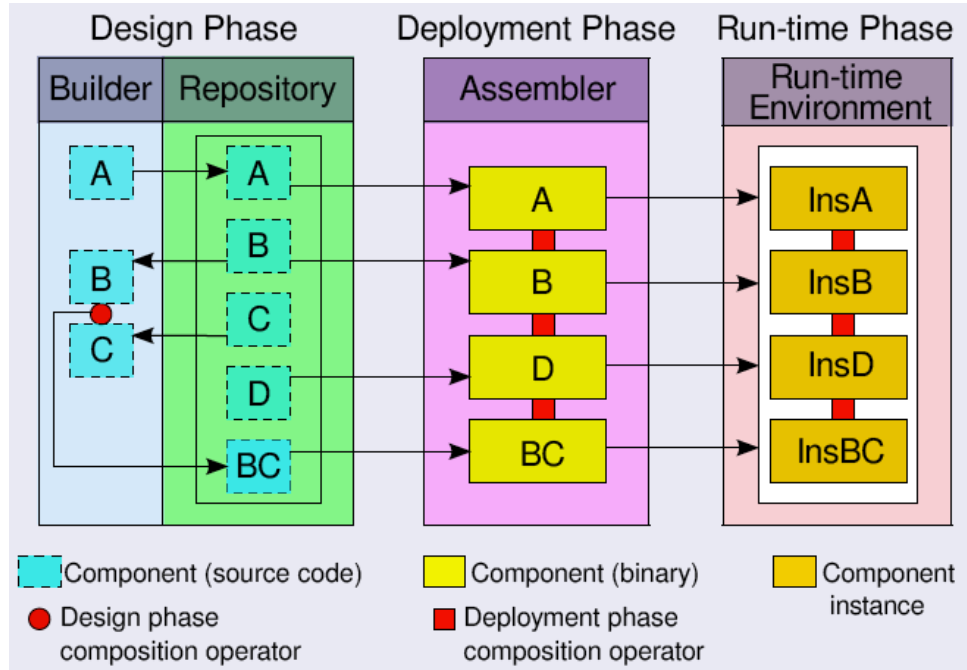


Figure 2.2: An idealized component life cycle

current component models tend not to have composition theories, even those with a composition language, such as ADL composition language [Allen and Garlan, 1997], Lumpe et al. composition language [Lumpe et al., 2003] and CoCo composition language [Tansalarak and Claypool, 2005]. One work with composition theory is the contract composition theory for reactive components [He et al., 2006]. Such a theory allows us to calculate and, thus, predict the result of applying a composition operator to our components.

Component and System Life Cycle

Another important characteristic of CBSE is the separation of the development processes of individual components, named *component life cycle*, from the development process of the overall system *system life cycle* [Crnkovic et al., 2006]. As illustrated in Figure 2.3, an idealized component life cycle entails an idealized system life cycle and should be separate from system life cycle [Lau et al., 2011].

One of popular system life cycles is the V Model, such as V-Lifecycle prescribed in the CENELEC standard of railway safety-critical software for control and protection systems [EN-50128, 2011]. V-Lifecycle has been adapted for component-based development, such as W-Lifecycle [Lau et al., 2011], Y-Lifecycle [Capretz, 2005], and X-Lifecycle [Tomar and Gill, 2010].

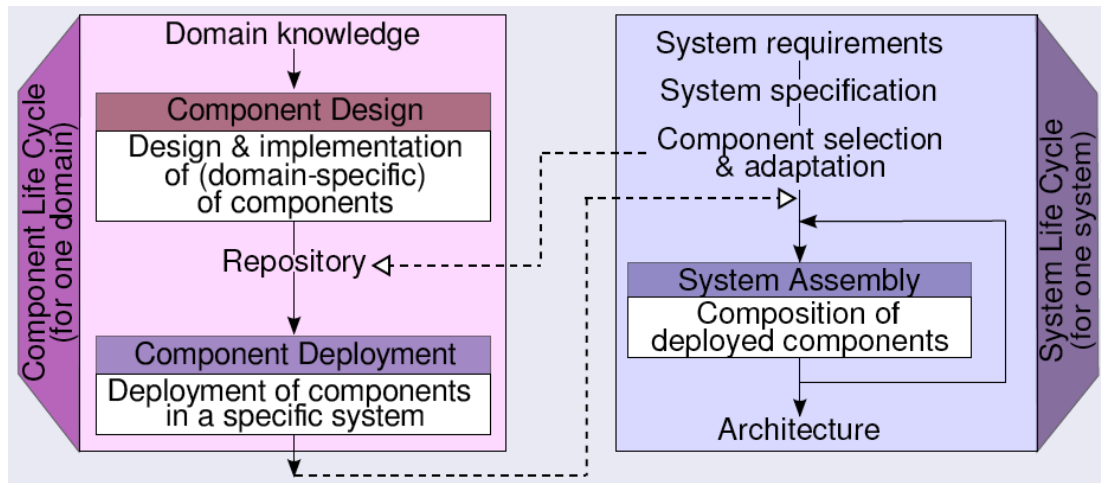


Figure 2.3: Idealised component and system life cycles

All of them have some strengths and drawbacks. For example, most of them consider that the complete system should be constructed with component based development. As a consequence, they still propose an idealized CBD lifecycle. They do not discussed in detail domain specific requirement issues. For example, due to train control application market structure (many manufacturers and vendors), it is very difficult to build the complete system by only using these idealized CBD lifecycles. Indeed, there are always some requirements that are customer specific and vary from customer to customer, and still developed with traditional software development process. As a consequence hybrid development process is required to support both component based software development as well as traditional software development.

What is incontestable in this domain is that applications still developed with V-model, but there are more and more focus on reusing existing software artifacts.

One of reusing existing software artifacts is the *Rational unified process (RUP)* [Kaur and Singh, 2010], which focus on reuse of existing classes on object oriented development. Another one is the *component-based model-driven development (CBMDD)* [Chen et al., 2009], which concentrates on the integration of CBD into a Model-Driven Development (MDD) [Thomas and Barry, 2003]. The advent of model-driven development, whose principles are to use models systematically at different phases of system development process and to increase the level of automation, provides a new landscape for dealing with some longstanding software development challenges, such as traceability management [Santiago et al., 2012].

2.3 Focus on CBSE for Embedded System Design

This section contains the brief information about the embedded safety-critical control software. Particular focus is given to train control systems. All these concepts are essential to understand the focus on domain-specific component-based approaches.

2.3.1 Embedded System characteristics

Embedded systems is everywhere in almost every domain of everyday modern life such as automobile, avionic and railway control system, mobile phones or small sensor/actuator controllers in industrial process and health sector, and much more [Marwedel, 2011]. The percentage of all computer systems belong to embedded systems today can be estimated at 98 % [ARTIST, 2004, Crnkovic, 2005]. In the following list, the general definition of embedded systems, research roadmap or challenges of embedded system design and the industrial needs are introduced.

- **IEEE Definition.** *“An Embedded Computer System is a computer system that is part of a larger system and performs some of the requirements of that system; for example, a computer system used in an aircraft or rapid transit system. (IEEE, 1992)”*.
- **Research Roadmaps.** The embedded systems usually must meet stringent specifications for safety, reliability, availability and other attributes of dependability [ARTIST, 2004]. As explained in the roadmaps for embedded systems design research [ARTIST, 2004] and CBSE for embedded systems [Crnkovic, 2005], most of such embedded systems are also characterized as *real-time systems*, which means that the real-time properties such as response time, worse case execution time, etc., are important design concerns. In addition, the increased complexity of embedded real-time systems leads to increasing demands with respect to requirement specification engineering, high-level design, early error detection, productivity, integration, verification and maintenance [Crnkovic, 2005].
- **Industrial needs.** Although the general characteristics is almost the same in different embedded domains, the adoption of CBSE approaches for embedded systems in a specific embedded industry will depend on the industrial needs, its methodology, its process, its technology and its market structures. For example, Table 2.1 summarises the main industrial needs and the concerns they impact on [Panunzio and Vardanega, 2010].

This table was derived from European Space Agency (ESA) initiatives to harmonized the methodology, process and technology concerns of software reference architecture around the agency missions. No relative priority was set on these industrial needs. It is just given to illustrated that the adoption of any software development approach, specially CBSE approaches in an industrial domain will depend on the special industrial needs. In the following we will specially focus on the railway domain.

Industrial needs	Impacts on
Reduced development schedule	Methodology
Product quality	Methodology, process, technology
Increased cost-effectiveness of software development	Methodology, process, technology
Reduced effort intensiveness of Verification and Validation	Methodology, process
Multi-team development and product policy	Process, technology, market structure

Table 2.1: The main industrial needs and the concerns they impact on

2.3.2 Focus on Railway Embedded Real-Time Control Systems

One of our concerns in this work is the application of CBD in the railway control systems, which belong to embedded systems family discussed in above Section 2.3.1. Train control systems have various characteristics, such reliability, availability, safety, maintainability, efficiency, and real time constraints, that are possessed by other transport control systems, such as aircraft or car control systems. However, train control system has some difference with other vehicle control as follows:

- **Many Control levels.** One of difference between train and car is that train control have many levels of control than car control. Most of time train consists of a number of wagons, so it is more complex in terms of controlling the communication, controlling the doors and brakes, and so on. Figure 2.4 illustrate some high-level of controls, such as *traffic control*, *wayside control* and *train control* [Johansson, 2001].

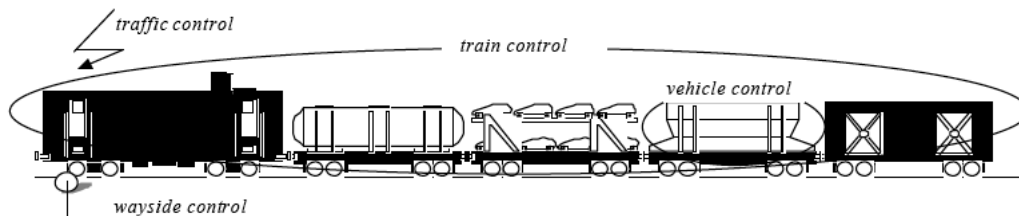


Figure 2.4: Different levels of control in a train control system

Traffic control system has the responsibility for route planning, navigation, signalling, surveillance and safety. It keeps track of all trains that might interact, their current position and destination. *Wayside control* is a set of systems, such as Radio Block Center (RBC) and interlocking, intended for surveillance, control and wireless communication. *Train control*, also known as train command and control or Automatic Train Control (ATC), is responsible for navigation, surveillance and safety. It is also responsible for the control and synchronizes all the computer based system in the car. The ATC responsibilities are to control and supervise the traction and auxiliary equipment, brake system and the driver's desk operation.

These different levels of a train control have different types of requirements, such

as functional, timing and dependability requirements. But at the same time dependability requirements can be similar to both, there could be large differences in functional as well as timing requirements and this makes the system more complex [Johansson, 2001]. In addition, the different parts of train are most of time built by different vendors and they have different requirements. All these factors make the train control system more complex to build and to inter-operate.

- **Interoperability.** Interoperability of the rail system within Europe is a key challenge to enable railway industries to capture the variety of rail market segments and to strengthen the competitiveness of rail products and operations [Collart-Dutilleul et al., 2014]. In the past, a number of different Automatic Train Control (ATC) systems has evolved in different countries at different times. As a consequence, when a train crosses a border, it needs to change its on-board signalling system for example. This generates an important financial cost. In order to establish international standardisation of ATC systems, the System Requirement Specification (SRS) of the European Rail Traffic Management System/European Train Control System (ERTMS/ETCS) is introduced [ERTMS/ETCS, 2014]. However, the management of railway signalling in ERTMS is based on local rules pertaining to each country and not on global rules. This makes it difficult to evaluate the system in terms of safety and then for certification.
- **Certification.** Railway or other transportation vehicles are an international matter. For example trains and cars cross borders daily and a single car may be shifted among different trains during its journey to the destination. This is an area where international agreements, standards and certifications are needed : *e.g.*, [EN-50128, 2011] for railway, and [ISO-26262, 2009] for automotive.

Meeting the strict safety requirements in critical software development is today crucial for the safety-related industrial environment, especially railways. To be able to prove that all safety properties are captured in the system requirements and software specifications, as well as that the final software product satisfies all specifications, a formal verification and validation is the most convenient and recommended.

2.4 V&V Background

Verification and Validation (V&V) are at the heart of the process of developing software for applications that require high dependability, such as the railway safety-critical control applications, discussed in Section 2.3.2. In this section, we give the background on general Verification and Validation (V&V) approaches and we focus on Observer-Based Verification (OBV) in which our approach work takes place.

2.4.1 V&V Definitions

Software V&V are independent activities that are used together for checking that a software system or a software component meets requirements and specifications and that it fulfills its intended purpose. Although the two activities are not the same thing, they are quite often confused. IEEE Standard Glossary of Software Engineering Terminology provides following definitions.

- **Verification:** Software verification is the process of evaluating software or software component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. [IEEE-Std-610, 1990].
- **Validation:** Software validation is the process of evaluating a software or software component during or at the end of the development process to determine whether it satisfies specified requirements. [IEEE-Std-610, 1990]

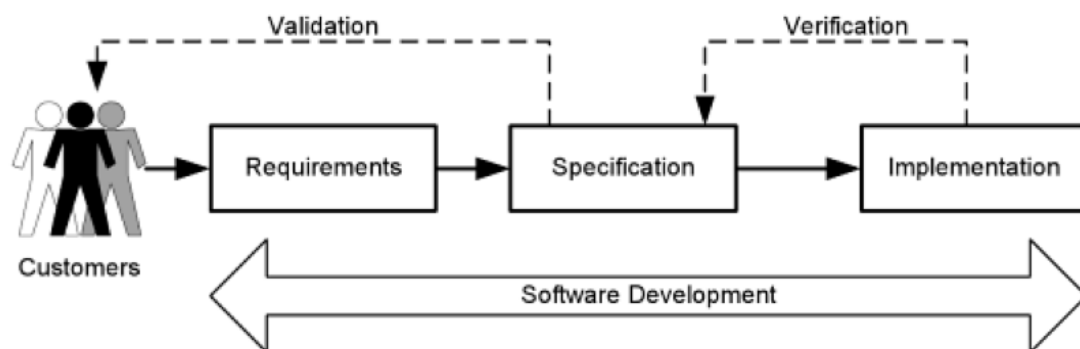


Figure 2.5: Verification and validation in software development

Figure 2.5 illustrates the basic relationships involved in V&V in software development process [Knight, 2012]. In other words, software verification is applied most commonly to show that an implementation implements a specification correctly, while its validation ensures that the software actually meets the customer requirements, and that the requirement specification was correct in the first place. Note that a specification is generally the result of requirements elicitation process in a domain [Pineiro, 2004].

2.4.2 V&V Applicability

For software V&V activities, there are numerous techniques and tools that may be used in isolation or in combination with each other. However, with an effort of classification, V&V techniques can be classified in five broad groups [Collofello and Institute, 1988]:

Software Technical Reviews

The software technical review process includes techniques such as inspections, walk-throughs and audits. Software technical reviews can be used to examine all the products of the software development and evolution process. In particular, they are especially applicable and necessary for those products not yet in computer processable form, such as requirements or specifications written in natural language and in proper documentation [Schneider et al., 1992].

Such documents need to be properly structured, in order to ease the proper understanding of requirements, which shall satisfy two main quality attributes: (i) requirements relatedness: each requirement is conceptually connected with the requirements in the same section; (ii) sections independence: each section is conceptually separated from the others [Ferrari et al., 2013]. As a consequence, the utilization of these techniques as V&V activities requires an additional level of V&V activities, such as requirement tracing.

Requirement Tracing

Requirement tracing is a technique for insuring that the product, as well as the testing of the product, addresses each of its requirements. There are several ways in which requirement tracing can be performed, as discussed in detailed in Chapter 3. The usual approach to performing requirement tracing uses traceability matrix, which can be classified in three types.

The first type of traceability matrix maps requirements to software modules. Such construction and analysis of this matrix can help insure that all requirements are properly addressed by the product and that the product does not have any superfluous capabilities. The second type of traceability matrix maps requirements to test cases or verified properties. Such construction and analysis of this matrix can help insure that all requirements are properly tested or verified. A third type of matrix maps requirements to their evaluation approach. The evaluation approaches may consist of review, simulation, testing and proof of correctness. This analysis shows that the requirement and evaluation tracing insures that all requirements will undergo some other form of V&V activities.

Simulation and Prototyping

Simulation and prototyping are techniques for analyzing the expected behavior of a product. There are many approaches for constructing simulations and prototypes that are documented in the literature [Emmelmann, 2003, Huang et al., 2011].

For V&V purposes, simulations and prototypes are normally used to analyze requirements and specifications to insure that they reflect the user's needs. Simulations and prototypes can also be used to analyze predicted product performance, especially for candidate

product designs, to insure that they conform to the requirements. It is important to note that the utilization of simulation and prototyping as V&V techniques requires that the simulations and prototypes themselves be correct. As a consequence, the utilization of these techniques requires an additional level of V&V activities, such as software testing or formal verification.

Software Testing

Software testing is the process of exercising a product to verify that it satisfies specified requirements or to identify differences between expected and actual results. There are several ways to test software because testing depends on the levels of testing, such as module testing, integration testing, system testing and regression testing [Myers and Sandler, 2004]. For all levels of testing, there are different techniques of testing, such as functional testing, structural testing, error-oriented testing and so on, which can be applicable [Myers and Sandler, 2004].

It is important to note that the utilization of software testing depends also on software development. For example in component-based software testing [Groß, 2005], testing refers to all activities that are related to component testing and application testing in the scope of a component-based development project because ideal software component is supposed to be reused in different context.

Software testing is one of software engineering disciplines used as a widespread validation approach in industry. For example earlier studies, such as [Beizer, 1990, NIST, 2002] estimated that testing can consume fifty percent, or even more, of the development costs. However, a recent detailed survey in software testing research points out some open challenges, such as compositional testing and some longstanding dreams, such as 100% automatic testing [Bertolino, 2007].

Proof of Correctness

Contrary to previous categories, the proof of correctness is a collection of techniques that apply mathematical rigor to help establish a variety of properties. This category of techniques is also often referred to as *formal methods* [Knight, 2012]. When applied carefully, formal methods are powerful because they can help us establish properties such as freedom from certain classes of faults. However, using the formal method is neither a *panacea* that solves all problems nor a *curiosity* that provides no value [Knight, 2012]. In this sense formal methods can always be supplemented with various other V&V techniques discussed above.

A formal method is an application of mathematics, usually discrete mathematics, in software engineering. As such, a formal method provides the software engineer (i) a *formal languages*, such as Z or Alloy language [Jackson, 2006], to replace much of the use of natural language in software artifacts, such as requirements, specifications and designs; and

(ii) a *formal verification* techniques to analysis software artifacts produced during software development. Formal verification brings the rigor of formal methods to the challenge of verification [Knight, 2012]. There are several formal verification techniques. In particular, we rely on two categories:

1. **Correctness by construction.** The *Correctness by construction* principle is first advocated by Dijkstra in [Dijkstra, 1972]: "Argument three is based on the constructive approach to the problem of program correctness", where program construction should follow the construction of a solid proof of correctness. Many decades later, the approaches to correctness by construction is promoted in different approaches. The three main approaches are:

- **Refinement approaches.** In construction of software using refinement, software is built by developing a series of transformations or refinements from the high-level abstract specification to a low-level concrete implementation [Knight, 2012]. Each refinement makes the initial formal specification more concrete, and transformation is continued until an executable implementation has been produced. The key of fault avoidance is that transformations are selected and applied carefully, and proofs are constructed to show that the properties of the input are maintained by the output.

The B Method [Abrial, 1996] is the most complete and most comprehensive instantiation of refinement approach category. Several powerful tools, such as Atelier B tools, have been developed to support the B Method. For instance, in France, the functional requirements of the SACEM system present in RER Line A in Paris were formally constructed in the B language [Guiho and Hennebert, 1990] as well as for the automatic train system of the metro line 14 which was the first driverless metro line in Paris [Behm et al., 1998].

- **Analysis.** In the construction of software using analysis, the software is built by developing a series of program increments using a fairly conventional manual development approach [Knight, 2012]. Design using procedural abstraction can be used to develop the necessary procedures and functions, data structures can be designed in manner that provides the semantics for the application, and so on. The key to the use of analysis is the availability of a mechanism to verify each of the increments that is applied during development.

The Correctness by Construction manifesto of Praxis High Integrity Systems [Hall and Chapman, 2002, Chapman, 2006] is the most complete practice of analysis approach category. The most extensive use of this static analysis approach is with SPARK language and toolset, such as [SPARK, 2014]. This original approach is a code source-based development, contrary to the following synthesis category approaches based on high-level language

- **Synthesis approaches.** In the construction of software using synthesis, the implementation in a high-level language is actually built automatically by an *application*

generator [Knight, 2012]. The application generator reads a formal specification and generates the application software automatically. In some application generators, the synthesis process is guided by humans, and in others synthesis is completely automatic.

One particular technique that has been developed is a *model-based development*, specially component-based model development. The concept is to develop a specification using a formal notation, and to have an application generator tool to translate that specification into an executable program, much as a compiler translates a high-level-language program into an executable program. Most often the specification language is a *domain-specific language*, designed to be used by domain experts and software engineers in a particular application domain. Many synthesis application are available, including Mathworks [Simulink, nd] and [SCADE, nd].

2. **Model Checking.** Model checking is an important formal techniques that is quite different from the other formal methods, which have been discussed above. At the heart of model checking is the idea that analysis can be carried out on a *model* of an artifact rather than the artifact itself, hence the name model checking. The most common application to software is in concurrent software. Concurrent software is difficult for humans to reason about and to analyze using formal analysis verification because of the non determinism that is inherent in concurrent programs [Knight, 2012]. Regarding the specification of real time properties, two categories of works have been proposed to specify properties.

- **Temporal Logics specification.** The first category is based on temporal logic. It provides most of the theoretically well-founded body of works, such as complexity results for different fragments of realtime temporal logics [Henzinger, 1998]: Temporal logic with clock constraints (TPTL); Metric Temporal Logic (MTL, MITL); Event Clock Logic; etc. The algebraic nature of logic-based approaches make them expressive and enable an accurate formal semantics. However, it may be impossible to express all the necessary requirements inside the same logic fragment if we ask for an efficient model-checking algorithm (with polynomial time complexity). For example, Uppaal [Larsen et al., 1997] choose a restricted fragment of TCTL with clock variables, while Kronos [Yovine, 1997] provides a more expressive framework, but at the cost of a much higher complexity. As a consequence, this category requires different model-checkers for each interesting fragment of these logics, and a way to choose the right tool for every requirement which may be impractical. However, this is not an issue limiting the large adoption. An important issue limiting the large adoption of model checking technologies by the industry is the difficulty, for non-experts, to express their requirements using the formal specification languages supported by the verification tools.
- **Pattern-Based Specification.** The second category is based on specification patterns. Patterns propose a user-friendly syntax which facilitates their adoption by

non-experts. In the seminal work of [Dwyer et al., 1998], property specification patterns are defined with temporal logics, such as LTL and CTL. “Globally, Q responds to P” is expressed as $AG(P \Rightarrow AF(Q))$ in CTL or $\Box(P \Rightarrow \Diamond Q)$ in LTL. In this case, there is no need to provide a verification approach because efficient model-checkers are available for these logics. This work on patterns has been extended for the real-time constraints. For example, [Konrad and Cheng, 2005] extends the patterns language with time constraints and give a mapping from timed pattern to TCTL and MTL. Another related work is [Gruhn and Laue, 2006], where the authors define observer automata (*observers*) based on Timed Automata for each pattern. For each pattern, observer automata, which can be applied directly in a timed model checking tool, are constructed. We call this approach an *Observer-Based Verification*. However, these primary works lack a formal framework for proving the correctness or innocuousness of their observers and they have not integrated their approach inside a model-checking tool chain. But, verification by means of observers has been experimented in different works, such as [Aceto et al., 2003, Bayse et al., 2005, Abid et al., 2014]

Summary

The V&V categories presented in this section are not meant to provide a partitioning, since there are some approaches that span categories. We give, a practical view of V&V approaches shown in the literature in order to focus in observer-based verification in which our work take place.

2.5 Focus on Observer-Based Verification

As shown in the of previous Section 2.4, V&V are at the heart of the process of developing software for applications that require high dependability, such as the railway safety-critical control application. However, one of important issue limiting the adoption of formal verification technologies by the industry is the difficulty, for non-experts, to express their requirements using the formal specification languages supported by the verification tools. As a consequence, we focus on pattern-based specification for which our approached is related.

2.5.1 Property specification patterns

Dwyer et al. [Dwyer et al., 1998, Dwyer et al., 1999] introduced several patterns applicable to software property specifications written in different formalisms, such as Linear-time Temporal Logic (LTL) [Manna and Pnueli, 1992], Computational Tree Logic (CTL) [Clarke et al., 1986] or Graphical Interval Logic (GIL) [Ramakrishna et al., 1996]. These property specification patterns consist of a set of *patterns* and a set of *scopes*. Patterns specify what

must occur and scopes specify when patterns must hold. More precisely, a scope indicates the portions of system execution in which a pattern should hold. Let P and Q represent a state or event, the five basic kinds of scopes can be defined as follows:

- *Global*: the pattern must hold during the complete system execution;
- *Before P*: the pattern must hold up to the first occurrence of a given P;
- *After P*: the pattern must hold after the first occurrence of a given P;
- *Between P And Q*: the pattern must hold from an occurrence of a given P to an occurrence of a given Q;
- *After P Until Q*: the same as "*Between P And Q*", but the pattern must hold even if Q never occurs.

Patterns are classified into occurrence patterns and order patterns. Details of these patterns can be found in [Dwyer, nd]. Occurrence patterns are used to express requirements related to the existence or lack of existence of some states or events during system execution. Order patterns are used to express requirements related to pairs of states or events during system execution. The important patterns can be briefly described as follows:

- *Absence*: P does not occur within a scope;
- *Universality*: P may occur throughout a scope;
- *Existence*: P must occur within a scope;
- *Bounded Existence*: P must occur at least, exactly or at most k times within a scope;
- *Precedence*: P must always be preceded by Q within a scope;
- *Response*: P must always be followed by Q within a scope, as illustrated in the *untimed response* Example 1.

Example 1. "It is always the case that if $P\{x = 0;\}$ holds, then $Q\{y = 1;\}$ **eventually** holds."

These original specification patterns are not adapted for the specification of real-time properties, since they do not support quantitative reasoning about real-time [Konrad and Cheng, 2005]. As a consequence, Konrad and Cheng [Konrad and Cheng, 2005] refer to these patterns as qualitative specification patterns and include a set of quantitative patterns for real-time specification. Their collection of patterns was classified into three broad categories of real-time properties. The *duration* category describes properties that can be used to place bounds on the duration of an occurrence. The *periodic* category describes properties that address periodic occurrences. The *real-time order* category describes properties that place time bounds on the order of two

occurrences. They also offer a structured English grammar to support both qualitative and real-time specification patterns, as illustrated in the *time-bounded response* pattern Example 2, which is a *real-time order* category of above *untimed response* Example 1. Following the observer representation of [Gruhn and Laue, 2006], the observe automata of Example 2 is shown in Figure 2.6

Example 2. “It is always the case that if $P\{x = 0;\}$ holds, then $Q\{y = 1;\}$ holds **after at most k time units**.”

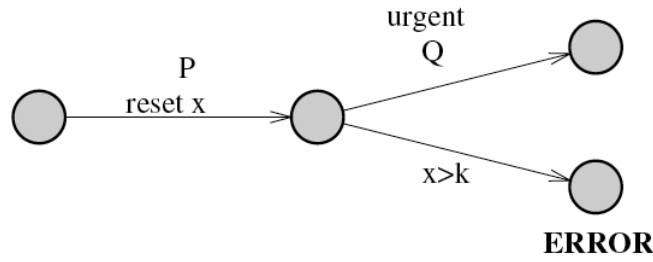


Figure 2.6: Observer of time-bounded response pattern

There are many other real-time extensions of the Dwyer et al. classification, such as [Chechik and Paun, 1999, Yu et al., 2006, Mekki et al., 2011, Abid et al., 2014] and other real-time specifications which are not directly based on the Dwyer et al. classification, such as [Bitsch, 2001, Sadani et al., 2005]. For example, [Bitsch, 2001] presents a classification comparable to the Dwyer et al. classification, but it is restricted only to safety patterns occurring in the specification of industrial automation systems. Recent studies realized by Bianculli et al. [Bianculli et al., 2012] and Abid et al. [Abid et al., 2014] have shown that Dwyer’s patterns with data-awareness and time delay-awareness patterns, as in Example 3, are the most used in practice in academia and industry.

Example 3. “Message m' must occur before a max delay of k time units after the first occurrence of message m .”

In this example, instead of directly considering states of processes as in Example 1 and 2, the requirements are expressed using two messages: the first message m represents the input of the process and the second m' its output. The delay between the two messages is constrained to be in the time limit. Our objective is to verify these real-time patterns in the railway safety-critical domain.

2.5.2 Railway domain-specific property specification patterns

We particularly focus on domain-specific rail-road temporal safety specifications. For instance, Example 4 shows a safety requirement specification of rail-road level crossing systems.

Example 4. “The gates must not be opened before the train has passed the rail-road crossing. The gates must be in the closed state for 6 seconds before the rail-road crossing has the status safeguarded. The safeguard of a level crossing is only permitted to be terminated, strictly after the rail-road level crossing has been completely vacated.” (Taken from [Bitsch, 2001]).

The specification of Example 4 explicits time duration before the safeguard of rail-road Level Crossing (LC). But, it does not consider the speed of trains that may cross the LC. Indeed, in most European countries, railway traffic has absolute priority over road traffic. So, the gate is opened to road traffic and then closed as soon as another train is detected. In this way, when another train is coming very shortly, the gate will be closed promptly after being opened. Consequently, car drivers in the crossing section may become stressed and make wrong decisions. The first sentence of Example 5 is stated to avoid this short opening duration between successive closure cycles. Conversely, it is also important to avoid keeping the gate closed for an unnecessarily long time when the train has passed. The second sentence of Example 5 constrains the closure duration.

Example 5. “When the gate is opened to road traffic, it must stay open at least T_{min} time units. Once closed and when there is no train approaching, the gate must be kept closed at least T_{begin} and at most T_{end} , where T_{begin} and T_{end} are the time interval limits prescribed.” (Taken from [Mekki et al., 2012]).

Remark 1. Note that, generally, these complex requirement specifications match more than one of patterns mentioned above. For example, the first sentence of Example 5 matches the after duration (stay open at least T_{min}) pattern and its second sentence matches time-interval (at least T_{begin} and at most T_{end}) pattern.

2.6 Comparative Analysis and Discussion

We have already introduced some related component-based approaches in previous sections. However, they have not yet been compared and discussed. In this section, we give a comparative analysis of some related work in Section 2.6.1. From this comparative analysis, we discuss in particular in Section 2.6.2 four requirements that are the heart of the development process of software that require high dependability, such as that railway safety-critical control software.

2.6.1 Comparative Analysis

Many software component models and frameworks have been proposed and compared over the last few years [Crnkovic et al., 2011, Pop et al., 2014]. The results of these evaluations show that there is no single winner: each of evaluated approaches has a strengths and limitations. As a consequence, they provide recommendations that can provide a guide in selecting a suitable component technology based on the specific requirements and target domain

of a considered application. According to this, two categories of component models have been distinguished: the *general-purpose* and the *specialized-purpose*. In addition, as shown in Section 2.2.2, current component models fall into two main categories [Lau and Wang, 2007]: (1) models where components are collections of objects, as in Object-Oriented Programming (OOP) and (ii) models where components are architectural units, as in software Architectures Description Language (ADL).

- Examples of general-purpose OOP-based approaches:
 - **EJB**: Enterprise JavaBeans [Rubinger and Burke, 2010]
 - **Fractal**: Fractal component model [Bruneton et al., 2006]
- Examples of general-purpose ADL-based approaches:
 - **AADL**: Architecture Analysis and Design Language [Feiler and Gluch, 2012]
 - **Pin**: Pin Component Technology [Hissam et al., 2005]
- Examples of specialized-purpose OOP-based approaches:
 - **Think**: Fractal model C-implementation [Fassino et al., 2002]
 - **Chess**: Chess project Component Model [Panunzio and Vardanega, 2010]
- Examples of specialized-purpose ADL-based approaches:
 - **ProCom**: Progress Component Model [Sentilles et al., 2008]
 - **IEC-61499**: IEC-61499 function blocks [IEC-61499, 2005]
 - **AUTOSAR**: AUTomotive Open System Architecture [AUTOSAR, 2006]

The specialized-purpose models for embedded and potentially also for real-time systems are sometimes related to *domain-specific* models. However, as discussed in Section 2.3.1, the domain of embedded real-time systems is relatively broad. They are ranging from small computational units to complex distributed control systems as in automotive, railway and airspace industry. Therefore, it is rather difficult to state universal criteria for comparison of component frameworks supporting development of such systems. Taking this into account, we do not focus on specific criteria that are important for component-based development in general, rather we focus on those that are necessary to meet the dependability requirements in component-based development for domain-specific embedded real-time systems. We particularly focus on *traceability*, *interoperability*, *V&V* and *Certification* criteria.

Table 2.2 shows a comparative table of approaches evaluated. We use a check mark ✓ if the approach proposes solutions or deals with the criteria, while a check mark between parenthesis (✓) indicates that the approach does not fully handle the criteria. Each of criteria is discussed in following Section 2.6.2.

Categories	Sub-categories	Approaches	Comparison criteria			
			Traceability	Interoperability	V&V	Certification
General-Purpose	OOP-Based	EJB		(✓)	(✓)	
		Fractal		(✓)	(✓)	
	ADL-Based	AADL	(✓)	(✓)	✓	(✓)
		Pin		(✓)	✓	(✓)
Specialized-Purpose	OOP-Based	Think		(✓)	(✓)	
		CHESS		(✓)	(✓)	
	ADL-Based	ProCom		(✓)	(✓)	
		IEC-61499		(✓)	✓	(✓)
		AUTOSAR	(✓)	✓	✓	(✓)

Table 2.2: Synthesis of comparison for some CBD-V approaches

2.6.2 Discussion

The above comparative analysis shows that with some exceptions, almost all of the discussed component models or frameworks have some support for interoperability and V&V but not all of them provide necessary tools to exploit this support completely. In contrast, the traceability and certification support is generally weaker in academic component models or frameworks mostly due to extensive costs to deal with this requirements.

R1. Traceability. Traceability of concerns is one of problems faced by developers during the development of components in safety-critical domains. However, as shown in Table 2.2, academic component models or frameworks evaluated do not explicitly representing and reasoning with traceability in their model. This is some time intentional. For example, for CHESS, [Panunzio and Vardanega, 2014] argue that traceability of model entities (inter-faces, components, etc.) to requirements shall also be managed externally (without embedding any information in the component model itself). As the others, they do not explicitly handle this concerns, may be due to extensive cost and time to deal with this requirement.

However, recent empirical research [Mader and Egyed, 2012] shows that system development practice with traceability is progressing, and it can be cost benefit in long term if it is adapted to the project-specific needs [Dömges and Pohl, 1998]. Contrary to the CBSE, the traceability of concerns has intensively studied in the MDE. As a consequence, in Chapter 3 we discussed the most relevant work related to traceability of concerns and highlight the benefits for component-based model driving development.

R2. Interoperability. The interoperability of components is a fundamental design desiderata of component-based development. With some exceptions, all of the approaches

discussed provide a partial interoperability, in terms of *composability*, *substitutability* and *reusability* *i.e.*, there is no need to know the design and the implementation when composing a component with other parts of the system, substituting a component with another one or reusing it in another application.

However, this is not sufficient in a specific domain, where domain knowledge, such as data format and a protocol of communication, is required for an effective interoperability. In addition, formal modeling of interfaces that enables such interoperability is necessary to reason and predict the application of composition, substitution and reusability mechanisms.

- R3. **V&V.** The V&V requirement is centered around the possibility of ensuring functional correctness as well as dependability properties *e.g.*, RAMS properties, *i.e.*, Reliability, Availability, Maintainability and Safety, by using one of mix V&V techniques shown in Section 2.4. Most of the discussed frameworks offer a formal execution model, *e.g.*,

- *Wright* for EJB [Sousa and Garlan, 1999];
- *Alloy* for Fractal and Think [Merle and Stefani, 2008];
- *Timed or Colored Petri Nets* for AADL [Renault et al., 2009];
- *UML state-charts* for Pin [Hissam et al., 2005];
- *Correctness by Construction* for CHESS [Panunzio and Vardanega, 2014];
- *Priced Timed Automata* for ProCom [Bures et al., 2008].
- *UPPAAL timed automata* for IEC-61499 [Soliman et al., 2012];
- *Simulink formalism* for AUTOSAR [AUTOSAR-Simulink, nd];

Consequently, tools for the particular formalism can be used.

However, this does not imply that these tools can be used without substantial additional effort to integrate them into the development process of the particular framework. For example, for Think, Alloy can also be used to check integrity constraints on finite architectures, but it is not directly integrated into ready-to-be-used V&V tools such as the Topcased IDE for ADL [Berthomieu et al., 2009].

- R4. **Certification.** This requirement facilitates certification of systems developed with the particular component model or framework (*e.g.*, IEC-61499 function blocks, AUTOSAR) according to corresponding safety standards (*e.g.*, IEC 61508 for general electronic devices, ISO 26262 for automotive systems). This is typically done by providing safety cases of the framework or process for certifying in a system using the component model or framework.

The certification support is generally weaker in academic component models or frameworks mostly due to extensive cost and the need for clear industrial application. Out of the frameworks considered, some certification support exists in case of AADL. AADL seems to be used mostly in avionics and space industry, accompanied by additional

development methodology which is compliant with DO-178B for safety of airborne systems. For Pin, there exist an approach [Chaki et al., 2007] for certification of components based on proof-carrying-code and certifying model checking.

2.7 Summary

In this chapter, we have briefly introduced a background of CBSE and V&V that states some principles and basic concepts we will use throughout the dissertation. Giving a short explanation of each one of them in Section 2.2 and Section 2.4, respectively, we have focused on CBSE for EDS in Section 2.3 and pattern observer based verification in Section 2.5. Then, the comparative analysis of some selected approaches is given in Section 2.6, by focusing particularly on *traceability*, *interoperability*, V&V and *Certification* criteria.

With some exceptions, all of the approaches discussed provide a partial support , in terms of interoperability, V&V. The traceability and certification support is generally weaker in academic component models or frameworks, mostly due to extensive costs and the need for clear industrial application. Regarding the traceability of concern, contrary to the CBSE, it has been intensively studied in the MDE.

As a consequence, in Chapter 3, we will discuss the most relevant work related to traceability of concerns and highlight the benefits for component-based model driving development.

Chapter 3

Traceability of Concerns

Contents

3.1 Introduction	39
3.2 Traceability Background	40
3.2.1 Traceability Definitions	40
3.2.2 Traceability Modes	40
3.2.3 Traceability of Concerns	42
3.2.4 Traceability of Functional and Non-Functional Concerns	44
3.3 Traceability Approaches in MDE	46
3.3.1 Description of Traceability Approaches	47
3.3.2 Summary of Traceability Approaches	48
3.4 Comparative Analysis and Discussion	48
3.4.1 Comparative Analysis	49
3.4.2 Discussion	54
3.5 Summary	56

3.1 Introduction

In this chapter, we give a brief introduction to the Traceability of Concerns (ToC). The objective of this chapter is to give a brief background in which ToC can take place in component-based development discussed in Chapter 2. Particular importance is given to the survey of ToC approaches in Model-Driven Engineering (MDE) in which component-based model driven development takes place.

The chapter is structured as follows. Section 3.2 introduces the traceability background. In Section 3.3, we review the survey of traceability approaches in MDE. Section 3.4 describes the comparative analysis and discussion. Finally, Section 3.5 summarizes the ideas presented in this chapter.

3.2 Traceability Background

The traceability is one of key activities in the development process of software systems. This is particularly important in safety-critical, such as railway control and protection software, because these systems emphasize high qualities in terms of Safety Integrity Level (SIL) [EN-50128, 2011]. The traceability has been defined and discussed in various domains.

3.2.1 Traceability Definitions

Several attempts have been done both from academic and industrial domains to define the main characteristics of traceability. We give below the most commonly used definitions.

- **IEEE definition** “Traceability is the degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another”. [IEEE-Std-610, 1990].

This definition is adopted in many standards of safety critical system, such the railway standard of software for railway control and protection systems [EN-50128, 2011].

- **Gotel and Finkelstein’s definition** “Requirement traceability refers to the ability to describe and follow the life of a requirement, in both a forward and backward direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases.)” [Gotel and Finkelstein, 1994].

Gotel and Finkelstein’s definition has become the common definition of requirement traceability. They make explicit that in order to follow (i.e., to trace) the life of a requirement you have to describe it.

- **Pinheiro’s definition** “Requirement traceability refers to the ability to define, capture, and follow the traces left by requirements on other elements of the software development environment and the traces left by those elements on requirements.” [Pinheiro, 2004].

Pinheiro uses two important aspects of requirement traceability to extend Gotel and Finkelstein’s definition. The first one is the ability to capture the traces we want to follow, and the second one is the ability to follow afterwards traces which should be viewed as natural occurrences.

3.2.2 Traceability Modes

There are several ways in which requirement tracing can be performed. As regards the direction of tracing, a requirement may be traced in a forward or backward direction; as regards requirements evolution, a requirement may be traced to aspects occurring before or after its inclusion in the requirement specification; and as regards the type of the objects involved, we may have inter or extrarequirements traceability [Pinheiro, 2004].

Backward and Forward Traceability

The definitions of forward and backward traceability have given in the literature as follow:

- **Forward traceability** is the ability to trace a requirement to components of a design or implementation. [Wieringa, 1995].
- **Backward traceability** is the ability to trace a requirement to its source, i.e. to a person, institution, law, argument, etc. [Wieringa, 1995].

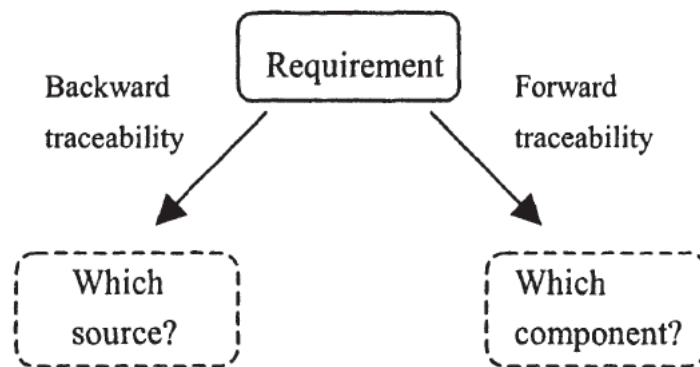


Figure 3.1: Backward and forward traceability

The concept of forward and backward traceability is intuitively illustrated in Figure 3.1 [Pinheiro, 2004]. For example, a requirement is traced forward when the requirement is changed and we want to investigate the impact of the change. On the other hand, a requirement is traced backward, for example, when there is a change and we want to understand it, investigating the information used to elicit the changed requirement.

Pre- and Post-Requirement Specification Traceability

We consider that a requirement is different to a requirement specification. Indeed, a requirement specification is the result of the elicitation process [Pinheiro, 2004]. The tracing of a requirement can be (1) a Pre-Requirement Specification (Pre-RS) traceability to get information related to the process of elicitation, prior to its inclusion in the requirement specification or a Post-Requirement Specification (Post-RS) traceability to get information related to its use, after the requirement has been elicited and included in the requirement specification.

- **Pre-RS traceability** refers to those aspects of a requirement's life prior to its inclusion in the requirement specification. [Gotel and Finkelstein, 1994].
- **Post-RS traceability** refers to those aspects of a requirement's life that result from inclusion in the requirement specification. [Gotel and Finkelstein, 1994].

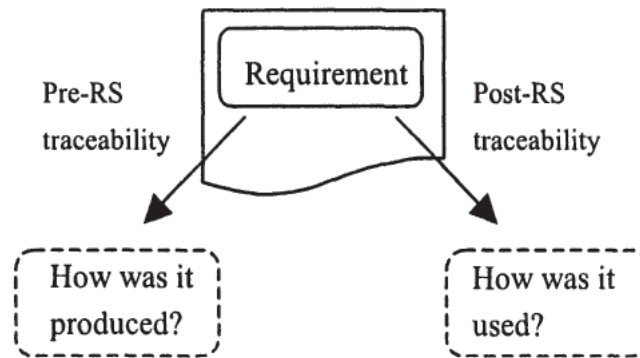


Figure 3.2: Pre-RS and Post-RS traceability

Figure 3.2 illustrates the definition of Pre-RS traceability and Post-RS traceability [Pinheiro, 2004]. For example, the Pre-RS traceability is useful when there is a change to a requirement and we want to get the requirement's sources or the people supporting it to validate the change. On the other hand, Post-RS traceability is useful to get the design module to which a requirement was allocated or the test procedures created to verify the requirement.

Inter and Extra-Requirement Traceability

The ability to trace the links between requirements is called inter-requirement traceability. The links between requirements and other artefacts are captured by extra-requirement traceability.

- **Inter-requirement traceability** refers to the relationships between requirements. [Pinheiro, 2004].
- **Extra-requirement traceability** refers to the relationships between requirements and other artifacts, such as specifications, diagrams and code [Pinheiro, 2004].

Figure 3.3 illustrates the definition of Pre-RS traceability and Post-RS traceability. For example, inter-requirement traceability is important for requirement analysis and to deal with requirement change and evolution. On the other hand, extra-requirement traceability is important for requirement refinement into design artifacts and artifact analysis in order to define new design artifacts or to change existing artifacts.

3.2.3 Traceability of Concerns

The traceability definitions given in Section 3.2.1 and traceability modes given in Section 3.2.2 should be relaxed to closely reflect the modes we want to trace in the chosen development process.

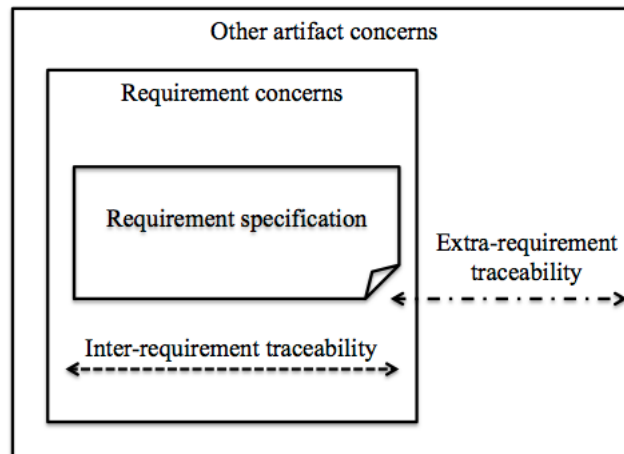


Figure 3.3: Inter and extra-requirement traceability

Firstly, the definition of traceability modes were defined as starting in a requirement. This is inappropriate since in many occasions the starting point of tracing is not a requirement. For example, the backward and forward traceability modes were defined as starting in a requirement and going backward or forward to other objects. For instance, if we get an unexpected result in a test procedure it may be necessary to trace the test procedure back to its related requirements or models in order to understand the results. The same happens if we have a change in some intermediate models, we may want to get all requirements already elicited from these sources to see if they remain valid.

For this reason, we prefer the use the term *Traceability of Concerns* where the definition of traceability modes were defined as starting in a concern, which is a particular set of information that has an effect on requirement or other development artifacts.

Secondly, the scope of traceability modes should overlap. For example, backward and forward tracing may happen at the same time as Pre-RS tracing or Post-RS tracing. When tracing a requirement forward to the code built to meet the requirement we do post-RS and forward tracing. In the same way, we do post-RS and backward tracing when we investigate an error, tracing a component back to the requirement it was intended to satisfy.

For this reason, we consider that the traceability of concerns includes inter-requirement and extra-requirement traceability which can be overlap with the three other traceability modes shown in Section 3.2.2.

Finally, the scope of traceability modes can depend on the development paradigm, *e.g.*, Aspect-Oriented Software Development (AOSD) [Kassab and Ormandjieva, 2006], Model-Driven Development (MDD) [Galvao and Goknil, 2007] and Software Product Line Development (SPLD) [Anquetil et al., 2010]. For example considering the MDD the traceability of concerns can be illustrated by Figure 3.4

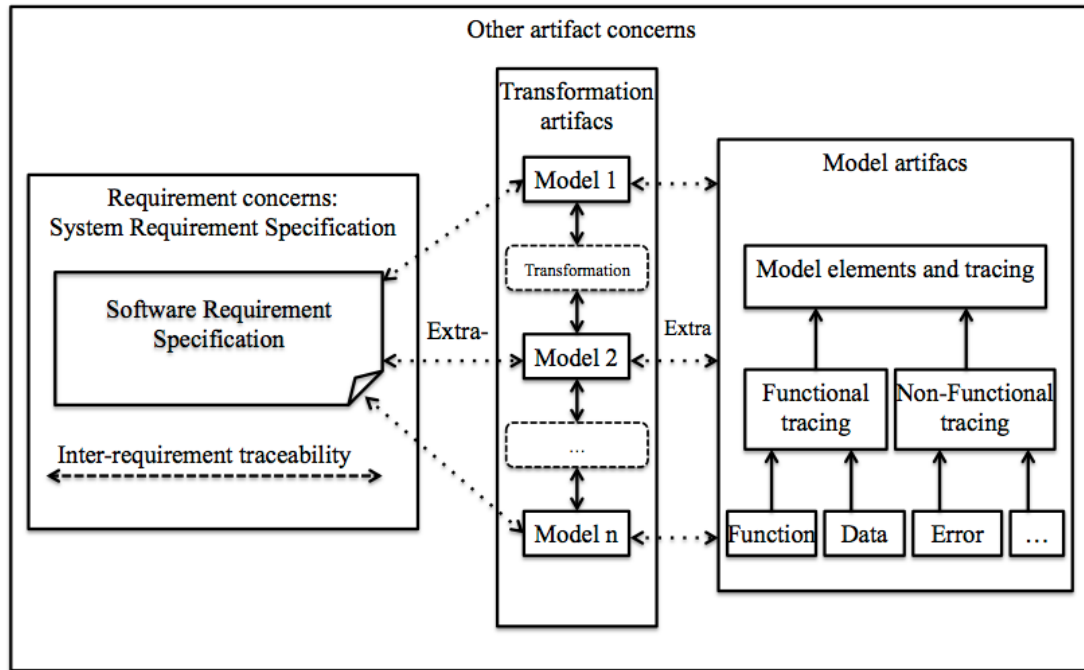


Figure 3.4: Traceability of concerns

To summarize, we focus on software development artifacts and define the traceability of concerns as following:

Definition 1. TRACEABILITY OF CONCERNS

Traceability of concerns refers to the ability to define, capture, and follow the traces from requirements concerns to the others concerns of the chosen software development artifacts, and the traces within or across those concerns.

3.2.4 Traceability of Functional and Non-Functional Concerns

We focus on the traceability of software functional and non-functional concerns. As illustrated in Figure 3.4, the functional tracing, such as function trace, state and data trace, is related to the functional concerns of software development, while, the non-functional tracing, such as error trace and performance trace, is related to the tracing of non-functional concerns of software development.

Traceability of Functional Concerns

Functional traces are those related to well established mappings between components or objects [Pinheiro, 2004]. Thus, it occurs naturally when well-defined models and notations

are used to describe components or objects. Indeed, the traces are derived from the syntactic and semantic connections or mappings prescribed by the models or notations used. Some examples in object-oriented models are the relationships between UML diagrams, classes, attributes and method. Note that traceability inside a model is sometimes called *vertical traceability* and between different models is sometimes called *horizontal traceability* [Lindvall and Sandahl, 1996].

Definition 2. SUFFICIENT CONDITION FOR FUNCTIONAL TRACING

Given a model, we can functionally trace an element represented in that model by following its syntactically and semantically defined relationships. The sufficient condition to functionally trace an element to another is that there is a mapping between the two elements.

For example, functional tracing properly answers questions like “what are the test cases assigned to this requirement”. In these cases it is sufficient to trace the requirement following its syntactically defined relations to test cases. On the other hand, if we are trying to answer the question “what are the tests which test this requirement”, this can be done in a functional way only if the semantics of the relationship is sufficient to guarantee that every test associated to a requirement is indeed a test for the requirement. Otherwise, we are dealing with non-functional traces and we have to resort to interpretation and analysis of the contents of the related components or objects.

Traceability of Non-Functional Concerns

Non-functional tracing is related to the tracing of non-functional concerns of software development. They are usually related to quality aspects and result from relationships to non-tangible concepts such as the definition of the reason, the purpose, the context or technical non-functional concerns, such as error or performance tracing. Regarding the number of non-functional concerns [Mylopoulos et al., 1992, Glinz, 2007, Chung and do Prado Leite, 2009], it is important to clearly identify the non-functional concerns we want to trace. As illustrated in Figure 3.4, we focus on error and performance tracing for temporal safety concerns.

Contrary to the sufficient condition for functional tracing shown in Definition 2, we cannot derive a sufficient condition for non-functional tracing. The first reason is that non-functional concerns cannot directly related to mapping between components as in functional concerns. Although, the translation of non-functional concerns into functional corresponding can help to automatically perform non-functional tracing, not all aspects of non-functional traces can be captured in a functional way [Pinheiro, 2004]. Moreover, it is not convincing that this is a complete solution because the complexity of the model may impair the tracing process. The second reason is that it is very difficult to decide, in fine-grained, which component carries traces of a non-functional requirement. For example, for a performance requirement, this amounts to ask which component is responsible for achieving the

required level of performance. There may be no definite answer in terms of single components. Non-functional requirements tend to be satisfiable only by an entire system and this satisfaction cannot be traced to a particular set of design elements[Mylopoulos et al., 1992]

One solution to facilitate traceability of non-functional concerns, even if it coarse-grained traces, is the efficient production and separation of functional and non-functional concerns during software modeling. Indeed, the production of traces and their capture for trace perception are as important as the traces themselves [Pinho, 2004]. Otherwise a trace model may be just too complex to be efficiently used.

As illustrated in Figure 3.4, in this dissertation, we rely on model driven engineering for dealing with traceability of functional and non-functional concerns. In the next section, we study the state of the art approaches for traceability in MDE.

3.3 Traceability Approaches in MDE

In this section, we survey different traceability approaches related to requirement and model driven engineering in order to observe the main techniques used for representing traceability information. Under this perspective, we selected in particular twelve traceability approaches encountered in the research literature:

1. Event Based Traceability (EBT) [Cleland-Huang et al., 2003]
2. Event Based Traceability with Design Patterns (EBT-DP) [Cleland-Huang and Schmelzer, 2003]
3. Reference Models for Requirements Traceability (RMRT) [Ramesh and Jarke, 2001]
4. Morphological Schema of Traceability (MST) [Konigs et al., 2012]
5. Behavior Tree Traceability (BTT) [Wen and Dromey, 2004]
6. Behavior Tree Traceability with Tree Merging (BTT-TM) [Wen et al., 2014]
7. Scenario Driven approach to Trace Dependency Analysis (SDTDA) [Egyed, 2003]
8. Precise Transformation Traceability Metadata (PTTM) [Vanhooff and Berbers, 2005]
9. On Demand Merging of Traceability (ODMT) [Kolovos et al., 2006]
10. Traceability Framework for Model Transformations (TFMT) [Falleri et al., 2006]
11. Model-Driven Traceability Approach for Software Product Lines (MDTA-SPL) [Anquetil et al., 2008]
12. Model-Driven Traceability Framework for Software Product Lines (MDTF-SPL) [Anquetil et al., 2010]

3.3.1 Description of Traceability Approaches

The traceability approaches that we have studied are listed in above Section 3.3. All the evaluated approaches have the capability to represent traceability information, but they used different techniques to represent traces or traceability links. Here, we summarize the main representation techniques used to represent traceability links:

- **Cross-References.** The cross-reference links are embedded as pointers in a text which may be an informal natural language text, a formal or a graphical specification. Links between diagrams can be viewed as cross-references. The use of cross-references is simple to understand. For example, a requirement specification is generally a document with many cross-references among parts of the document. Existing packages like LaTeX contain cross-referencing facilities as hyperlinks.

Cross-referencing is useful for written specifications but not for a concise representation of links, which can be done with traceability matrix [Wieringa, 1995]. Cross-references are often binary links, *i.e.*, 1-to-1 links. In this case links of higher arity links, *i.e.*, m-to-n links, cannot be easily represented.

- **Matrix.** A simple way to represent links between elements is a two-dimensional grid or matrix in which the horizontal and vertical dimensions list the elements that can be linked. The entries in the matrix represent links between these elements. The elements in both dimensions may or may not be the same.

An advantage of the matrix representation is that it is easy to understand. It provides a format that can be discussed by stakeholders with different backgrounds. However, in two-dimensional traceability matrix, only finite 1-to-1 links, between elements can be represented. However, some links may be m-to-n links.

- **Tree-Based.** Beyond 1-to-1 traceability links, a tree or a graph provides a means to represent m-to-n links. Indeed, trace information forms a tree where links may relate more than two vertices. Although a tree-based representation is a very natural and intuitive graphical representation, it is more challenging than a matrix presentation.

However, contrary to 1-to-1 traceability links, m-to-n links can help to reduce the number of required links and therefore simplify the adaptation.

- **Meta-Model.** Links between elements can also be represented by entity-relation models. An abstraction of these entity-relations models is called meta-model. In this case, the linked elements are entities and the links are relationship instances.

The advantage is that links with arity higher can be represented. Moreover an meta-model of links can be implemented using any database or XML technology. The use of database or XML technology has the advantage that ad-hoc query and reporting facilities are easily available.

- **Event-Based.** Contrary to traceability meta-model where traceable elements are directly linked by entity-relations, in traceability event-based subscription traceable elements are no longer directly related, but linked through publish-subscribe relationships. Instead of establishing direct and tight coupled links between elements, links are established through an event service, through an event-based server

The advantage is that generation and maintenance of traceability link is automatic. However, when the project grows, the most difficult problem is to maintain a good performance of the event-based server.

3.3.2 Summary of Traceability Approaches

Table 3.1 summarizes the traceability approaches evaluated. The first row contains the traceability approaches under study, listed in Section 3.3. The first column contains the traceability representation techniques derived in Section 3.3.1. We use a check mark ✓ if the approach proposes solutions or deals with the different representation techniques.

Traceability Representation Techniques	Traceability Approaches											
	EBT	EBT-DP	RMRT	MST	BTT	BTT-TM	SDTDA	PTTM	ODMT	TFMT	MDTA-SPL	MDTF-SPL
Cross-Reference							✓		✓			
Matrix											✓	✓
Tree-Based		✓		✓	✓	✓	✓				✓	✓
Meta-Model			✓	✓				✓	✓	✓		
Event-Based	✓	✓										

Table 3.1: Summary of some traceability approaches.

3.4 Comparative Analysis and Discussion

We have already introduced some traceability approaches in Section 3.3. Then, we have summarized in Table 3.2 the evaluated approaches by focusing on the main representation techniques used to represent traceability links. However, they have not yet been compared and discussed. In this section, we give a comparative analysis of these traceability related works

in Section 3.4.1. From this comparative analysis, we discuss in particular in Section 3.4.2 four open issue traceability requirements that we found to be important for flexibility and adaptability of traceability to project-specific needs.

3.4.1 Comparative Analysis

In [Galvao and Goknil, 2007, Konigs et al., 2012, Anquetil et al., 2010], interesting related works are presented where the authors review the most recent advances on technologies to automate traceability and discuss the potential role that Model Driven Development (MDD) can play in this field. For example, in [Galvao and Goknil, 2007], a survey on traceability approaches in MDD and the elaboration of a traceability taxonomy is presented. Based on this survey and recent related works [Wen et al., 2014, Konigs et al., 2012, Anquetil et al., 2010], we compare the evaluated approaches listed in Section 3.3. There are several criteria that could be used to evaluate traceability approaches in software engineering [Knethen and Paech, 2002, Winkler and Pilgrim, 2010]. As our proposal is based on the CBMDD, which is a particular component-based MDD approach, here, we present a comparative analysis of selected approaches with respect to the common comparison criteria of Galvao et al. [Galvao and Goknil, 2007]: *representation*, *mapping*, *scalability*, *change impact analysis*, and *tool support*.

The *representation* criterion compares the approach’s capability to represent traceability information. The *mapping* criterion analyzes whether the approach is capable of generating traces among the models at different levels of abstraction. The *scalability* criterion analyses whether the approach can be efficiently applied to large-scale systems. The *change impact analysis* criterion evaluates whether the approach provides support for determining the impact of changes on the artifacts across the software development lifecycle. Finally, the *tool support* criterion evaluates whether the approach provides any tool support for facilitating traceability.

Table 3.2 shows a comparative table of approaches evaluated. We use a check mark ✓ if the approach proposes solutions or deals with the scalability, change and tool support criteria, while a check mark between parenthesis (✓) indicates that the approach does not fully handle the criteria. In particular, regarding representation criterion, the approaches are distributed into two *categories*. Regarding mapping criterion, we observe, in particular, the support to *intra*-level relationships (traces among artifacts of the same abstraction level), *inter*-level relationships (traces among artifacts of different abstraction levels), or both *intra* & *inter*-level relationships. Each of criteria is discussed in following paragraphs.

Representation Criterion

For representation criterion, all the evaluated approaches have the capability to represent traceability information, but in different abstraction levels. For this

Categories	Sub-categories	Approaches	Comparison criteria			
			Mapping	Scalability	Change	Tool
Requirement-Driven	Event-based	EBT	inter		✓	✓
		EBT-DP	intra & inter		✓	(✓)
	Reference-based	RMRT	intra & inter	✓	✓	✓
		MST	intra & inter	(✓)	✓	✓
	Behavior tree-based	BBT	intra & inter	(✓)	✓	(✓)
		BBT-TM	intra & inter	✓	✓	✓
Model-Driven	Model-based	SDTDA	intra & inter	✓	(✓)	(✓)
		PTTM	intra & inter	(✓)	✓	
	Transformation-based	ODMT	inter	(✓)		✓
		TFMT	intra & inter			✓
	Product-based	MDTA-SPL	intra & inter		(✓)	
		MDTF-SPL	intra & inter	✓	✓	✓

Table 3.2: Comparison of some traceability approaches

reason, the evaluated approaches are distributed into two categories, see Table 3.2. The first category, the *requirement-driven* approaches use requirement models as abstractions to represent traceability information and to guide their traceability methods. In this category, we consider three sub-categories: the *event-based* (e.g., [Cleland-Huang et al., 2003, Cleland-Huang and Schmelzer, 2003]), *reference-based* (e.g., [Ramesh and Jarke, 2001, Konigs et al., 2012]) and *behavior tree-based* (e.g., [Wen and Dromey, 2004, Wen et al., 2014]). The second category, the *model-driven* approaches, according to the MDE paradigm, are interested in how meta-models, models and transformation frameworks are involved in traceability of any concerns, including inter and extra requirement traceability. This category is classified in two sub-categories the *model-based* (e.g., [Vanhooff and Berbers, 2005] and our approach) and the *transformation-based* (e.g., [Kolovos et al., 2006, Falleri et al., 2006]), which uses model transformation mechanisms for generating trace information.

Mapping Criterion

As already defined, the mapping criterion evaluates whether the approach supports traceability of model elements at different levels of abstraction. A comparison of the evaluated approaches from the perspective of *intra* or *inter* relationships is presented in the “mapping column” of Table 3.2.

Both *event-based* traceability approaches [Cleland-Huang et al., 2003] and [Cleland-Huang and Schmelzer, 2003] use event-based mechanisms to support the in-

direct mapping from requirements to other artifacts. This indirect mapping is the inter-level relationship. Indeed, in this method, requirements and other traceable artifacts, such as design models, are no longer directly related, but linked through publish-subscribe relationships. Based on design patterns, [Cleland-Huang and Schmelzer, 2003] approach also supports intra-level traceability between various non-functional requirements. In the *reference-based* traceability approach [Ramesh and Jarke, 2001], the proposed low and high meta-models support intra-level and inter-level traceability provides mappings between requirements and many other elements (system objectives, system components, functions, etc). In [Konigs et al., 2012], a morphological schema of traceability is proposed for systems engineering practice in the automotive industry. The application of the schema focuses on intra or inter trace links among system-oriented line and product-oriented line. The *behavior-based* traceability approaches [Wen and Dromey, 2004, Wen et al., 2014] support intra and inter-level traceability. In fact, they distinguish between two kinds of traceability: horizontal traceability and vertical traceability. The horizontal traceability is the traceability between different software artifacts within the same version of software, while the vertical traceability is the traceability in the same design artifact in different versions of software in order to enable evolutionary changes from one version to the next version. While the behavior-based traceability approach is designed specifically for the behavior engineering [Dromey, 2003], the approach can be supported in other software engineering approaches, such as model-based approaches. For example, Kim et al. propose a transformation from Behavior Tree models to UML state machines as a formal path from natural language requirements to an executable model of the system [Kim et al., 2012].

In the *model-based* traceability approach [Vanhooft and Berbers, 2005], Vanhooft and Berbers define UML profiles based on a meta-model that gives support to the model transformation between source and target model elements. As the proposed transformation traceability meta-model is mapped to UML profiles, links among various UML model elements can be traced. In our approach, intra-level and inter-level links are explicitly defined within or across the different levels of component model abstraction independently of any model transformation mechanisms. In the *transformation-based* approach [Kolovos et al., 2006], traceability information is stored in separate trace models which can be merged with a corresponding primary model from which annotated models with traceability information are generated. This approach uses external traceability links, i.e., inter traceability links, and adopts EML trace meta-model as a merging language for the generation of annotated models. In [Falleri et al., 2006], the trace meta-model and the transformation framework are proposed to enable the automatic tracing of model transformation in the form of transformation chains. Depending on the definition of source and target models, the approach allows intra-level and inter-level traceability links. However, there is no evidence of scalability, as discussed in the following paragraph. In the *product-based* approach [Anquetil et al., 2008], the authors have looked at the interaction of Traceability, Model Driven development and Software Product Line. Then, in [Anquetil et al., 2010] a Model-Driven Traceability Framework for Software Product Lines is proposed. The framework is based on traceability meta-model which defines traceable artifacts and trace links between these artifacts.

Scalability Criterion

Since real software projects are naturally larger during their development or become larger during their evolution, scalability is an important criterion to be considered when evaluating the usage of traceability approaches. The “scalability column” of Table 3.2 shows if the evaluated approaches totally \checkmark support, partially (\checkmark) support or do not support scalability.

As the *event-based* traceability approaches [Cleland-Huang et al., 2003, Cleland-Huang and Schmelzer, 2003] are based on an event server, which manages some links between the requirement and its dependent artifacts, the most difficult problem when the project grows, is to maintain a performance of the event server. However, this performance has not been discussed in these papers. In this way, the scalability of these approaches is questionable. In [Cleland-Huang and Schmelzer, 2003], there is no evidence that the method can be applied with success to support more complex design patterns of non-functional requirements. The reference models described in the *reference-based* approach [Ramesh and Jarke, 2001] may be scalable due to possible use for traceability activities in different complexity levels. Indeed, Ramesh and Jarke [Ramesh and Jarke, 2001] follow an empirical approach and focus interviews conducted in software organizations to study a wide range of traceability practices. As a result of this work, the authors constitute reference models that include the most important kinds of traceability links for various software development elements. In the *behavior-based* approach [Wen et al., 2014], two case studies are introduced to illustrate the concept and the scalability of the traceability model. The process to merge different Design Behavior Trees (DBTs) and generate an Evolutionary Design Behavior Tree (EvDBT), is a core part of the traceability model. They have proven that there is a way to compute the versioned tree merging algorithm in $O(n \log n)$ time. This result is important because it shows that the tree merge algorithm is at least theoretically scalable.

In the *model-based* traceability approach [Vanhooff and Berbers, 2005], the authors do not present a practical application of their approach. Therefore, it may be scalable because it is associated with the UML largely accepted and used. In our approach, we investigate the scalability issue in two directions. First, our approach has been shown scalable in different case studies due to its possible use of traceability activities in different component-based abstraction levels to deal with different complexity levels. Second, we have studied the empirical scalability of our approach for large-scale systems by increasing the complexity of each case study. We have concluded that the environment context awareness is one of the promising approaches to deal with scalability for large scale systems. In the *transformation-based* approach [Kolovos et al., 2006], the on-demand merging of traceability links with models may be a scalable approach due to the fact that the traceability information is maintained in a separate model and a generic trace meta-model is used for flexibility reasons. Contrary to our approach, in [Falleri et al., 2006], the authors implementation of trace generating code is tangled with the transformation code. This makes the traceability mapping table less reusable. In addition, in this paper, there is no evidence of scalability of this approach with relation

to larger projects with longer transformation trace chains. Despite that, the defined meta-model may be extendable and scalable. In the *product-based* approach [Anquetil et al., 2010], the relational database option is used to deal with scalability contrary to several approaches including ours where scalability is based on whether XML scales up nicely or not.

Change Impact Analysis Criterion

With the *change impact analysis criterion*, we want to evaluate if an approach provides support for determining the effect of change on the entire system or on the artifacts across the software development lifecycle. The “change column” of Table 3.2 shows if the evaluated approaches totally ✓ support, partially (✓) support or do not support scalability.

The event-based traceability approaches [Cleland-Huang et al., 2003, Cleland-Huang and Schmelzer, 2003] support change impact analysis. In the first approach, the event server manages some links between the requirement and its dependent artifacts by using some information retrieval algorithms. The second approach supports the identification of critical elements that should remain in the system in order to keep the integrity of a traceable non-functional requirement. The *reference-based* approach described by Ramesh and Jarke [Ramesh and Jarke, 2001] provides a means to analyze change impacts according to the description of the rationale submodel. The *behavior-based* approach [Wen et al., 2014] offers a new approach to software change impact analysis at the architecture levels. It differs from other change impact analysis methods, such as [Zhao et al., 2002], in the way that the traceability model handle vertical traceability by merging multiple versions of a software system and by tracing the design evolution back to the requirement evolution.

Contrary to our approach, the other evaluated *model-based* approach [Vanhooff and Berbers, 2005] and the *transformation-based* approaches [Falleri et al., 2006, Kolovos et al., 2006] do not provide any mechanisms for performing change impact analysis. In our approach, we have implemented a set of intentional trace queries that can trace requirement concerns to component elements and vice versa. To assess the impact of change, we use forward tracing queries, which trace the architectural elements starting from a given set of concerns. Backward queries are also used to inspect the set of concerns that an architectural element is related to. In the *product-based* approach [Anquetil et al., 2010], a trace query instance provides means to perform specific queries on a set of trace links and artifacts. It uses the framework basic query capabilities to execute more complex and powerful queries such as feature interaction detection and change impact analysis.

Tool Support Criterion

Tool support is fundamental for a good application of a traceability method, not only for visualization and management of generated traces among software artifacts, but also for the proper support for reasoning on this information. The evaluated approaches are compared

from the perspective of the provisioning of any (hypertext or graphical editors) tool support in the “tool column” of Table 3.2.

The components of the event-based traceability approach [Cleland-Huang et al., 2003] were implemented as client-server architecture based on observer design patterns. The event trigger was implemented on top of the DOORS requirements management system to manually capture change events as they occurred. Cleland-Huang and Schmelzer [Cleland-Huang and Schmelzer, 2003] support the static and dynamic generation of traceability links across the development phases, although only a few characteristics of their approach are fully implemented. The reference meta-models described in the *reference-based* approach [Ramesh and Jarke, 2001] were encoded in a knowledge-based meta database management system called ConceptBase [Jarke et al., 1995]. The efficiency of the tools which have implemented these metamodels was not evaluated. However, they were adopted later in several commercial traceability tools, such as SLATE and Tracenet. In the *behavior-based* approach [Wen et al., 2014], the case studies are investigated using a software environment called Integrare [Wen et al., 2007]. “Integrare” integrates a number of different software tools to support the BE approach.

In their model *transformation-based* approach, Kolovos et al. use Epsilon Merging Language (EML) [Kolovos et al., 2006] to implement the merging of models with traceability links. From a tool-support perspective, the execution engine of EML supports the Eclipse Modeling Framework (EMF) and Meta-Object Facility (MOF) models, as well as XML documents. In [Falleri et al., 2006], Falleri et al. have implemented the transformation chain trace meta-model in a model-oriented language compatible with EMF called Kermeta [Triskel project (IRISA), nd]. The *product-based* approach [Anquetil et al., 2010] has also adopted the EMF model-driven framework to implement their traceability framework for software product lines. In the *model-based* traceability approach [Vanhooff and Berbers, 2005], the authors do not relate the existence of a tool that gives support to their approach. In contrast, our approach has a partial tool support. Indeed, we have implemented concern traceability meta-model in the XML-based tool for the representation of the models and we use XQuery queries for the inference of tracing information. This renders our approach practical for models created with any modeling tool that has XML exporting capabilities. However, our approach does not yet support a graphical user interface. We are therefore working on enhancing the XML-based tool with a more visual representation of the results to the user.

3.4.2 Discussion

From above Section 3.4.1 comparative analysis, where the regular traceability requirements (representation, mapping, scalability, change and tool) are discussed, here we discuss four main requirements (the *separation* of concerns [Hirsch and Lopes, 1995], the *adaptability* of software artifacts to projects specific needs [Dömges and Pohl, 1998], the *uncertainty* on software development [Aizenbud-Reshef et al., 2006] and the *abstraction* of design decisions

[Anquetil et al., 2008]) that we found to be important to facilitate the maintenance of traceability for flexibility and adaptability of traceability to project-specific needs, such as the impact of uncertainty on safety-critical project.

R5. Separation. The separation of concerns is a fundamental design principle for separating a particular set of information, named concerns. Since Dijkstra [Dijkstra, 1982a], it is a long known best principle of software development, but sometimes much neglected in practice. It consists in separating different software concerns through software development activities in order to enable separate reasoning for each of focused specification. The extent of separation has obvious benefit for traceability of concerns and the reuse of the same functional concerns under different non-functional concerns.

However, experience shows that this benefit is much more difficult to achieve than it may seem [Panunzio and Vardanega, 2014]. Indeed, it requires substantial effort to arrive at common and stable specifications, clean interface design and effective consolidation of functional and non-functional part of system components.

R6. Adaptability. Adapting traceability environments to project-specific needs is a fundamental requirement of any traceability framework when considering project cost and time [Dömgies and Pohl, 1998]. A few of evaluated approaches, such as [Konigs et al., 2012] adapt traceability to automotive project specific needs. Recent empirical research [Mader and Egyed, 2012] shows that system management practice with traceability is progressing.

However, the same studies also point out that full capture of all conceivable traces according to these advanced models is neither desirable nor feasible when considering project cost and time. In this sense, if requirement traceability is not customized it can lead to an unwieldy mass of unstructured and unusable data that will hardly ever be used. The adaptation of trace capture and usage to project-specific needs is a prerequisite for successfully establishing traceability within a project and for achieving a positive cost-benefit ratio [Dömgies and Pohl, 1998].

R7. Uncertainty. Uncertainty plays a role in any system that needs to evolve continuously to meet the specified or implicit goals of the real world [Lehman and Ramil, 2002]. In safety-critical systems, the impact of uncertainty on software development can be even more severe than traditional software systems. It is for this reason, in development process of safety-critical systems, the Safety Integrity Level (SIL) is required. For example, the railway standard [EN-50128, 2011] address five Safety Integrity Levels (SIL), where 0 defines the lowest and 4 the highest one. The value of SIL defines the acceptable probability of Tolerable Hazard Rate (THR), which means a dangerous failure per hour in continuous operation. A system with safety function must keep THR between $10^{-5} f/h$ and $10^{-9} f/h$.

Under this perspective, traceability of design decisions in safety-critical software development is an important and relevant issue, as these are key points where uncertainty influences the design process. For performing traceability in the presence of

uncertainty, the focus of attaching additional information to traceability links, such as the rationale for its creation and the confidence we have in this rationale, should be one of the solutions to resolve design decisions [Anquetil et al., 2008].

- R8. **Abstraction.** To deal with the complexity of traceability, it is essential to specify a certain level of abstraction, which is the essence of simple and effective software design [Jackson, 2006]. This fundamental principle is accessible in most of evaluated approaches at different abstraction layers. Some approaches, provide a low-level programming interface to address such an abstraction layer. The advantage of this low-level programming interface is to bring flexibility to the approach, as it is thus not tied to a specific implementation language. Other approaches rely on a model-based abstraction to describe a domain or Domain-Specific Modeling Language (DSL or DSML).

However, although model-based approaches are well suited to provide an abstraction layer, the main issue that may arise is the granularity level considered in those models. Indeed, a model could be either too coarse-grained or fine-grained, thus not enabling the modeling of low level functionalities or, on the contrary, being too specific. However, the proposed approaches deal with such issues since models can be refined in the model-driven engineering. For example, a general DSL is implemented to fit a certain level of abstraction, and a specific DSL of component-based approaches rely on the principles of *component* and *composite*, which can contain several components to define in addition several levels of granularity.

3.5 Summary

In this chapter, we reviewed the state-of-the-art approaches for traceability of concerns. In this survey, we have briefly introduced some principles and basic concepts derived from traceability background in Section 3.2. From the traceability approaches selected in Section 3.3, the comparative analysis of approaches has been given in Section 3.4. From this comparative analysis where the regular traceability requirements (representation, mapping, scalability, change and tool) are discussed, we particularly discussed four main requirements (*separation* of concerns, the *adaptability* of software artifacts to projects specific needs, the *flexibility* of software processes and the *uncertainty* of design decisions) that we found to be important to facilitate the maintenance of traceability for flexibility and adaptability of traceability to project-specific needs, such as the impact of uncertainty on safety-critical project.

As shown by the summarized Table 3.3, there exists no evaluated approach which, at the same time,

- provides a coarse-grained and fine-grained *separation* of concerns to facilitate the weak coupling between models and model elements,

Categories	Sub-categories	Approaches	Discussion criteria			
			Separation	Adaptability	Uncertainty	Abstraction
Requirement	Event-Based	EBT	(✓)			
		EBT-DP	(✓)	(✓)		
	Reference	RMRT	✓	(✓)		✓
		MST	✓	✓		✓
	Behavior-Tree	BBT	(✓)			✓
		BBT-TM	(✓)			✓
Model-Driven	Model-based	SDTDA				(✓)
		PTTM	(✓)	(✓)		✓
	Transformation	ODMT	✓			
		TFMT				✓
	Product-Based	MDTA-SPL	(✓)		(✓)	(✓)
		MDTF-SPL	✓	(✓)		✓

Table 3.3: Synthesis of comparison for some traceability approaches

- is flexible enough to *adapt* traceability environments to project specific-needs,
- attaches additional *uncertain* information to traceability links to analyse the influences of uncertainty in the design rationale,
- provides a correct granularity and *abstraction* levels to enable the compromise between the coarse-grained and fine-grained modeling, and the high-level and low-level abstraction.

In this dissertation, we claim that a component-Based modeling with traceability of concerns is one possible way to deal with the challenges raised. As a consequence, considering the approaches and concepts discussed in this chapter, we describe in the next part of the dissertation our contribution for component-based modeling with traceability of concerns. We propose in particular a meta-model bringing together CBSE principles that addresses the first research question and goal we discussed in Chapter 1.

Part III

Contribution

Chapter 4

Component-Based Modeling with Traceability of Concerns

Contents

4.1 Introduction	61
4.2 Motivation and Challenges	63
4.2.1 Motivating Example	63
4.2.2 Challenges	65
4.3 SARA Meta-Model	66
4.3.1 Concern Meta-Model	66
4.3.2 Component Meta-Model	68
4.3.3 Traceability Meta-Model	71
4.4 Process to Use the Meta-Model	73
4.4.1 Actor Roles	74
4.4.2 Implementation Phase	75
4.4.3 Modeling Phase	77
4.4.4 Tracing Phase	77
4.5 Challenges Revisited and Lessons Learned	78
4.5.1 Challenges Revisited	78
4.5.2 Lessons Learned	79
4.6 Summary	81

4.1 Introduction

As shown in Chapter 2, the Component-Based Model Driven Development (CBMDD) [Chen et al., 2009] is based on Model Driven Engineering (MDE) [Schmidt, 2006] and

Component-Based Software Engineering (CBSE) [Szyperski et al., 2002]. The idea promoted by the CBMDD is to use the *software component model* [Lau and Wang, 2007] systematically at different phases of systems development process. As shown in Section 2.6.1, many software component models and frameworks have been proposed and compared over the last few years [Crnkovic et al., 2011, Pop et al., 2014]. However, CBSE still is a new technology and there is much space for research in this field. For example, as CBSE spreads to specific critical domains, such as aeronautical, automotive and railway real-time control systems [CHESS, 2012], a novel component-based process with separation of functional and non-functional concerns is introduced in [Panunzio and Vardanega, 2014] for the development of embedded real-time software systems. However, as discussed in Section 2.6.2, although these various approaches make useful contributions to the discipline, few of them provide an efficient method to handle explicitly traceability of concerns, may be due to extensive cost and time to deal with the traceability management.

In addition, although the take-up of CBMDD has been particularly evident in the automotive and aeronautical engineering domains [AUTOSAR, 2006, CHESS, 2012], the railway sector has lagged behind, partly due to outdated standards and lack of awareness [Favaro and Sartori, 2014]. But the situation is now changing rapidly, and the environment in recent works, e.g., [openETCS, 2012] is representative of a European awareness and push toward implementation of model-based approaches in railway engineering [Favaro and Sartori, 2014]. For these reasons, we highlight the main advantages of CBMDD, i.e., requirement refinement, domain abstraction and separation of concerns in our railway domain-specific component-based model, named SARA [Sango et al., 2014a, Sango et al., 2014b], in order to facilitate the maintenance of traceability links consistently in SARA model when software artifacts are evolving.

One clear benefit of referring to component-based development for traceability is a distinction between high-end and low-end representation of traceability. Indeed, in our component model, we distinguish between two granularity layers for software specification: (i) a high layer, where software is modeled as a set of components communicating through message passing (ii) and a low layer, where their internal behavior is specified. With this separation of concerns we can refer to distinct low-end and high-end traceability representation for a wide range of traceability links, such as intra-level and inter-level traceability links.

This chapter thus covers the complete phase of concern modeling, component modeling and trace modeling. The chapter is structured as follows: Section 4.2 describes a motivating example (Section 4.2.1) by highlighting the need for traceability of concerns and discusses the related challenges (Section 4.2.2). Section 4.3 presents the SARA meta-model composed of a concern meta-model (Section 4.3.1), a component meta-model (Section 4.3.2) and a traceability meta-model (Section 4.3.3). Section 4.4 proposes a generic process, which can be used to instantiate the meta-models for change impact analysis in the case of software evolution. Finally, Section 4.5 discusses several aspects regarding our approach, while Section 4.6 concludes the chapter.

4.2 Motivation and Challenges

This section introduces a motivating example for component-based modeling with traceability of concerns. It illustrates the need for separation and traceability of concerns when using component models and, a change in one model must be propagated through the other models. Based on this example, we then summarize challenges related to modeling a critical software in an component-based architectural fashion with traceability of concerns.

4.2.1 Motivating Example

To motivate our approach, let us consider an example of a System Requirement Specification (SyRS): the SyRS of European Rail Traffic Management System/European Train Control Sub-system [ERTMS/ETCS, 2014]. This system is specified to replace the many incompatible different Automatic Train Control (ATC) systems used in different countries at different times. Note that a SyRS is generally the result of requirement elicitation process in a domain [Pinheiro, 2004]. We focus on Software Requirements Specification (SoRS), which is derived from SyRS.

Assumption 1. SOFTWARE REQUIREMENTS SPECIFICATION TRACEABILITY

We assume that the SoRS is *traceable* if (i) the origin of each of its requirements is clearly related to the SyRS ones and if (ii) it facilitates the referencing of each requirement in the future software development or evolution artifacts.

This assumption requires a change impact analysis in case of change in the SyRS or SoRS for diverse evolution reasons. For example, the SyRS of ERTMS/ETCS is not yet stable as illustrated in the different versions of the ERTMS/ETCS documents [ERTMS/ETCS, 2006, ERTMS/ETCS, 2014]. For instance, the last version [ERTMS/ETCS, 2014] defines some additional functionalities, such as the procedure to supervise the level crossing section or a set of procedures that specify the driver indications related to the specific track-conditions. As depicted by Figure 4.1, the ideal response to a change in the SoRS is that we can quickly determine in various model abstraction levels introduced by any model transformation (1) where to make the change (2) how the change affects the architecture of the existing software (3) which components of software, including their interfaces and behaviors, are affected by the change. Intuitively, the concept of requirement tracing is quite simple: it consists to follow relationships or links.

However, one major challenge when developing large systems is to remember all links that were made to connect system elements. Some traceability models are based on the definition of very rich sets of traces and traceable objects, in an attempt to devise all conceivable (or at least reasonable) traces [Pinheiro, 2004]. This is not a complete solution because the complexity of a model may impair the tracing process. It may even be impossible to know that such links exist in a multi-team development, such as a three-tier party development

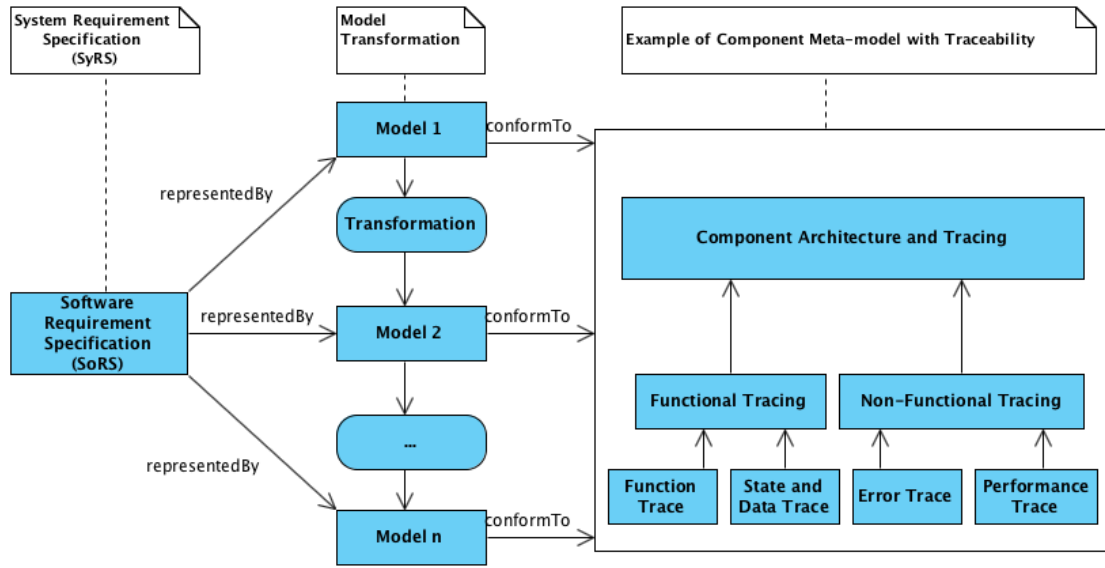


Figure 4.1: An example of model driven view

promoted by CBSE if the different teams do not use the same model to explicitly define traceability links.

Moreover, the matter is complicated when dealing with non-functional tracing beyond functional tracing [Pinheiro, 2004]. Indeed, as illustrated in Figure 4.1, the functional tracing, such as function trace, state and data trace, is related to the functional concerns of software development, while, the non-functional tracing, such as error trace and performance trace, is related to the tracing of non-functional concerns of software development. Functional traces are those related to well established mappings between objects. Thus, it occurs naturally when well-defined models and notations are used to describe objects. On the other hand, non-functional traces are usually related to quality aspects and result from relationships to non-tangible concepts such as goals, contexts, decision and responsibilities, or technical non-functional requirements, such as safety and temporal concerns. This is much like non-functional requirements are usually translated to functional corresponding in order to be traced and verified [Pinheiro, 2004].

However, the solution where non-functional requirements, particularly temporal requirements, have to be re-expressed in terms of functional ones may not be the complete answer because the perception of critical traces and their capture are as important as the traces themselves. Let take an example. In our example of Figure 4.1, suppose that a component supports a safety temporal requirement but, for some reasons, such as invisibility reason, traceability managers do not perceive that connection and fail to register the trace between the component and the requirement. In this situation the trace still exists, because the component still supports the requirement, but this critical trace will be unavailable inside the traceability matrix. As a consequence, certification agencies will fail to get this trace

information and will ask to software designers and developers to provide accurate traceability information to guarantee that this critical requirement is properly designed and implemented. This will be a loss of time. In this context, should non-functional concerns be part of the component external interface specification in the form of annotations?

4.2.2 Challenges

The example previously discussed shows that when relying on component-based modeling with traceability of concerns, the model must refer to the ability to represent, perceive, and follow the traces from requirement concerns to software component concerns, and vice versa. To achieve this ability, we identify three main challenges that we face in this chapter:

- C1. **Selection and separation of concerns.** Modeling of concerns with traceability of concerns implies to select what we want to trace with separation of concerns, whatever functional or non-functional concerns. Indeed, if the traceability of concerns is not customized it can lead to an unwieldy mass of unstructured and unusable data that will never ever be used. The first challenge is thus to identify clearly the concerns to be modeled.
- C2. **Integration of dependability concerns in component-based modeling.** Our goal is not to provide a new component model, but to put together existing component model concepts in order to provide a lightweight component meta-model with the integration of temporal safety concerns. Temporal concerns are behavioral concerns that change over time. Temporal safety concerns are class of temporal concerns that state that "something bad does not happen". The second challenge is thus to provide an abstract component meta-model to describe component models that can be used to model safety-critical software with traceability of concerns.
- C3. **Tracing of concerns throughout component model abstraction levels.** In addition to inter-requirement traceability, which refers to the relationships between requirements, our goal is to provide extra-requirement traceability, which refers to the relationships between requirement concerns and other design concerns and vice versa. In this way, one must be able to model explicitly trace between the focused concerns and other concerns independently from model transformation mechanism. The third challenge is thus to enable traceability of concerns within or across model abstraction levels introduced by any model transformation mechanism.

4.3 SARA Meta-Model

This section describes in details the meta-model of our SARA approach. It is composed of a concern meta-model for modeling system scenario concerns which are themselves related to system requirement and architecture concerns (Section 4.3.1), a component meta-model for modeling component elements with separation of concerns (Section 4.3.2) and a trace meta-model for modeling traceability links among requirement concerns and component elements (Section 4.3.3). Here, meta-models and their constraints are represented with UML class diagrams [OMG, 2005] and Object Constraint Language (OCL) [OMG, 2003], respectively.

4.3.1 Concern Meta-Model

Several concerns, such as requirement, functional architecture, physical architecture, scenario, context, etc, are involving in System Engineering. For example, based on the literature review of functional architecture patterns, Pfister et al. propose a system concern meta-model for formalizing system engineering knowledge [Pfister et al., 2012]. Indeed, leveraging patterns on three main system engineering activities, *i.e.*, requirement engineering, functional architecture design and physical architecture allocation, they argue that functional architecture patterns are key elements for knowledge reuse in system engineering.

Beyond pattern knowledge, designers of complex domain specific systems, such as safety-critical software systems, have to focus on specific concerns for particular industrial domains and engineering needs, such as safety-critical requirement engineering and software engineering. Therefore, our purpose with the concern meta-model is not to provide an entirely new concern meta-model, but a simple concern meta-model with a clear separation of concerns in order to facilitate traceability of concerns during safety-critical software development. As a consequence, we gather the literature existing concepts and reuse them to obtain what we define as the concern meta-model, depicted by Figure 4.2.

In our concern meta-model, we focus on `RequirementConcern`, `ArchitectureConcern` and `ScenarioConcern`. Here, a scenario is one way the system is envisaged to be used in operational life cycle. An interaction of architecture components in order to satisfy the specification of requirements determine a scenario. The key idea is that if a scenario, derived from requirements and related to architecture components, is changed, its model elements can be repeatedly tracked whenever there is a change.

Our requirement-based taxonomy is based on Glinz's concern-based taxonomy of requirements [Glinz, 2007] and the use of non-functional requirements in the software development process [Chung and do Prado Leite, 2009]. As a consequence, requirements are either `FunctionalRequirement` or `NonFunctionalRequirement`. Functional requirements are performed by `Functions` to which they are allocated. Functions compose the functional architecture, which is separated from physical architecture.

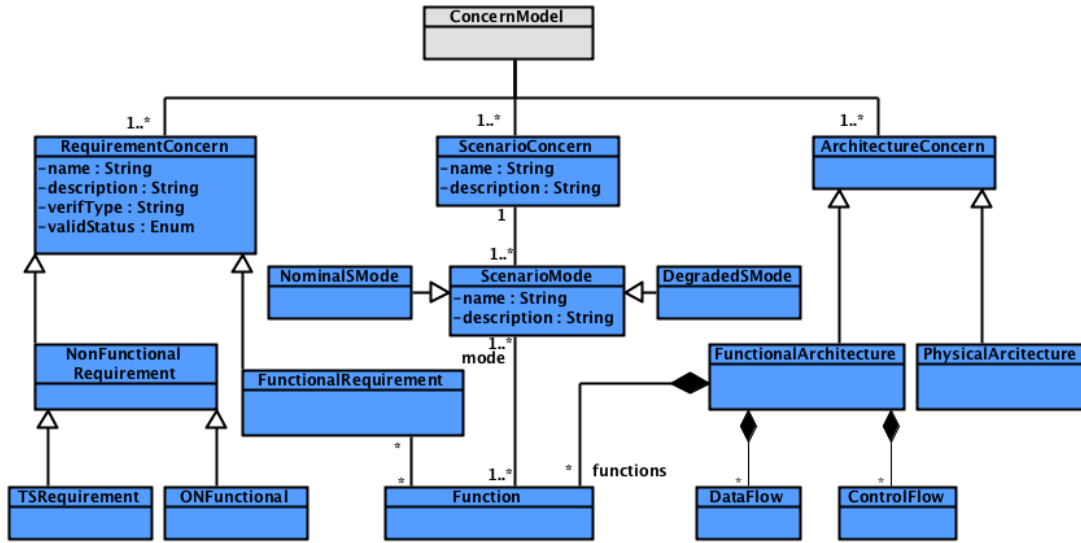


Figure 4.2: Concern meta-model

Here, we focus on functional architecture regardless of physical architecture. Indeed, in [Cloutier and Verma, 2007], a system architecture taxonomy is proposed. Based on this taxonomy, Pfister et al. [Pfister et al., 2012] argue that functional architecture patterns are key elements for knowledge reuse in system engineering, because they capture how the system does what it does by using control loops, algorithms, etc. In this way, a designer has the ability to specify executing conditions of the functions, either by the use of control structures (ControlFlow), or by triggering data (DataFlow), or a combination of the two semantics.

Since non-functional concerns cover considerable characteristics, such as reliability, availability, maintainability, safety, security, usability, flexibility, interoperability and performance, which include time/space bounds, response time, speed, throughput and so on [Glinz, 2007, Chung and do Prado Leite, 2009], we focus on safety requirements subjected to real-time constraints, named TSRequirement, which are separated from other non-functional requirements, named ONFunctional in Figure 4.2. Each requirement is identified by a unique name (name) and can be described (description) in different ways (natural language, mathematical expression, etc.). Requirements can contain other properties, such as the document type, the author and the verification type, that characterize them. For example, for verification activities, we characterize each requirement with attributes verifType, validStatus. The verification type (verifType) is a string characteristic used to indicate whether the verification used is model or code verification, a test or a formal proof, while, the validation status (validStatus) is an enumeration that characterizes the status (pass, fail or unchecked) of the verification results for a specific scenario.

As already mentioned, a ScenarioConcern consists of one or more Function, which are themselves related to functional RequirementConcern. For example, several procedures, such as start of mission, RBC/RBC Handover, passing a level crossing procedure, are defined in chapter 5 of system requirement specification of the ERTM-

S/ETCS system [ERTMS/ETCS, 2014]. Note that in these kinds of safety-critical systems, degraded situations are generally defined to deal with the fault cases. For this reason, a `ScenarioConcern` can have different `ScenarioMode`, which can be either `NominalSMode` or `DegradedSMode`, which deals with fault or exception situations. The current mode can be updated by a method as defined by the OCL method `setMode` shown in Listing 4.1.

```
1 context ScenarioConcern :: setMode(M: Mode)
2     pre: self.Mode -> includes(M)
3     post: self.currentMode = M
```

Listing 4.1: A method to update the current mode of a scenario

The main contribution of this meta-model is to represent functional and non-functional requirements as scenarios with nominal and degraded modes in order to deal with specific dysfunctions affecting temporal and safety concerns. With this concern meta-model, users can use the same uniform formalism to define their software scenario from system requirement specification. The concern modeling provides the foundation for our component-based modeling by elucidating functional and non-functional concerns we want to trace.

4.3.2 Component Meta-Model

Figure 4.3 presents our component meta-model. It expresses that a model is composed of an organized set of `ComponentUnit` and performs an organized set of `Functions`, which compose a `ScenarioMode` shown in the concern meta-model, see Figure 4.2. In the concern meta-model, we have seen that a scenario can have a nominal mode and different degraded modes. Characterizing a mode consists in instantiating the two links `component-mode` and `mode-function` shown in Figure 4.3. Indeed, these links allow us to specify in a static manner (`staticAllocation`) which component units must be used and which functions must be performed for each mode.

A `ComponentUnit` refers to an artifact in the software life cycle. As a consequence, a `ComponentUnit` can be viewed in different `ModelAbstractionLevel` throughout the software model life cycle (e.g., model design level, model analysis level and model implementation level). To refer to component units for traceability process, `ComponentUnit` includes the name and level attributes. A `ComponentUnit` consists of one or more `ComponentElement`, which can be a `ComponentEntity`, `ComponentConnection` and `ComponentOperation`. A `ComponentEntity` is either a `BasicComponent` or a `CompositeComponent`. A `BasicComponent` directly encapsulates behavior, whereas a `CompositeComponent` is built from basic components by using `CompositionConnector`, which can be for example a sequencing, branching or looping connector. Both basic or composite component could be either `ActiveComponent` or `PassiveComponent`. An `ActiveComponent` has its own dedicated thread of execution,

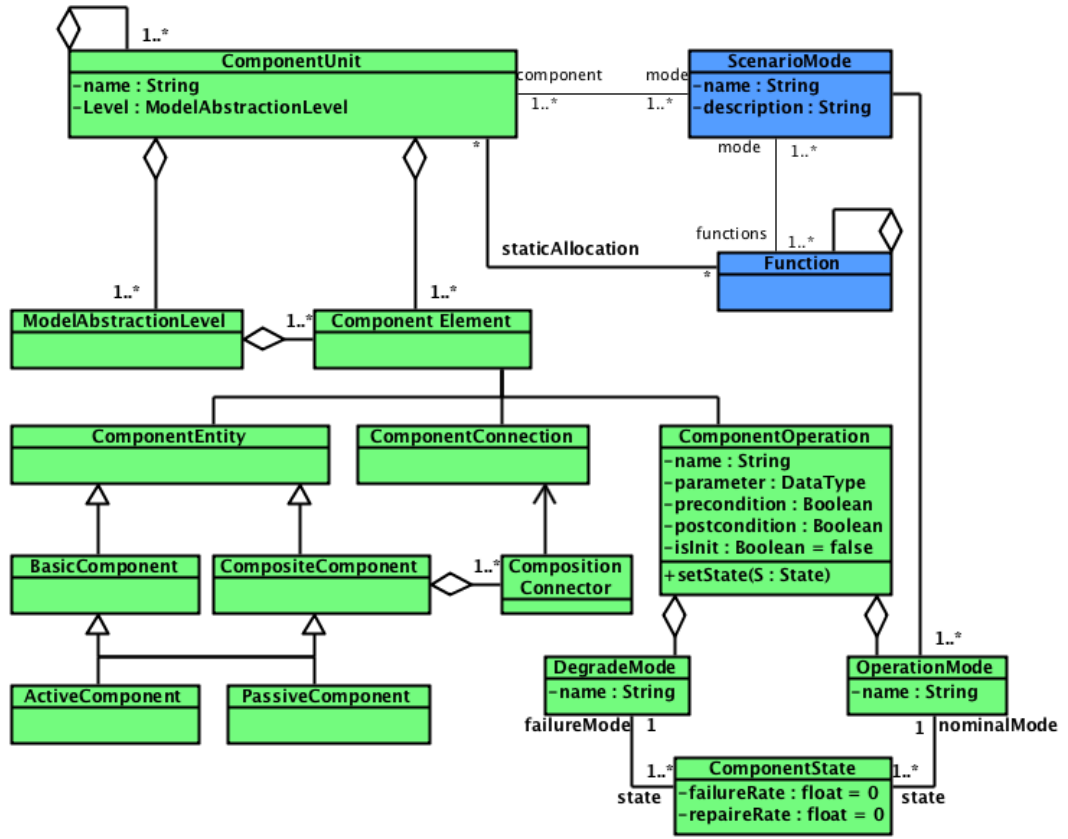


Figure 4.3: Component meta-model

while a `PassiveComponent` is only executed when invoked or triggered by an external execution thread. Each component has a set of `ComponentOperation` that describes its behavior in accordance with functions or a scenario mode that it performs. Each operation has a signature including its name and a set of ordered parameters. The name identifies this operation. Each parameter-passing mode could be in, out or in-out. The parameter is typed by an already defined data types, such as basic types of any programming language (e.g., integer) or composite types (e.g., record). By keeping the separation of concerns in mind, each operation is annotated by non-functional concerns (pre and postcondition).

Moreover, the component model represented in Figure 4.3 shows the evolution model of the component in the form of a state transition system specified by the `isInit` attribute and `setState` method in `ComponentOperation`. The formal definition of our component state transition model is detailed in Section 5.3.1. Here we give an intuitive presentation. First, an initial state of a component and its uniqueness must be defined as illustrated by the OCL constraint of Listing 4.2. Then, a current state of a component can be updated by the method `setState` defined in the OCL constraint of Listing 4.3.

As illustrated in Figure 4.3, the link between the classes `ScenarioMode` and `OperationMode` allows the designer to specify the nominal operation mode (`nominalMode`) of a component among the different operation modes. We assume the

```
1 context Component inv:
2     self.OperationMode -> one(om: OperationMode | om.isInit = True)
```

Listing 4.2: Every component must have one unique initial state

```
1 context Component::setState(S: ComponentState)
2     pre: self.OperationMode.ComponentState -> includes(S)
3     post: self.currentState = S
```

Listing 4.3: The current state of a component can be updated by the operation setState

existence and the uniqueness of a nominal operation mode of each component involved in the application. This is ensured by the OCL constraint of Listing 4.4.

```
1 context ScenarioMode inv:
2     self.Component.OperationMode -> one(om: OperationMode | om = self
        .nominalMode)
```

Listing 4.4: Every component must have one nominal operation mode.

As we focus on safety-critical systems, each component can have different `OperationMode` and `DegradedMode`. An `OperationMode` represents functional properties of the component, while a `DegradedMode` represents dysfunctional properties of the component to deal with fault situations. Indeed, in adaptive safety-critical systems, particularly in railway control software systems, due to the duality between availability and safety, systems should be able to degrade their functionality as long as failure and repair rates permit it [EN-50128, 2011]. In the case where failure and repair rates do not permit further degradation or recovery, the system applies the fail-safe actions (e.g., stop the train) for safety reasons. As a consequence, in addition to the nominal operation mode stated in OCL constraint Listing 4.4, we also define the fail safe mode `failureMode` related to the degraded mode and the capability to fail and repair. The failure rate (`failureRate`) and repair rate (`repairRate`) are presented as attributes in the component state, see Figure 4.3. In this way, every component must have at least one operational mode (nominal mode or degraded modes) and one fail-safe mode, as ensured by OCL constraint in Listing 4.5.

The benefit of our component-based meta-model compared to other component-based meta-models, such as [Rychly, 2011, Becker et al., 2009], is that it supports the integration of dependability concerns coming from a system concern modeling into a software component-based modeling. The benefit for traceability is that model elements are clearly designed with separation of concerns in order to facilitate the traceability of concerns.

```

1 context Component inv:
2   self.OperationMode -> exists (om: OperationMode | om.name = "POM")
3   and self.DegradedMode -> one(dm: DegradedMode | dm.name = "FSM")

```

Listing 4.5: Every component must have at least one Passive Operation Mode named POM and one Fail Safe Mode named FSM

4.3.3 Traceability Meta-Model

In the concern meta-model shown in Figure 4.2, we focused on RequirementConcern, ArchitectureConcern and ScenarioConcern. In Figure 4.3, we present a component meta-model that provides a support to represent with separation of concerns software scenarios as an organized set of components. The idea was that if a scenario derived from requirement specification is changed for diverse evolution reasons, the component elements can be easily tracked, re-designed and re-tested. In this section, we introduce the traceability model to facilitate the traceability of concerns. Figure 4.4 shows our traceability meta-model.

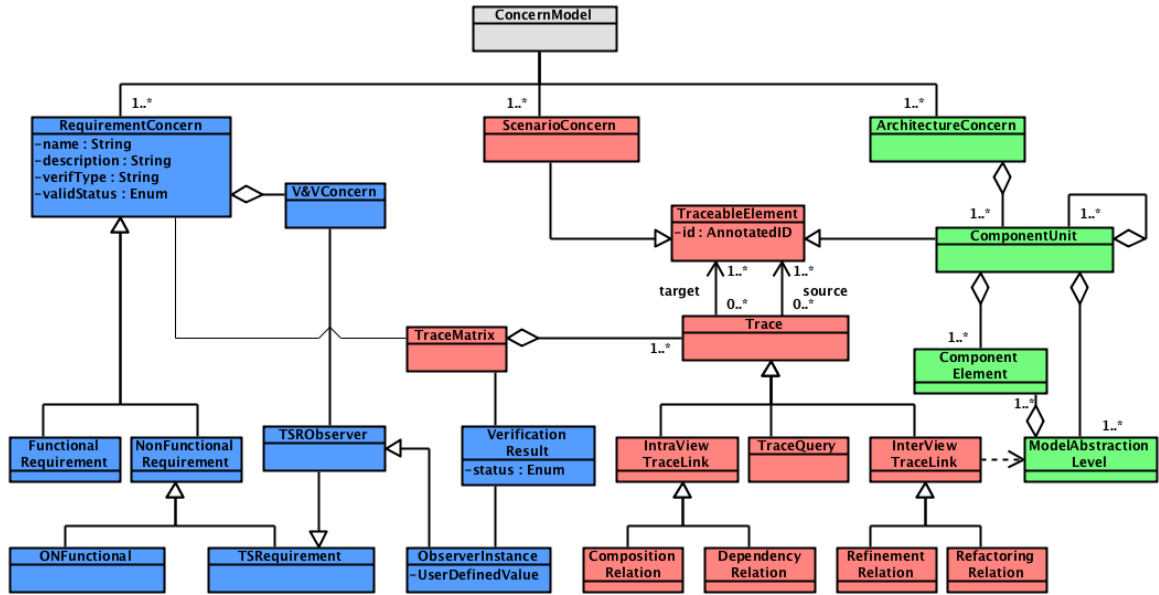


Figure 4.4: Traceability meta-model

A TraceableElement is either a ComponentElement through a ComponentUnit and RequirementConcern through a ScenarioConcern. Trace elements are represented explicitly by Trace, which links one or more source elements to one or more target elements. As we focus on tracking model elements, within or across the different abstraction levels, a Trace link is either an IntraViewTraceLink or an InterViewTraceLink. In an IntraViewTraceLink, traces are generated by the model construction relation, such as

CompositionRelation or DependencyRelation, while, in an IntraViewTraceLink, traces are generated by the process of model transformation as it is specified in the standard of Meta Object Facility (MOF) 2.0 Query/View/ Transformation Specification [OMG, 2011].

Many transformation languages, *e.g.*, ATL [Jouault, 2005, Yie and Wagelaar, 2009], and meta-programming environment, *e.g.*, Kermeta [Falleri et al., 2006], include the definition of a traceability meta-model. Generally, these traceability models are often depended to the transformation model. Moreover, as we focus on concern change impact in different model abstraction levels, there are different techniques to calculate the difference or union between models. Most of them are based on the use of a persistent identifier [Alanen and Porres, 2003]. However, if we add an identifier to each element from the input model and then we apply a no independent transformation model, there is no guarantee of finding these identifiers in the output model. This is particularly true when the transformation is used as a black box. For this reason, we propose to identify each TraceableElement by an identifier and to forbid its modification by transformation models. Precisely, we propose an annotation mechanism, which plays the role of an identifier (AnnotatedID) in the TraceableElement, allowing the traceability of elements after the execution of the transformation. Thus, each element's name in the model should conform to the minimal AnnotatedID BNF syntax, shown in Listing 4.6. The mark describes the string used to separate the different information. The choice of the mark is left to the designer who has a minimal knowledge of the transformation language. For example, we use an annotation mark "@" related to our transformation rules [Sango et al., 2014a] used to translate the SARA model into the timed automata model for time-related property verification.

```

1<annotatedId> ::= <mark><name><mark><parent><mark><metaclass><mark><type><mark>
2<mark> ::= @[String] /*the mark to separate the different information*/
3<name> ::= String /*the name of the current element*/
4<parent> ::= String /*the name of the direct parent of each element */
5<metaclass> ::= Attribute | Operation | Class | Package /*the meta class*/
6<type> ::= String | Integer | Boolean | Class /*the type of the current element*/

```

Listing 4.6: An annotation mechanism to maintain traceability links

In the traceability meta-model shown in Figure 4.4, a Trace can be stored and managed in TraceMatrix. Then, an explicit TraceQuery is defined in order to query or deduce impact analysis information. Several representations (matrix, cross-references, or graph-based representations) can be found in the literature [Wieringa, 1995]. We use a simple mapping table to represent the traces, *i.e.*, to keep the traceability link between source elements and target ones. To build a matrix table, we use a simple algorithm shown in Listing 4.7, which analyzes the annotated source and target model to retrieve information about their elements. This basic mapping table is a two dimension matrix (SourceElements, TargetElements). It is intended to simplify the visualization and representation of any traces between source and target elements.

```

1 input : Model @sourceModel, @targetModel; /* annotated models */
2 output: TraceMatrix mappingTable;
3 /* forward mapping */
4 foreach (@source in @sourceModel) do
5 /* analysis of the annotated target model to find
6 one or more elements that information matches source element */
7   if @source exists in @targetModel then
8     insert-into mappingTable @source as source element
9     /* Verification of corresponding @target element in @targetModel */
10    if @source = @target then
11      insert-into mappingTable @target as unchanged target element;
12    else
13      insert-into mappingTable @target as changed target element;
14  else
15    insert-into mappingTable @source as source element;
16    insert-into mappingTable null as target element;
17 /* backward mapping */
18 foreach (@target in @targetModel) do
19   if @target is not associated with an element in @sourceModel then
20     insert-into mappingTable @target as target element;
21     insert-into mappingTable null as source element;
22 return mappingTable;

```

Listing 4.7: Building of traceability matrix by using a simple mapping table

The difference in our approach compared to other approaches, such as [Falleri et al., 2006], is that our approach is independent of particular transformation mechanism, because in Listing 4.7, we consider the transformation from @sourceModel to @targetModel as a black box. In [Falleri et al., 2006], the trace generating code is tangled with the transformation code on the definition of a tracing operation. In addition, our meta-models are defined as generic as possible. Thus, it can be specialized and integrated in a meta-programming environment, such as Kermeta framework [Falleri et al., 2006], in order to cover traceability of concerns beyond the traceability for model transformation. However, we do not claim that our meta-models to be exhaustive. We can just fairly argue that this is enough to support component-based modeling with traceability of concerns in most of our use cases, as shown in Chapter 6. In Section 4.4, we propose a process, which can be used to instantiate or extend the defined meta-models for a change impact analysis.

4.4 Process to Use the Meta-Model

The conceptual meta-models of our approach are defined in the previous Section 4.3. The meta-models are defined to be quite general, but like any meta-model they can be first extended, if necessary, and then used. In this section, we propose a generic process shown

in Figure 4.5, which can be used to instantiate the meta-models for change impact analysis in the case of software evolution. This process can be summarized in three phases: (1) the *implementation phase*, which consists in implementing the meta-model with a chosen programming language, (2) the *modeling phase*, which consists in modeling identified concerns, architecture elements and trace-links and (3) the *tracing phase*, which consists in tracing elements within or across model abstraction levels. Figure 4.5 is a UML activity diagram with three UML vertical swimlanes (columns) for the three phases. It also depicts the roles of actors: *architect*, *project engineer* and *application engineer*.

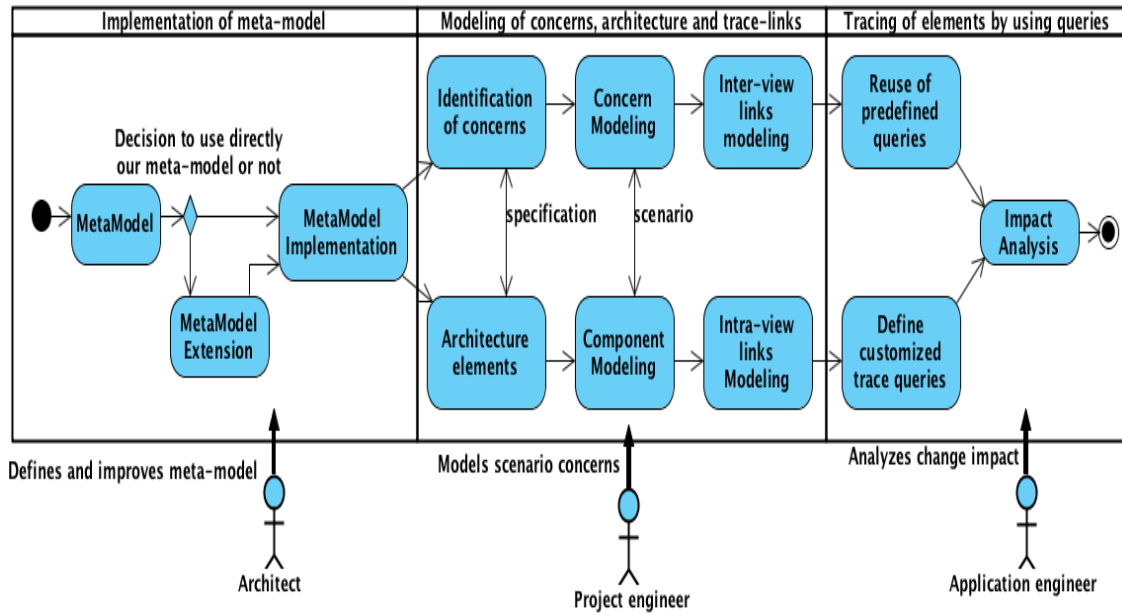


Figure 4.5: A process to apply the metamodel and tracing concerns

4.4.1 Actor Roles

Our approach distinguishes between three roles of actors, *architect*, *project engineer* and *application engineer*.

- **Architect.** An architect is an expert in the particular domain targeted by software component-based modeling with traceability of concerns. He/She is responsible for defining or extending both the meta-model and its implementation in order to facilitate the tasks of project engineers. For example, he/she can customize the meta-model to fit with project-specific needs.
- **Project engineer.** A project engineer has all the information about a particular project, and thus defines the related scenarios, which are different ways the software is envisaged to be used in operational lifecycle. A scenario is thus an interaction of software

components in order to satisfy requirement specification. He/She uses the meta-model to model scenario with traceability of concerns. We recall that models are used systematically at different phases of the software development process, including design phase, implementation phase and verification phase.

- **Application engineer.** The application engineer is the final user. Based on scenario models that include traceability links, application engineers can determine the impact of changes on the software concerns during the software development process. For this, he/she uses predefined or customized trace queries to infer traceability information within or between different models.

4.4.2 Implementation Phase

The implementation phase is shown in the left-hand column of Figure 4.5. Our meta-models like any meta-models can be first extended, if necessary, and then implemented. Here, the syntactic and semantics of our meta-model elements are described in the Document Type Definitions (DTD) of XML language.

```

1<?xml version="1.0" encoding="UTF-8"?>
2<!-- This DTD defines the component meta-model elements. -->
3<!ELEMENT ModelAbstractionLevel (CompView)+>
4<!ATTLIST ModelAbstractionLevel
5      id CDATA #REQUIRED
6      ref CDATA #IMPLIED
7      name CDATA #REQUIRED
8      type CDATA "Component model">
9<!ELEMENT CompView ((CompView)*, (CompElement)*)>
10<!ELEMENT CompElement (CompEntity | CompOperation | CompConnection)*>
11<!ELEMENT CompEntity (CompEntity | CompConnection)*>
12<!ELEMENT CompOperation ((CompOperation)*, (Parameter)*)>
13<!ELEMENT CompConnection ((Parameter)+)>
14<!ELEMENT Parameter (out | in)+>
15<!ELEMENT in (CompEntity)+>
16<!ELEMENT out (CompEntity)+>

```

Listing 4.8: Implementation of component meta-model with DTD

Listing 4.8 illustrates the DTD for component modeling in accordance with the meta-model shown in Figure 4.3. The most important to notice here, is that each component model consists of one or more `ModelAbstractionLevel`, see line 3 of Listing 4.8. Each level includes the attributes `id`, `ref`, `name` and `type`. A component view in a level consists of subviews and contains component elements, which are component entities, component operations and component connections. By keeping in mind the separation of concerns, the different views of model abstraction are separately referenced in the XML instance as illustrated in the modeling phase, Section 4.4.3, Listing 4.11.

```

1<?xml version="1.0" encoding="UTF-8"?>
2<!-- This DTD defines concern meta-model elements. -->
3<!ELEMENT ConcernModel (ConcernGroup+, TraceableEltBetweenView+,ComponentModel+)>
4<!ELEMENT ConcernGroup (ReqConcern+)>
5<!ATTLIST ConcernGroup name CDATA #REQUIRED>
6<!ELEMENT ReqConcern (Description)>
7<!ELEMENT Description (#PCDATA)>
8<!ELEMENT ComponentModel (CompUnit+)>
9<!ATTLIST ComponentModel name CDATA #REQUIRED>
10<!ELEMENT CompUnit (CompUnit*)>
11<!ELEMENT TraceableEltBetweenView EMPTY>
12<!ATTLIST TraceableEltBetweenView
13      HighModelElement IDREF #IMPLIED
14      LowModelElement IDREF #IMPLIED>

```

Listing 4.9: Implementation of concern meta-model with DTD

As in component model, Listing 4.9 presents the concern model. The concern model also follows its corresponding meta-model shown in Figure 4.2. It provides an explicit definition of the traceable elements between two (high and low) model abstraction levels (see lines 12-14 of Listing 4.9).

```

1<?xml version="1.0" encoding="UTF-8"?>
2<!-- This DTD defines the traceability meta-model elements. -->
3<!ELEMENT Trace (IntraViewTraceLink | TraceQuery | InterViewTraceLink)+>
4<!ELEMENT IntraViewTraceLink ((Source | SourceQuery), (Target | TargetQuery))>
5<!ELEMENT InterViewTraceLink ((Source | SourceQuery), (Target | TargetQuery))>
6<!ELEMENT TraceQuery (ForwardTraceQuery, BackwardTraceQuery)>
7<!ELEMENT ForwardTraceQuery (SourceQuery, TargetQuery)>
8<!ELEMENT BackwardTraceQuery (SourceQuery, TargetQuery)>
9<!ELEMENT Source (traceable-element)+>
10<!ELEMENT Target (traceable-element)+>
11<!ELEMENT SourceQuery (#PCDATA)>
12<!ELEMENT TargetQuery (#PCDATA)>
13<!ELEMENT traceable-element EMPTY>
14<!ATTLIST traceable-element id CDATA #REQUIRED type CDATA #REQUIRED>

```

Listing 4.10: Implementation of traceability meta-model with DTD

Finally, as in component and concern models, Listing 4.10 presents the traceability model. It is also defined in accordance with its meta-model shown in Figure 4.4. A trace consists of explicit intra and inter view trace links, and intentional trace queries. Basically, tracing concerns within or across views can support several goals, such as change impact assessment, regression testing and software maintenance. Here, we focus on the impact analysis of concern evolution. As a consequence, in the trace model, forward and backward tracing queries are intentionally defined (see line 6 of Listing 4.10). Forward tracing queries

are intentionally defined to inspect component model elements from a given set of requirement concerns, and backward tracing queries are intentionally defined to inspect the set of concerns that model entities are related to.

4.4.3 Modeling Phase

The modeling phase of concerns and architecture elements is shown in the middle column of Figure 4.5. It consists in identifying concerns and system architecture from the software requirement specification. Once the concerns and the architecture elements are identified, the concern and architecture component models are explicitly described in XML language in accordance with the DTDs presented in Section 4.4.2.

```

1<?xml version="1.0" encoding="UTF-8"?>
2<!DOCTYPE ModelAbstractionLevel PUBLIC "ModelAbstractionLevel" "component-model.
   dtd">
3<ModelAbstractionLevel id="Use_case" ref="" name="Use case model">
4  <CompView id="Sara_View_ID" ref="Sara.xml" type="design view"></CompView>
5  <CompView id="Verif_View_ID" ref="Verif.xml" type="verif view"></CompView>
6  <CompView id="Impl_View_ID" ref="Impl.xml" type="impl view"></CompView>
7</ModelAbstractionLevel>

```

Listing 4.11: XML instance of component model DTD

For example, in Listing 4.11, we focus on three views of model abstraction: SARA component modeling view for the model design level (see line 4 of Listing 4.11), its analysis view for the model verification level (see line 5 of Listing 4.11) and its programming language implementation view for the model implementation level (see line 6 of Listing 4.11). Indeed, after the implementation of meta-models, we need to provide their instances for specific case studies, as illustrated in Section 6.4.2.

4.4.4 Tracing Phase

The tracing phase of elements is shown in the right-hand column of Figure 4.5. As already mentioned, the traceability of concerns can be used in different activities, such as change impact assessment, regression testing and software maintenance. Here, we focus on the impact analysis of concern evolution. After explicit modeling of concerns and architecture elements, we define the trace-links among the concerns and the architecture elements within or across views. This is again done by instantiating XML from the meta-model DTD that represents the trace model, shown in Listing 4.10. In the trace model, relationships between a source and a target are either explicit intra or inter view links, or intentional trace queries. To query elements in an XML representation, we have implemented a set of XQuery queries. These queries are grouped by two types of queries:

1. *ForwardTraceQuery* (*views*, *concernID*) which matches component elements for a given set of concerns.
2. *BackwardTraceQuery* (*views*, *ElementID*) which determines concerns for a given set of component elements.

```
1(: Declaration of forward trace query:)
2declare function oxy:ForwardTraceQuery($oxy:view as xs:string, $oxy:query as xs:
  string) {
3    let $link := $TargetQuery ($oxy:query)
4    return
5      if (empty ($link)) then $oxy:query
6      else $oxy:query union oxy:ForwardTraceQuery ($view, $SourceQuery)};
```

Listing 4.12: Implementation of of forward trace query

For example, the implementation of forward trace query is illustrated in Listing 4.12. Its use for a specific case study is presented in Section 6.4.2.

4.5 Challenges Revisited and Lessons Learned

4.5.1 Challenges Revisited

We have presented in this chapter the conceptual meta-models of our component-based modeling with traceability of concerns. Let us now revisit the challenges identified in Section 4.2.2 and discuss how our approach faces them.

- C1. **Selection and separation of concerns.** To face this challenge, we have defined a concern meta-model, which identify clearly the concerns we want to trace. The main contribution of this meta-model is to represent functional and non-functional requirements as scenarios with nominal and degraded modes in order to deal with specific dysfunctions affecting temporal and safety concerns.
- C2. **Integration of dependability concerns in component-based modeling.** To face this challenge, we have provided a component meta-model that abstracts the component model and component state evolution model. We rely on existing concepts found in the literature of component models, gathered in our component meta-model. In addition, our component meta-model supports the integration of dependability concerns, precisely temporal safety concerns, coming from a system concern modeling. The benefit for traceability is that model elements are clearly designed with separation of concerns in order to facilitate the traceability of concerns, whatever functional or non-functional concerns.

C3. Tracing of concerns throughout component model abstraction levels. To face this challenge, we have described how requirement concerns in concern meta-model and component elements in component meta-model can be traced by using explicit intra and inter view trace links and intentional trace queries. Our approach is independent of particular transformation mechanism, because in our traceability matrix construction, we consider the transformation from a source model to a target model as a black box.

4.5.2 Lessons Learned

In above Section 4.5.1, we have discussed the *specific challenges* that we have faced in this chapter. In Section 3.4, *general challenges* of traceability are discussed and comparative analysis of the state of the art approaches are presented in Table 3.2 around four main traceability requirements: separation (R5), adaptability (R6), uncertainty (R7) and abstraction (R8). Here, we summarize in Table 5.1 the learning we got, the contribution and limitations of our SARA approach relative of these general requirements.

Categories	Sub-categories	Approaches	Discussion criteria			
			Separation	Adaptability	Uncertainty	Abstraction
Requirement	Event-Based	EBT	(✓)			
		EBT-DP	(✓)	(✓)		
	Reference	RMRT	✓	(✓)		✓
		MST	✓	✓		✓
	Behavior-Tree	BBT	(✓)			✓
		BBT-TM	(✓)			✓
Model-Driven	Model-based	SDTDA				(✓)
		SARA	✓	(✓)	(✓)	✓
		PTTM	(✓)			✓
	Transformation	ODMT	✓			
		TFMT				✓
	Product-Based	MDTA-SPL	(✓)		(✓)	(✓)
		MDTF-SPL	✓	(✓)		✓

Table 4.1: Synthesis of comparison for some traceability approaches including ours

R5. Separation. It is clear for every software practitioners that the separation of concerns is one of fundamental design principle of software development. However, experience shows that this benefit is much more difficult to achieve than it may seem. Indeed, it requires substantial effort to arrive at common and stable specifications, clean inter-

face design and effective consolidation of functional and non-functional part of system components.

To face this challenge, our concern meta-model comes with the coarse-grained separation of requirement concerns and our component meta-model comes with fine-grained separation of component elements in order to facilitate traceability of concerns. We also observed that the representation of external traceability links, stored in separate models that can be combined with the system primary models, can facilitate the weak coupling between models and traceability information.

- R6. **Adaptability.** If requirement traceability is not customized it can lead to an unwieldy mass of unstructured and unusable data that will never ever be used. The adaptation of trace capture and usage to project-specific needs is a prerequisite for successfully establishing traceability within a project and for achieving a positive cost-benefit ratio. Adaptable traceability environment depends on several factors: (i) pre-definition of trace, (ii) definition of project-specific trace, and (iii) experience based improvement [Mader and Egyed, 2012].

To face this challenge, our concern meta-model comes with an uniform traceability meta-model, which can be used by (1) a traceability architect to improve the predefined traces based on its experience, (2) by a project engineer to define a project-specific traces, and (3) by an application engineer, which uses predefined or customized trace query to infer traceability information within or between different models. At technical level, although m-to-n links are more difficult to represent graphically, compared to 1-to-1 traceability links, they can help to reduce the number of required links and therefore simplify the adaptation. For example, when we want to adapt a traceability to a project-specific requirement, representing the causality between this requirement and several design artifacts is important to know all the design artifacts that relate jointly to the same requirement.

- R7. **Uncertainty.** Uncertainty plays a role in any system that needs to evolve continuously to meet the specified or implicit goals of the real world. In safety-critical systems, the impact of uncertainty on software development can be even more sever than traditional software systems. Traceability of design decisions in safety-critical software development is an important and relevant issue, as these are key points where uncertainty influences the design process.

For performing traceability in the presence of uncertainty in our model, one can attach rational information, such as the safety integrity level which gives the confidence we have in the creation of traceability links between components. However, in safety-critical systems, it is crucial to establish to what extent individual components can be trusted and depended on, as well as developing a solid understanding of the impact from individual components on the overall dependability of the global system.

- R8. **Abstraction.** Abstraction is another fundamental principle of software development. To deal with the complexity of traceability, it is essential to specify a certain level of ab-

straction, as well as to use the suitable model granularity in each abstraction level. For example, model could be either too coarse-grained or fine-grained, thus not enabling the modeling of low level functionalities or, on the contrary, being too specific.

Our proposed approach deals with such issue since models can be refined in the model-driven engineering, particularly the component-based model driven development. First, the meta-model can be implemented with Domain Modeling Language to fit a certain level of abstraction. For simplicity, our meta-model is implemented with the DTD/XML language. Second, we particularly focus on the component-based modeling language which rely on the principles of *component* and *composite*, which can contain several components to define several levels of granularity.

4.6 Summary

In this chapter, we have presented meta-models for component-based modeling with traceability of concerns. We have first illustrated our approach with an example of railway system requirement specification in Section 4.2.1. Then, we derived three main challenges that we have faced in Section 4.2.2. Relying on the defined meta-models in Section 4.3, one can define the complete phase of concern modeling, component modeling and trace modeling. We also proposed a generic process in Section 4.4, which can be used to instantiate the meta-models for change impact analysis in the case of software evolution.

Although we do not claim that our meta-models to be exhaustive, we can fairly argue that this is enough to support component-based modeling with traceability of concerns in most use cases. For example, we use the generic process proposed to evaluate our approach in railway case studies, as presented in Chapter 6.

In our approach, explicit intra- and inter-view links and intentional trace queries are defined in order to trace information within or across the different abstraction levels independently of any transformation mechanisms. However, since we focus on safety-critical systems, it is important to present an efficient transformation approach to translate our high-level design model to the low-level analysis level, for which we can verify temporal safety concerns that we are focused on. As a consequence, in Chapter 5 of this dissertation, we formalize our high-level component-based model and present a transformation mechanism into low-level model for observer-based verification.

Chapter 5

Observer-Based Verification with Patterns of Properties

Contents

5.1 Introduction	83
5.2 Motivation and Challenges	84
5.2.1 Motivating Example	85
5.2.2 Challenges	87
5.3 SARA to TAIO Formal Model	88
5.3.1 SARA Formal Definition	88
5.3.2 TAIO formal definition	100
5.3.3 From SARA model to TAIO Model	103
5.4 A 3-Layer Approach for OBV	108
5.4.1 Upper layer	108
5.4.2 Middle layer	110
5.4.3 Lower layer	110
5.5 Challenges Revisited and Lessons Learned	112
5.5.1 Challenges Revisited	112
5.5.2 Lessons Learned	112
5.6 Summary	114

5.1 Introduction

Verification and Validation (V&V) are at the heart of the development process of software that require high dependability, such as the train control software, as discussed in Chapter 2,

Section 2.4. However, one of the important issues limiting the large adoption of formal verification technologies is the difficulty, for non-experts, to express their requirements using the formal specification languages supported by verification tools. For this reason, some approaches, based on specification patterns, are proposed. Patterns propose a user-friendly syntax which facilitates their adoption by non-experts.

As shown in Section 2.5, Observer-Based Verification (OBV) is one these techniques. It is a model checking technique where patterns of properties to be verified are defined as observer automata, which can be applied directly in timed model checking tool. Indeed, OBV consists in writing a system properly as an observer, *i.e.*, an Input and Output (IO) machine that synchronously listens to the IO operations of a system to detect an execution trace that violates a system property. Several formal verification methods have been applied over OBV: [Bhatti et al., 2011, Soliman et al., 2012, Mekki et al., 2012]. However, unlike the timed temporal logic approaches [Henzinger, 1998], most of pattern-based approaches are lacking in well-founded theory or use inappropriate definitions. One of our goals is to contribute toward reversing this situation by using the well-founded theory of time traces of the Timed Input-Output Labeled Transition System (TIOLTS) [Krichen and Tripakis, 2009]

In this chapter, we introduce a formal model of our SARA meta-model presented in Chapter 4. The first reason is to provide a formal definition of the SARA model including formal definition of component interoperability by using the theory of component contract [He et al., 2006]. The second is to facilitate the transformation of this formal model into timed automata model, for which we can use a timed model checking tool to apply OBV. Finally, by using the composition theory of TIOLTS, we demonstrate that the defined observers have no impact on the behavior of the system under observation, meaning that any trace of the observed system is preserved in the composition of the system and the observers.

This chapter thus covers the formal concerns of our approach. The chapter is structured as follows: Section 5.2 describes a motivating example by highlighting the need for the transformation of high-level design language into low-level language for verification. It also discusses the related challenges. Section 5.3 presents SARA to TAO formal models. Section 5.4 presents the overview of our OBV approach by highlighting its soundness. Finally, Section 5.5 discusses several aspects regarding our approach, while Section 5.6 summarizes the chapter.

5.2 Motivation and Challenges

This section introduces a motivating example for observer-based verification with patterns of temporal safety properties. It illustrates the need for transformation of our design model into a formal model for which we can use a timed model checking tool for formal verification of properties. Based on this example, we then summarize challenges related to verifying a component model with patterns of properties.

5.2.1 Motivating Example

To motivate our approach, let us consider the well-known rail-road Level Crossing Automatic Protection System (LC-APS). The detailed background of LC-APS is presented in Section 6.4.1. Here, we focus on an abstract example of software component-based architecture, for which we can deride challenges discussed in this chapter. The abstract example is shown in Figure 5.1.

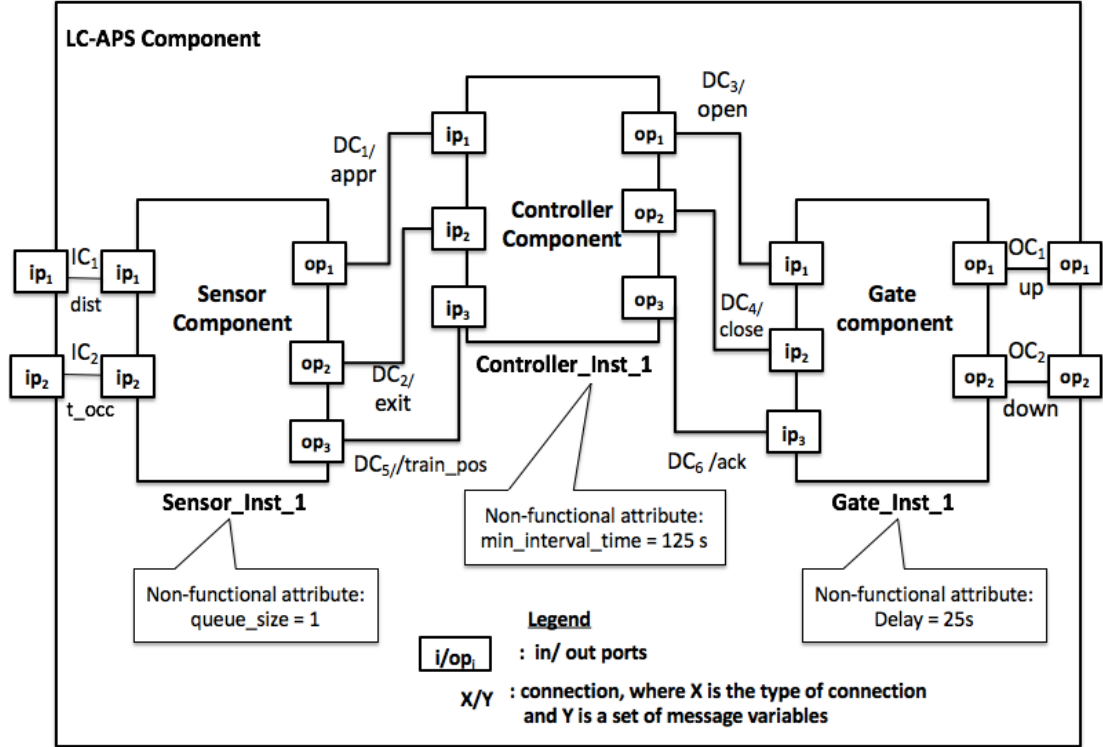


Figure 5.1: LC-APS motivating example

The top level component, named LC-APS component, contains three components: *Sensor*, *Controller* and *Gate*. In this level, components communicate through the port (e.g., ip_1 and op_1) by message passing (e.g., train position *train_pos* and track occupancy *t_occ*). Components have run-time behavior. For instance, the component instance *Controller_Inst_1* of component type *Controller* is the component that defines the main control loop with the non-functional real-time attribute *min_interval_time = 125s*. It uses the approaching *appr* and exiting *exit* information of trains in the monitored intersection, and acts as mediator between the *Sensor* and the *Gate* components. Indeed, the *Sensor* receives information from physical sensors of the track in order to detect trains that approach and exit in the monitored intersection. This information is translated into a *appr* event or *exit* event depending on the position of train *train_pos* from the LC. Based on the input messages, the *Controller* determines the output messages (*open* or *close* events) for the supervision of the gate. The *Gate* responds to events by moving barriers *up* or *down*.

Although Figure 5.1 presents the global concepts of CBSE shown in Section 2.2.2, like components, port-based interfaces, composition through ports, it does not provide component-based implementation techniques heavily related on object-oriented techniques. For example, Listing 5.1 shows an object-oriented specification with the Ada programming language for the LC-APS component-based architecture. Here, we focus on sensor component, which specification is given in lines 20-34 of Listing 5.1.

```

1 with Sensor; with Controller; with Gate;
2 with Ada.Calendar; —with Ada.Real_Time;
3 Procedure LC_APS is
4   — Input parameters:
5   Train_X_Coord : Float := 0.0; Gate_Down : Boolean := False;
6   — Component instances extends Sara.ComponentTask defined in Annex Listing A.5
7   package Sensor_Inst_1 is new Sara.ComponentTask
8     (Comp_Type => Sensor.CompType, Comp_Inst => Sensor.CompInst,
9      Queue_Size => 1, MIAT => 10, Operation => Sensor.Op);
10  package Controller_Inst_1 is new ...; package Gate_Inst_1 is new ...;
11  —Top-level component resources:
12  use Sensor_Inst_1; use Controller_Inst_1; use Gate_Inst_1;
13  use Ada.Calendar; — or Ada.Real_Time
14  Start, Finish : Time;
15 begin
16   Start := Clock; — get system start time
17   ...
18 end LC_APS;
19
20 package Sensor is
21   — Component type extends Sara.Component defined in Annex Listing A.4
22   type CompType is new Sara.Component.Comp_Type with record
23     Time_Var : Integer;
24     Train_Pos: Float;
25   end record;
26   — other data:
27   Tmax : constant Integer := 15;
28   — operations:
29   Procedure Op (C : in out CompType)
30     — operation annotations:
31     with
32     pre => (C.Time_Var = 0),
33     Post => (C.Time_Var <= Tmax and then C.Train_Pos=10.0);
34 end Sensor;

```

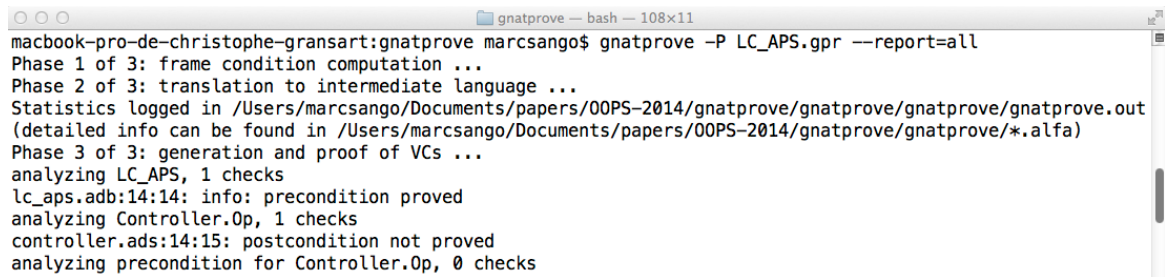
Listing 5.1: Component specification with Ada programming language

When relying on component-based implementation with contracts, one of challenges is to adapt contracts to the applications needs for component interoperability [Chen et al., 2007a]. Indeed, different usages of the component in different applications have different needs. For example, an interface contract for a component in a sequential system is

obviously different from one in a communicating concurrent system. The former only needs to specify the functionality of the methods, *e.g.*, in terms of their pre- and post-conditions as illustrated in lines 32-33 of Listing 5.1, whereas the latter should include a description of the communication protocol, *e.g.*, in terms of interaction traces or real-time constraints.

Another challenge is to ensure both the global refinement and the local refinement [He et al., 2006]. Indeed, global refinement or black-box specification is usually defined as a set containment of system behaviours, and can be verified deductively within a theorem prover. Local refinement or white-box specification is generally based on specification of individual operations, and can be established by simulation techniques or model checking techniques. Let us explain this challenge by considering a simple execution trace of our implementation example.

Figure 5.2 shows the sample runtime execution trace, which checks if the annotations of operations are respected or not. We use the formal proof tool of the Ada language annotations [SPARK, 2014]. However, tasks and synchronization are not currently supported in this tool to check real-time constraints, such as *min_interval_time* stated in the component-based architecture shown in Figure 5.1. For example, when we replace the statement *Ada.Calendar* with the statement *Ada.Real_Time* in line 13 of Listing 5.1 the proof fails.



```
macbook-pro-de-christophe-gransart:gnatprove marcsango$ gnatprove -P LC_APS.gpr --report=all
Phase 1 of 3: frame condition computation ...
Phase 2 of 3: translation to intermediate language ...
Statistics logged in /Users/marcsango/Documents/papers/OOPS-2014/gnatprove/gnatprove/gnatprove.out
(detailed info can be found in /Users/marcsango/Documents/papers/OOPS-2014/gnatprove/gnatprove/*.alfa)
Phase 3 of 3: generation and proof of VCs ...
analyzing LC_APS, 1 checks
lc_aps.adb:14:14: info: precondition proved
analyzing Controller.Op, 1 checks
controller.ads:14:15: postcondition not proved
analyzing precondition for Controller.Op, 0 checks
```

Figure 5.2: An execution trace of Ada implementation for LC-APS

This example clearly illustrates the need for mix verification. Indeed it is challenging to target both large modeling expressiveness and broad verification accuracy with the same language. Either the language is explicitly wide to include the expressiveness and cover a large number of properties that will be hardly supported by the formal proof tools, either the language is restricted and explicitly excludes some complex properties.

5.2.2 Challenges

The example previously discussed shows that when relying on formal component-based modeling and verification, the model must refer to the ability (i) to separate clearly component interface contracts, (ii) to integrate black-box and white-box specifications for both external and internal contracts, and (iii) to guarantee the model transformation chain for mix verification. To achieve these abilities, we summarize the three main challenges that we face in this chapter.

- C4. **Separation of component interface contracts.** An interface is a syntactic and semantics specification of a component. A syntactic information provides the access point of a component, while a semantics information provides the behavioral contract of a component. However, different usages of a component in different applications have different needs. For example, if a component is to be used in a domain-specific real-time applications, such as railway real-time control system, the contract of its interface must also specify real-time constraints, such as the lower and upper bounds of the execution time of a control method. The first challenge is thus to define clearly the interface contracts by separating the functional and non-functional concerns.
- C5. **Grey-box specification.** The challenge of gray box specification is to find a compromise between black-box specification and white-box specification for external and internal contracts of interfaces. In most of component model dedicated for embedded systems, the event-based model is used for black-box specification in support of component composition, and the state-based model is used for white-box specification in support of component internal design. By following this method, the second challenge is to provide high-level domain specific language that facilitates both the event-based simulation and the state-based refinement.
- C6. **Assurance of model transformation.** Domain specific modeling languages greatly simplify the task of system modeling because they present a higher level of abstraction that is easy to work with. Since it is difficult to have the same high-level language suitable for both modeling and verification as broadly as possible, intermediate or low-level languages are frequently introduced to generate verification models that may be used to verify specific properties by using appropriate verification tools, such as theorem provers or model checkers. The third challenge is thus to prove the semantics preservation of the model transformation.

5.3 SARA to TAIO Formal Model

This section describes in details the formal aspect of our component-based modeling and observer-based verification approach. In Section 5.3.1, we present the formal definition of the SARA domain-specific component model. In Section 5.3.2, we introduce the formal definition of Timed Automata model over Input and Output actions (TAIO). Then, in Section 5.3.3, we presents the transformation rules developed to translate SARA application models to TAIO models and prove the semantics preservation of the model transformation by bisimulation.

5.3.1 SARA Formal Definition

The SARA meta-model has been introduced with the aim of traceability of concerns in Chapter 4. In this section, the SARA component model is formally introduced in order to provide

the formal foundation of model elements and to facilitate its transformation into Timed Automata over Input and Output actions (TAIO), for which we can use a timed model checking tool for verification task. Formally, the SARA component model is defined as follows.

Definition 3. SARAMODEL

A SARA model is defined as a tuple $SARAModel = (ModelName, DataModel, ComponentModel, CompositionModel)$, where:

- **ModelName** is the name of the model, which is defined by its designer. Notice that the name of SARA model corresponds to the top level component name, *e.g.*, LC-APS in Figure 5.1.
- **DataModel** is used to describe the data types of variables exchanged between different model elements (see Definition 4).
- **ComponentModel** defines component elements and their relations (see Definition 5).
- **CompositionModel** defines the composition relations between model components (see Definition 8).

Definition 4. DATAMODEL

A DataModel is used for the description of variables of data types, which are the basic entities in our approach. The designer can define a set of data types such as primitive types, enumerations, ranged or constrained types, arrays or composite types. We assume that a set of variables is divided into two disjoint sets: observable variables and unobservable variables. Observable variables are in turn divided into input and output messages, which can be sent or received periodically, sporadically or aperiodically. Unobservable variables or local variables are hidden in the component refinement level considered. As a consequence variables that shall be typed with data types are defined as $DataModel = \{IV, OV, LV, TV\}$, where:

- **IV** is a set of observable input variables of SARAModel; *e.g.*, $IV = \{dist, t_{occ}\}$ in Figure 5.1.
- **OV** is a set of observable output variables of SARAModel; *e.g.*, $OV = \{up, down\}$ in Figure 5.1.
- **LV** is a set of local variables or unobservable variables of SARAModel. For technical reasons of transformation, we assume that all the local variables of model components occur somewhere in the top level component model structure. The disjointness of variables is ensured by attaching suffixes to the type of connection for which variables transit; *e.g.*, $LV = \{DC_1.appr, DC_2.exit, etc.\}$ in Figure 5.1.
- **TV** is a set of temporal variables that are used to describe the real-time constraints. For instance, real-time non-functional attributes that may be attached to the provided

operation of a component include period, deadline, bounded interval and finite queue size buffer; e.g., $OV = \{queue_size, min_interval_time, delay\}$ in Figure 5.1.

For convenience of examples, we introduce some basic notions of traces. Given a primitive type T , a sequence type $\mathbf{Seq}(T)$, and a sequence declaration $s : \mathbf{Seq}(T)$, we use $|s|$, $\mathbf{tail}(s)$, and $\mathbf{head}(s)$ to denote the length, tail, and head of s , respectively. $s_1 \bullet s_2$ denotes the concatenation of the sequences s_1 and s_2 . In the declarations “ $x_1 : \mathbf{in} T$, $x_2 : \mathbf{out} T$ and $x_3 : \mathbf{in out} T$ ”, $\{x_1, x_2, x_3\}$, **in**, **out**, and **in out** stand for a set of variables, an input, output, and input/output parameters, respectively.

Definition 5. COMPONENTMODEL

A component model is defined as a tuple $ComponentModel = (CompName, CompType, CompImpl, CompInst, CompInstBinding)$, where:

- **CompName** is the name of the component type, e.g., *Sensor* in Figure 5.1.
- **CompType** is the design entity that is specified in isolation, with no relationship with other components. The component type therefore specifies provided interfaces and required interfaces by referencing already-defined interfaces. It thus forms the basis for a reusable software artifact. For example, the root component type of SARA model is provided by *Sara.Component.Comp_Type* as presented in our SARA model implementation in Ada language (see Appendix A, Line 5 of Listing A.4).

We distinguish *component type* from *component implementation*. A component type may in fact have distinct implementations.

- **CompImpl** is an implementation of component type. It fulfils two roles. First, it is a concrete realization of a component type as illustrated by *Sensor.CompType* (see line 22 of Listing 5.1). Second, it is the subcontracting unit that must implement all the functional operations of its type. For example, the *Sensor* component implementation includes the procedure $op(C : \mathbf{in out} CompType)$, see line 29 of Listing 5.1.

In addition to the realization of functional operation in the form of sequential code, a component implementation may set of *functional input and output constraints* as illustrated in lines 32-33 of Listing 5.1 or a set of specific *non-functional real-time constraints* in real-time context. For example, a train *Sensor* operation may work correctly only if a train position *train_pos* is executed at a certain time interval e.g., $[0, T_{max}]$, which is related to concurrency and real-time concerns, such a periodic or sporadic release pattern, as stated in *CompInst*.

- **CompInst** is a component instance. It is a design unit instantiated from a component implementation. It is defined by a name and its behavior, i.e., $CompInst = (CompInstName_i, Priority, BehaviorModel)$, where *BehaviorModel* is defined in Definition 7.

CompInstName_i is a component instance name of a component type, where $i \in \{1, \dots, n\}$ because a component type implementation can have different instances. Priority is an integer that defines the execution order of CompInstName_i . For instance, the execution order of Figure 5.1 is: "Gate_Inst_1 < Controller_Inst_1 < Sensor_Inst_1", Where "<" is the execution order operator.

At this level, *non-functional attributes* are given to be compatible with the *non-functional constraints* required at implementation level. For example, for temporal constraints of Sensor component, the temporal attributes include a sporadic release pattern, which includes its Minimum Inter-Arrival Time (MIAT) and finite buffer *Queue_Size* for the incoming requests, as illustrated in lines 8-9 of Listing 5.1.

A component instance serves mainly on two purposes: (1) component *deployment* and (2) component *binding* [Crnkovic et al., 2011], as illustrated in Figure 5.3

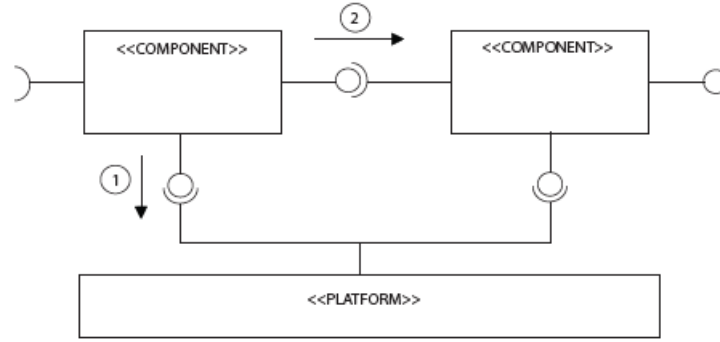


Figure 5.3: Component instance purposes : (1) deployment and (2) binding

Remark 2. Component deployment enables component integration into an execution platform, while component binding enables component interaction at application level. In our work, we focus on application level. As a consequence, we concentrate only on the component binding.

- **CompInstBinding.** A component instance binding is a design unit between one input port, denoted $ip_j \in IP$, and one output port $op_i \in OP$, where $IP = \{ip_1, ip_2, \dots, ip_n\}$ and $OP = \{op_1, op_2, \dots, op_n\}$ are respectively a set of input and output ports of components. A component instance binding is defined as a tuple $\text{CompInstBinding} = (\text{CompInstName}_i, \text{CompConnect})$. CompConnect is defined as a set of three types of connection, $\text{CompConnect} = \{IC, DC, OC\}$, where:

- IC is a set of connections that send input variables $iv_j \in IV$ of SARAModel to an input port $ip_l \in IP$ of k th component instance. It is defined as:
 - * $IC_n: \text{ModelName}.iv_j \rightarrow \text{CompInstName}_k.ip_l$,
e.g., $IC_1: LCAPS.dist \rightarrow \text{Sensor_Inst_1}.ip_1$ in Figure 5.1;
- DC is a set of direct connections between IP and OP of component instances. It is defined as:

* $DC_n : \text{CompInstName}_i.op_j \rightarrow \text{CompInstName}_k.ip_l$,
 e.g., $DC_1 : \text{Sensor_Inst_1}.op_1 \rightarrow \text{Controller_Inst_1}.ip_1$ in Figure 5.1;

- OC is a set of output connections between $op_i \in OP$ of the k th component instance and output variables $ov_j \in OV$ of SARAModel. It is defined as:

* $OC_n : \text{CompInstName}_k.op_i \rightarrow \text{ModelName}.ov_j$,
 e.g.: $OC_1 : \text{Gate_Inst_1}.op_1 \rightarrow \text{LCAPS}.up$ in Figure 5.1.

Remark 3. When bindings have been set, other non-functional attributes can be specified in addition to those given in component instance level. For example, for *end-to-end temporal constraints*, the temporal attributes include end-to-end deadlines on a call chains across components, and queuing protocols. The set of component binding in SARAModel, including the semantics of components, defines a component configuration as defined in Definition 6.

Definition 6. COMPONENTCONFIGURATION

A component configuration is a set of component instance bindings including the semantics of components.

First, we define the semantics of a basic components, *i.e.*, a component instance that directly encapsulates its behavior, by using state-transition systems [Manna and Pnueli, 1995]. A local state s of a basic component is defined as variable assignment $s : IV \cup OV \cup LV \cup TV \rightarrow Val$, where Val denote a set of variable values. A local transition between two local states s and s' is defined as $s \xrightarrow{Pre, Post, Prot} s'$ under the behaviour pre-condition (Pre), post-condition ($Post$) and protocole condition ($Prot$) defined in behavior model (see Definition 7).

Second, based on the semantics of basic components, we define the semantics of a composite component, *i.e.*, a component built from basic components, by using a global state-transition system defined in [Adler et al., 2011]. A global state σ of a composite component C consists of local state $\{s_1, \dots, s_m\}$ of basic components, where s_i is the state of $c_i \in C$, and an evaluation of variables. That is $\sigma = s_1 \cup \dots \cup s_m \cup ((IV \cup OV \cup LV \cup TV) \rightarrow Val)$. Two global states σ and σ' perform a global transtion denoted $\sigma \xrightarrow[CompInstBinding]{Pre, Post, Prot} \sigma'$ under the component instance bindings defined in component model (see Definition 5) and the behaviour conditions defined in behavior model (see Definition 7).

Given a set of inputs $IV' \subseteq IV$, a SARA model S is said to be input-enable with respect to IV' if it can accept any input in IV' at a global state σ . Let C_S denotes a component configuration of S , then $\forall \sigma \in C_S$ and $\forall iv' \in IV'$ implies $\exists \sigma' : \sigma \xrightarrow{iv'} \sigma'$. Let a $CompInstBinding = (\text{CompInstName_k}, \text{CompConnect} = \{IC, DC, OC\})$ of C_S , if $\exists ip \in IP$ such that $\text{ModelName}.iv' \rightarrow \text{CompInstName}_k.ip \in IC$, or if $\exists ov \in OV$ such that $\text{CompInstName}_k.op \rightarrow \text{ModelName}.ov \in OC$ then C_S is called a reachable configuration denoted $Reach(C_S)$ with respect to IV' .

Definition 7. BEHAVIORMODEL

A behavior model is defined as $BehaviorModel = \{BehaviorSpec, TAnnotation, BehaviorBody\}$, where:

- **BehaviorSpec** specifies the component behavior. This specification is given by component *interface contract*. A contract Ctr of interface Itf is a tuple $Ctr = \langle Itf, Init, Spec, Prot \rangle$, where:
 - Itf is the interface for which the contract is attached. It consists of two parts: the data declaration section, $Itf.D$, that introduces a set of variables x_i with their types T_j , and the method declaration section, $Itf.M$, that defines a set of method signatures. Each signature is of the form $m(x_1 : \mathbf{in} T_1; x_2 : \mathbf{out} T_2; x_3 : \mathbf{in out} T_3)$. A contract Ctr of interface Itf is denoted $Ctr.Itf = \langle Itf.D, Itf.M \rangle$.
 - $Init$ is the set of initial states of variables of interface declaration $Itf.D$. In other words, it is the allowable initial states depending on the initial condition Q over the variables of $Itf.D$. It is denoted $Ctr.Init = \langle q_0, \dots, q_n \rangle$. For example q_0 can be $x_1 = 0$. The empty condition is denoted $Ctr.Itf = \langle \rangle$.
 - $Spec$ is the functional specification of the contract. It assigns each method $m \in Itf.M$ a *static functionality* specification as pair of pre- and post-conditions of the form $Pre(x, Itf.D) \vdash Post(x, Itf.D, y', Itf.D')$, where non-primed and primed variables represent the values of the variables in the pre and post state of the execution of the method, respectively. If the precondition $Pre(x, Itf.D)$ is true, the pair will be abbreviated as $\vdash Post(x, Itf.D, y', Itf.D')$.
- The static functional specifications of operations are used to ensure that the user (the other components) provides correct inputs and the component returns with correct outputs. However, it does not force the order on the use of interface methods. In other words, it does not provide a *protocol* to coordinate the interactions between a component and its environment, which is specially important in real-time system.
- $Prot$ is the interaction protocol between the component with its environment. It is a set of sequences of operation requests, where a sequence is written in $?op_1(x_1), \dots, ?op_k(x_k)$ form, where $?op_i(x_i)$ is a call to operation $op_i \in Itf.M$ with the input value x_i .

In contrast to $Spec$, the protocols in the contracts are used to ensure *non-functional requirements*, such as time to avoid deadlock when putting components together.

Notice that the semantics of a protocol can be expressed by regular expression, a temporal logic or a trace logic formula. Here, we use the A Trace Logic for Local Security Properties [Corin et al., 2005]. According to this logic an *event* is a pair $\langle A : M \bowtie B \rangle$ where A, B are variables or constants, $\bowtie \in \{\triangleleft, \triangleright\}$, and M is a message, built from variables or constants. The event $\langle A : M \triangleright B \rangle$ should be read as “component port A sends message M with *intended destination* component

port B ". On the other hand, $\langle B : M \triangleleft A \rangle$ stands for " B receives message M apparently from B "

Let e , e_1 , and e_2 be events, F_1 and F_2 the pre-built and well-formed trace logic formulas and m be a message, a trace logic formula F is generated according to the following grammar:

$$F ::= \text{true} | \text{false} | F_1 \wedge F_2 | F_1 \rightarrow F_2 | \forall e : F | \exists e : F | \neg F | m | e_1 = e_2 | e_1 \neq e_2$$

- **TAnnotation** is a specific annotation language defined to express *non-functional temporal constraints* of *Prot*. They are built from common temporal requirement patterns shown in Section 2.5.1. Let t be the time that will hold between two events e_1 and e_2 . Let T_{begin} and T_{end} be the time interval limits, and D , D_{min} and D_{max} be respectively the exact, minimum and maximum delay, we consider the following annotations:

- $@after(T_{begin})$ i.e., $t \geq T_{begin}$;
- $@before(T_{end})$ i.e., $t \leq T_{end}$;
- $@mindelay(D_{min})$, i.e., $t \geq D_{min}$;
- $@maxdelay(D_{max})$, i.e., $t \leq D_{max}$;
- $@delay(D)$, i.e., $t = D$.

- **BehaviorBody** is an implementation of a contract of its interface *Intf* exposed in *BehaviorSpec* (see *BehaviorSpec*). The behavior body is hidden from the component environment, only the *interface contracts* of *BehaviorSpec* is visible from the component environment. To implement such a contract, the component may use operations provided by other components. These operations are called required operations and are specified as a contract of an interface that is called the required interface, denoted *RI*.

A *BehaviorBody* has the same structure as the interface contract, except a function *Code* that maps each method m of $\text{Ctr.Intf} = \langle \text{Itf.D}, \text{Itf.M} \rangle$ to its code. The *BehaviorBody* can also have private method declarations denoted *PrimDec* and its corresponding private function codes denoted *PrimCode*. As a consequence the structure of component body as *BehaviorBody* = (*Itf*, *Init*, *Code*, *PrimDec*, *PrimCode*, *RI*).

The *code* and *PrimCode* of *BehaviorBody* is mapped to the underlining programming language or to a *guarded design*. In fact according to result of [Hoare and He, 1998] any program can be abstracted as a guarded command, and further to a guarded design according to [He et al., 2006]. A guarded design is a pair of a *guard* g and a *design* D , denoted by $g \& D$ and defined by $D \triangleleft g \triangleright \text{idle}$ ². The guard g is used to describe the *availability* of the operation, i.e., the operation can be invoked only when g holds initially. The design D specifies the behaviors of execution for the operation once it is activated successfully. Without loss of generality, we always assume that the two functions *code* and *PrimCode* map each method to a guarded command.

²This is the shorthand of if g then D else Idle

Remark 4. It is important to note that our behavior model allows to distinguish *black-box* component from *gray-box* component. Black-box component only defines its BehaviorSpec, while gray-box component defines both BehaviorSpec and BehaviorBody. The *white-box* component is obtained when the code of BehaviorBody is mapped to the underlining programming language.

Example 6. Let consider that the connection type DC_1 shown in Figure 5.1, from which the data variable *appr* transit, is implemented by a basic buffer of integer. Figure 5.4 shows this basic buffer component with its syntactic interfaces : $Itf = \langle b : \mathbf{Seq}(int), put(item : \mathbf{in} int), get(appr : \mathbf{out} int) \rangle$. However, it does not provide the semantics information in the way the interface can be used. For example, it does not say that the buffer is a one-place buffer or not.

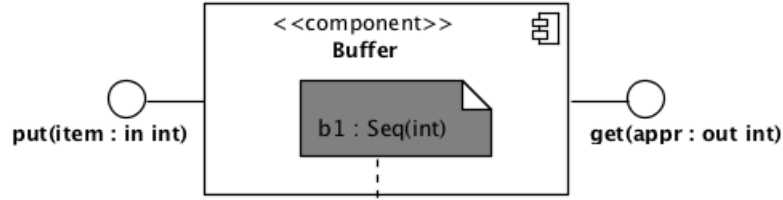


Figure 5.4: Example of black-box component: a buffer component which implements connection type DC_1 of Figure 5.1

Now, let suppose that a component is a one-place buffer, *i.e.*, only a single data value may held in the buffer at any one time. By using Definition 5, user can annotated it component instance with the *non-functional constraint*, *e.g.*, $Queue_Size=1$. Then, it can use the *interface contract* specification from Definition 7 to specify the semantics of buffer interface. We denote this interface contract by $B_Ctr = \langle Itf, Init, Spec, Prot \rangle$, where :

$$B_Ctr = \begin{cases} B_Ctr.Itf & \stackrel{df}{=} \langle b : \mathbf{Seq}(int), put(item : \mathbf{in} int), get(appr : \mathbf{out} int) \rangle \\ B_Ctr.Init & \stackrel{df}{=} |b| = 0 \\ B_Ctr.Spec(put) & \stackrel{df}{=} (\vdash b' = \langle item \rangle \bullet b) \\ B_Ctr.Spec(get) & \stackrel{df}{=} (\vdash b' = \mathbf{tail}(b) \wedge appr = \mathbf{head}(b)) \\ B_Ctr.Prot & \stackrel{df}{=} \{e_i \text{ is } ?put(x) \text{ if } i \text{ is odd and } ?get(y) \text{ otherwise}\} \end{cases} \quad (5.1)$$

Notice that at this stage a component is a *black-box* component with a syntactic and semantics interface sufficient for the component interaction with its environment. However, at this stage the component does not provide the semantics of composition, which is necessary to reason about composition in order to predict of applying the composition mechanism.

Definition 8. COMPOSITIONMODEL

As shown in the introduction of composition background (see Section 2.2), in order to reason about composition, we need a composition theory. Such a theory allows us to predict

the result of applying a composition mechanism to components. We have also shown that there are two kinds of composition mechanisms: *endogenous composition* and *exogenous composition*. In an endogenous composition mechanism, the handling of interaction between components is part of the components themselves, including ports and interfaces. Examples includes *plugging*, *Hiding* and *feedback* compositions. On the other hand, in an exogenous composition mechanism, the handling of interaction between components is realized with external *coordinator*.

As a consequence our composition model is defined as set of composition operators. Currently, the composition model is defined as a set $CompositionModel = \{PluggingCompos, HiddingCompos, FeedBackCompos, CoordinationCompos\}$.

Let $C_i = (PI\langle D, M \rangle, Init, Body\langle Code, PriMDec, PriMCode \rangle, RI)$ with $i \in \{1, 2\}$ be two gray-box components, where

- PI is an interface listing all the provided operations of C_i , including its declaration part and method specification.
- $Init$ is an initial condition that sets the initial value of the variables,
- $Code$ is a function that maps each method m of $Ctr.Itf = \langle Itf.D, Itf.M \rangle$ to its code, *i.e.*, underlining guarded command.
- $PriMDec$ is a set of method declarations that are internal to the component.
- $PriMCode$ is a function that maps each private method $m' \in PriMDec$ to its code.
- RI is an interface listing all the required operations of C_i .

we have the following definitions:

- **PluggingCompos** of C_1 and C_2 is denoted $C_1 \gg C_2$. If C_1 and C_2 have disjoint variable declarations, and none of the provided or private methods of C_2 appears in C_1 then $C_1 \gg C_2$ is a composite component that connects a provided operation of C_1 to a required operation of C_2 . It is defined by:

$$(C_1 \gg C_2) = \begin{cases} (C_1 \gg C_2).PI.D & \stackrel{df}{=} C_1.PI.D \cup C_2.PI.D \\ (C_1 \gg C_2).PI.M & \stackrel{df}{=} C_1.PI.M \cup C_2.PI.M \\ (C_1 \gg C_2).Init & \stackrel{df}{=} C_1.Init \wedge C_2.Init \\ (C_1 \gg C_2).Code & \stackrel{df}{=} C_1.Code \cup C_2.Code \\ (C_1 \gg C_2).PriMDec & \stackrel{df}{=} C_1.PriMDec \cup C_2.PriMDec \\ (C_1 \gg C_2).PriMCode & \stackrel{df}{=} C_1.PriMCode \cup C_2.PriMCode \\ (C_1 \gg C_2).RI.M & \stackrel{df}{=} (C_2.RI.M \setminus C_1.PI.M) \cup C_1.RI.M \end{cases} \quad (5.2)$$

An example of plugging composition is shown in Figure 5.5(a). After plugging, some methods can be hiding by the *hiding* operator.

- **HiddingCompos** of C with the regard of method $m \in C.Intf.M$ is denoted $C \setminus m$. It represents the composite component after removal of method m from it provided methods.

An example of hiding composition is shown in Figure 5.5(b). After hiding, it is possible that a component *feedbacks* some of its provided methods to some of its required methods.

- **Feedback** of C with the regard of method $m \in C.Intf.M$ and $n \in C.RI.M$ is denoted $C[m \hookrightarrow n]$. It represents the composite component which feeds back its provided operation m to the required operation n .

An example of feedback composition is shown in Figure 5.5(c). Obviously, feedbacking $C_1 \gg C_2$ consists to hide some of its provided and required methods. The above three composition operators can be successively applied as illustrated in Figure 5.5(d).

- **ParallelCompos** of C_1 and C_2 is denoted $C_1 \otimes C_2$. If $C_1.PI.D \cap C_2.PI.D = \emptyset$, and C_1 and C_2 do not share methods then $C_1 \otimes C_2$ is a composite component which has the provided operations of C_1 and C_2 as its provided operations, and the required operations of C_1 and C_2 as its required operations.

An example of disjoint parallel composition is shown in Figure 5.6(a). Plugging composition $C_1 \gg C_2$ is the same as disjoint parallel composition $C_1 \otimes C_2$ when the provided method of C_1 are disjoint from the required method of C_2 . The components we have studied until now are only *passive* components, *i.e.*, the component starts to execute when a provided method is called, and during the execution it may call operations of other components, and so on. In other words, passive basic or composite components provide a number of methods, but do not themselves activate the functionality specified in the contracts.

In general, specially in real-time systems, a component may be *active* as a task, *i.e.*, it has its own control and once it is started it can execute its internal actions, call operations of other components, and wait to be called by other components. As a consequence, we need the *active composite* components that implement a desired functionality by *coordinating* sequences of method calls.

- **CoordinationCompos** of disjoint union for a number of disjoint components $C_2 = C_i \otimes C_j, i, j = 3 \dots k$, by a coordinator C_1 is denoted by $C_1 \parallel_M C_2$. In fact, a coordination for C_2 is a component C_1 that calls a set of provided methods M of C_2 .

A coordinator interface $C_1.Itf$ has the same structure as an interface $Itf = \langle Itf.D, Itf.M \rangle$ shown in Definition 7, except that the semantics of Method $Itf.M$ is a set of method invocation signatures. Each of them is of the form $!m(x_1 : \mathbf{in} T_1; x_2 : \mathbf{out} T_2)$. In the same way, the contract of its interface as the same structure of component contract $Ctr = \langle Itf, Init, Spec, Prot \rangle$.

Examples of composition coordination are shown in Figure 5.6. In Figure 5.6(a), C_1 and C_2 are two one-place buffers. In Figure 5.6(b), *Cooordinator* is a task that keeps getting the item from C_1 and putting it to C_2 . It forms thus a three-place buffer in Figure 5.6(c). In Figure 5.6(d), the get of C_1 and the put of C_2 are *synchronized into an atomic step* by component C_4 . It forms thus a two-place buffer.

Remark 5. Notice that inspired by the work of Woodcock and Morgan [Woodcock and Morgan, 1990], Cheng. et al. have proven that $C_1 \parallel_M C_2$ is a component, and studied the algebraic laws of this composition [He et al., 2006]. The composition is defined similarly to the alphabetized parallel composition in CSP [Roscoe et al., 1997] with interleaving of events.

Example 7. In Example 6, we have shown a *black-box* component for a buffer simulation. Here, we review this buffer component as a *gray-box*, necessary to reason about composition semantics. Let consider the formal definition of two gray-box buffer components C_1 and C_2 as follows.

$$C1 = \begin{cases} C1.Itf.D & \stackrel{df}{=} \{b_1 : \mathbf{Seq}(int)\} \\ C1.Itf.M & \stackrel{df}{=} \{put(x : \mathbf{in} \ int), get_1(x : \mathbf{out} \ int)\} \\ C1.Body.Code(put) & \stackrel{df}{=} (b_1 := \langle x \rangle) \triangleleft (b_1 = \langle \rangle) \triangleright (put_1(\mathbf{head}(b_1)); b_1 := \langle x \rangle) \\ C1.Body.Code(get_1) & \stackrel{df}{=} (b_1 \neq \langle \rangle) \longrightarrow (y := \mathbf{head}(b_1)); b_1 = \langle \rangle) \\ C1.Body.MDec & \stackrel{df}{=} \{put_1(x : \mathbf{in} \ int)\} \end{cases}$$

$$C2 = \begin{cases} C2.Itf.D & \stackrel{df}{=} \{b_2 : \mathbf{Seq}(int)\} \\ C2.Itf.M & \stackrel{df}{=} \{put_1(x : \mathbf{in} \ int), get(x : \mathbf{out} \ int)\} \\ C2.Body.Code(get) & \stackrel{df}{=} (y := \mathbf{head}(b_2)); b_2 := \langle x \rangle \triangleleft (b_2 \neq \langle \rangle) \triangleright (b_2 := get_1(y)) \\ C2.Body.Code(get_1) & \stackrel{df}{=} (b_2 \neq \langle \rangle) \longrightarrow (b_2 \neq \langle x \rangle) \\ C2.Body.MDec & \stackrel{df}{=} \{get_1(y : \mathbf{in} \ int)\} \end{cases}$$

Then, the plugging composite component $C_1 \gg C_2$ is shown in Figure 5.5(a). By hiding the method get_1 in $C_1 \gg C_2$, the hiding composite *i.e.*, $C_1 \gg C_2 \setminus get_1$ is shown in Figure 5.5(b) and so on.

Remark 6. The formal model of component contract Ctr introduced here can be represented by UML diagrams. Graphically, the interaction protocol of the contract $Ctr.Prot$ can be represented as a *UML sequence diagram*, while the flow of method invocations and synchronization in the behavior body of a component can be modeled by a *UML state diagram*. The static functionality of interface methods, *i.e.*, $Ctr.Spec$ is given in terms of their *pre- and*

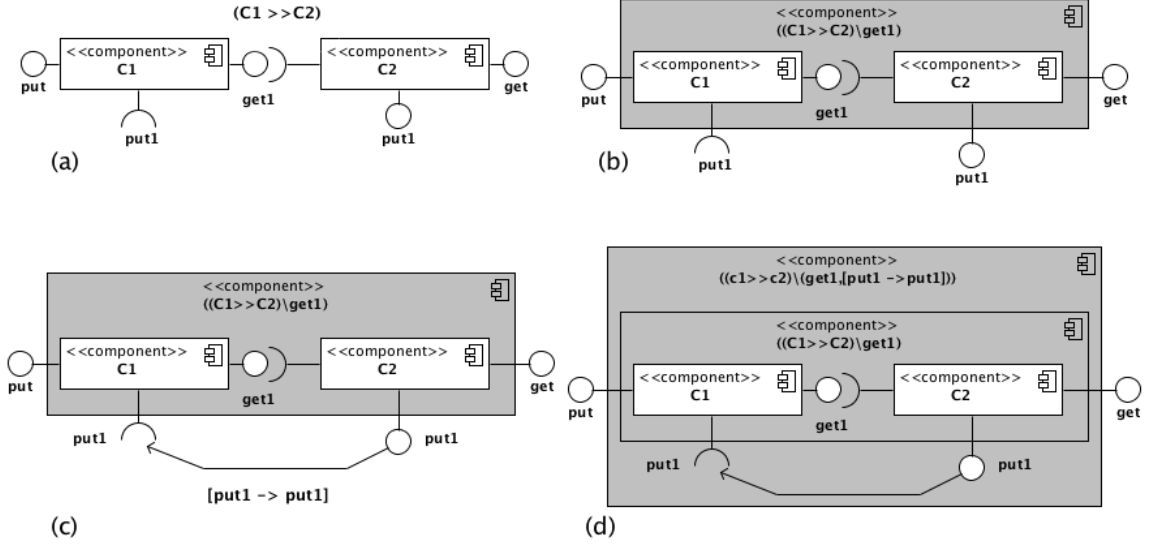


Figure 5.5: Examples of passive composition: (a) plugging composition, (b) hiding after plugging, (c) feedback, (d) hiding after feedback

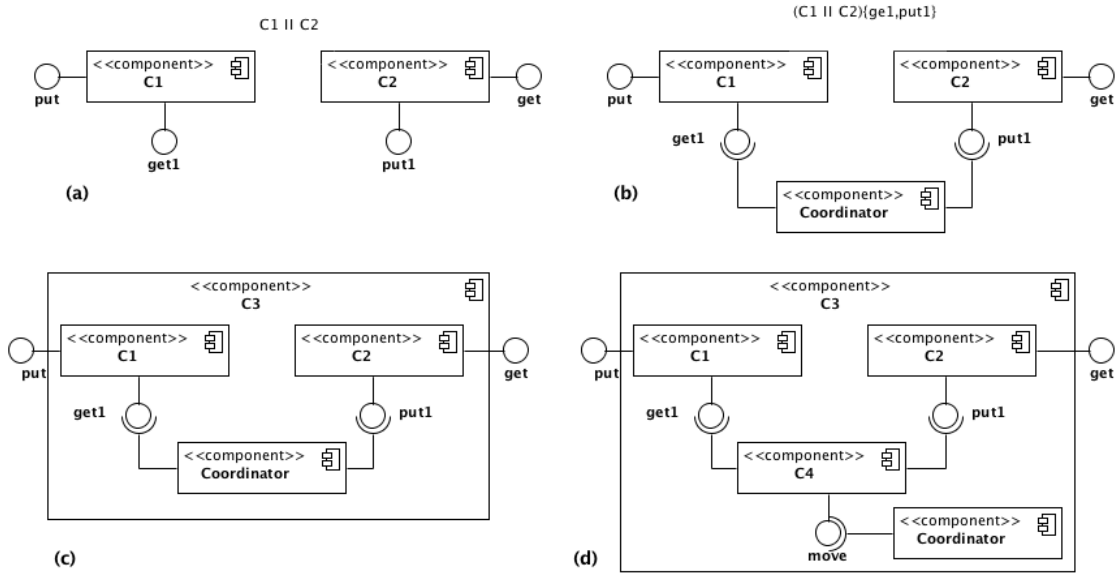


Figure 5.6: Examples of active composition: (c) coordination of two one-place buffers to define a three-place buffer, (d) coordination of two one-place buffers to define a two-place buffer

post-conditions, while the dynamic functionality of interface protocol *i.e.*, *Ctr.Prot* is given in terms of specific time annotations. Indeed, as we will see in the transformation phase, Section 5.3.3, the UML state machine is extended with the predefined annotations, named *TAnnotation* in Definition 7.

Proposition 1. SEPARATION OF CONCERNS AND MIX VERIFICATION

Formally speaking, the separation of *Ctr.Spec* and *Ctr.Prot* requires that the traces of the sequence diagram are accepted by the state machine defined by the state diagram.

Informally, the consistency of separately specified behavior in the sequence diagram and the state diagram must ensure that whenever the components follow the interaction protocol defined by the sequence diagram, the interactions will not be blocked by the system.

Proof.

First, *static consistency* between methods in the diagrams and the functional specification must be checked. This step is usually done by tools like a compiler.

Second, *dynamic consistency* ensures that the separately specified behavior in the sequence diagram and the state diagram are consistent:

- Formal deductive proof techniques, such as [SPARK, 2014], aided by a theorem prover can be used to prove some functional properties, *i.e.*, pre- and post-conditions of the methods. Verification of the functional properties, particularly its completeness, is a difficult issue. Some times it must be completed by inner invariants, such as the loop invariants, as proposed in [Aponte et al., 2012].
- Model checking techniques is used to check that the state machine defined by the state diagram accepts the time trace of protocol properties. In OBV context, we translated both the sequence diagram and the state diagram into TAIIO model (see Section 5.3.3). Then, we used the UPPAAL TAIIO model checker tool to check that the TAIIO model for the state diagram is trace equivalent to the TAIIO model for the sequence diagram.

□

5.3.2 TAIIO formal definition

Timed Automata (TA) is one of the most popular models adapted to real-time systems [Alur and Dill, 1994]. First, the TA model is well adapted for the verification of real-time properties because it defines a timed labeled transition system (TLTS). Second, a number of methods based on variants of the TA model, such as TA over Input and Output actions (TAIO) [Krichen and Tripakis, 2009], or other similar models, such as timed Petri nets [Merlin and Farber, 1976] have been proposed. Finally, a number of automatic model checker tools for TA have been efficiently developed, *e.g.*, Kronos [Yovine, 1997] and UPPAAL [Larsen et al., 1997]. For example, UPPAAL uses an extended version of TA, called UPPAAL TA, to specify a system as a network of TA models and to globally declare all variables in the declaration part for the synchronization of TA models. UPPAAL TRON is an extension of UPPAAL for conformance testing of real-time systems by using relative timed input/output conformance relation [Larsen et al., 2005]. We translate our SARA model into

the TAI0 model based on UPPAAL TA, and use the UPPAAL TRON tool for verification tasks.

Definition 9. TASYs

A TA verification system can be defined as a tuple $TASys = (TAModel, TADeclaration)$, where:

- **TAModel** is a set of all the TA models used in the global system model. Every TA model is defined according to the specific TAI0 (see Definition 10);.
- **TADeclaration** is the declaration part that contains all the variables of the system models (see Definition 15).

Definition 10. TAIOMODEL

A TAI0 model is defined as a tuple $TAIOModel = (TAIOName, TAIOSyntax, TAIOSemantic)$, where:

- **TAIOName** is the name of the TAI0 model, which appears in the system declaration part and can be used to arrange priorities on TAModels;
- **TAIOSyntax** is the syntax of TAI0 (see Definition 11);
- **TAIOSemantic** is the semantics of TAI0 (see Definition 12).

Definition 11. TAIOSYNTAX

A TAI0 A over input and output action Act is a tuple $A = (L, l_0, X, Inv, Act, E)$, where:

- L is a finite set of *locations*;
- $l_0 \in L$ is the initial location;
- X is a finite set of non-negative real valued *clocks*, $\{x_1, x_2, \dots, x_n\}$;
- Inv is a function over clock valuation \mathbb{R}_+^X that assigns an invariant to each location;
- $Act = Act \times \{!, ?\}$ is a set of the partitionned set of *observable* actions $Act = Act_{in} \cup Act_{out}$. Input actions are denoted $a?, b?$, etc, and output actions are denoted $a!, b!$, etc;
- E is a finite set of *edges* for transitions.

Each edge E is a tuple (l, g, r, d, a, l') , where:

- $l, l' \in L$ are respectively the source and destination locations;
- g is a set of time constraints of the form $x * c$, where $x \in X$ is a clock variable, c is an integer constant and $*$ $\in \{<, \leq, =, \geq, >\}$;

- $r \in X$ is a set of clocks to reset to zero, ($r := 0$);
- $d \in \{\text{delayable}, \text{eager}, \text{lazy}\}$ is the deadline which can be assigned to E.
- $a \in \text{Act}$ is the action.

Definition 12. TAIOSEMANTIC

A TAIIO $A = (L, l_0, X, \text{Act}, E)$ defines a Timed Input-Output Labeled Transition System (TIOLTS) [Krichen and Tripakis, 2009]. A TIOLTS is a tuple $(S^A, s_0, \text{Act}, T_d, T_t)$, where:

- $S^A = L \times \mathbb{R}_+^X$ is a set of *time states* associated to locations of A . A timed state is a pair $s = (l, v)$, where $l \in L$ and $v : X \rightarrow \mathbb{R}_+$ is a positive real clock valuation;
- $s_0 = (l_0, \vec{0})$ is the initial state, where $\vec{0}$ is the valuation assigning 0 to every clock of A ;
- T_d is a set of *discrete transitions* of the form $(s, a, s') = (s', v) \xrightarrow{a} (s', v')$, where $a \in \text{Act}$ and there is an edge $E = (l, g, r, d, a, l')$, such that v satisfies g and v' is obtained by resetting to zero all clocks in r and leaving the others unchanged;
- T_t is a set of *timed transitions* of the form $(s, t, s') = (s', v) \xrightarrow{t} (s', v + t)$, where $t \in \mathbb{R}_+^*$. T_t must satisfy the following conditions:
 - $(s, t, s') \in T_t \wedge (s, t, s'') \in T_t \Rightarrow s' = s''$,
 - $(s, t, s') \in T_t \wedge (s', t', s'') \in T_t \Rightarrow (s, t' + t, s'') \in T_t$,
 - $(s, t, s') \in T_t \Rightarrow \forall t' < t, \exists (s, t', s'') \in T_t$.

As mentioned in the TAIOSyntax, $\text{Act} = \text{Act}_{in} \cup \text{Act}_{out}$ are observable actions. We assume there is an unobservable action $\tau \notin \text{Act}$. For $(s_0, v_0), (s, v), (s', v') \in S^A, a_i \in \text{Act} \cup \mathbb{R}_+$ and $\tau_a \in \tau$ a sequence of the form $(s_0, v_0) \xrightarrow{a_1=a} (s, v) \xrightarrow{x_1=2} \dots \xrightarrow{\tau_a} \dots \xrightarrow{a_2=b} (s', v')$ is called a *run* of S^A . The *time trace* $TTrace$ of this run is $TTrace(S^A) = a_2\tau_a b$. In general, $TTrace$ is the *real-time sequence* over set $(\text{Act} \cup \mathbb{R}_+ \cup \tau)^*$.

Definition 13. OBSERVABLE TIME TRACE

Given a time trace $\gamma \in TTrace(S^A)$, there exists an observable time trace $\sigma \in \text{Obs}TTrace$ such that σ is the projection of γ to $\text{Act} \cup \mathbb{R}_+$, i.e., obtained by erasing from γ all actions not in $\text{Act} \cup \mathbb{R}_+$. Clearly, if $\gamma = \tau^{j_1} a_{i_1} \dots \tau^{j_N} a_{i_N}$ then $\sigma = a_{i_1} \dots a_{i_N}$. For example, the $\text{Obs}TTrace$ of above $TTrace(S^A)$ is $\sigma = a_2 b$. The time spent in a sequence, denoted $\text{time}(\sigma)$ is the sum of all delays in σ . For example $\text{time}(a_2 b) = 2$.

Definition 14. REACHABILITY STATE

A state $s \in S^A$ is *reachable* if there exists a time trace $\gamma \in TTrace(S^A)$ such that $s_0 \xrightarrow{\gamma} s$. The set of reachable states of S^A is denoted $\text{Reach}(S^A)$. A maximal set of reachable states that the system can be active is denoted a state configuration $\text{Reach}(C_{S^A})$

Definition 15. TADeclaration

A TADeclaration is defined as a set of three parts, TADeclaration = {GlobalDecl, ModelDecl, SysDecl}, where:

- GlobalDecl contains global integer variables, clocks, synchronization channels and constants for TASys.
- ModelDecl defines local variables, channels and constants for each TAI0 model.
- SysDecl defines the execution order by assigning priority to TAI0 models.

5.3.3 From SARA model to TAI0 Model

This section presents the transformation rules developed to translate a SARA model into a TAI0 model, for which we can use UPPAAL model checker tool for formal verification of real-time properties. They are based on the above SARA and TAI0 formal model definitions. We first present the transformation rules. Then, we present the semantics preservation of model transformation.

Transformation rules

There are approaches, such as [Varró, 2002, Mekki, 2012], that transform UML State Machine (UML SM) into Extended Hierarchical Automaton (EHA) and Timed Automata (TA). Although the behavior model of our SARA model is based on the UML SM with temporal annotations (see Remark 6), the transformation of UML SM into TA is not sufficient for our component model transformation. Based on these approaches and with some extensions, we develop the following transformation rules to translate a SARA model into a TAI0 model.

Rule 1. (mapping of component instances). The objective of this rule is to transform a behaviour of a SARA component instance into a TAI0 Model. For each CompInst = (CompInstName_i, Priority, BehaviorModel):

- **Rule 1.1** Based on the bi-simulation relation between UML SM and TAI0 model [Mekki, 2012], we encode a corresponding TAI0Model of BehaviorModel defined with UML State Machine (UML SM). For example, Figure 5.7(a) shows the UML SM diagram for the Gate component of Figure 5.1. In addition, the state machine is extended with predefined annotations, shown in Definition 7. For instance, the state annotations @maxdelay(4s) and @maxdelay(6s) in the states *Signaling* and *Closing* specified that the Gate responds to *down* signal by moving down and takes at most 10s to be completely closed. Indeed, with transition annotations @delay(4s) and @delay(6s), it takes 4s to activate the light warning when a vehicle approaches the gate, and 6s to close the gate. Conversely, it responds to the *up* signal by moving up and it takes 6s to be completely opened. The corresponding UPPAAL TAI0 model is shown Figure 5.7(b).

- **Rule 1.2** Each annotation is translated into an integer expression. For example, $@after(T_{begin})$, $@before(T_{end})$, $@mindelay(D_{min})$, $@maxdelay(D_{max})$, $@delay(D)$, are respectively translated into $x \geq T_{begin}$, $x \leq T_{end}$, $x \geq D_{min}$, $x \leq D_{max}$, $x = D$, as illustrated in Figure 5.7(b).

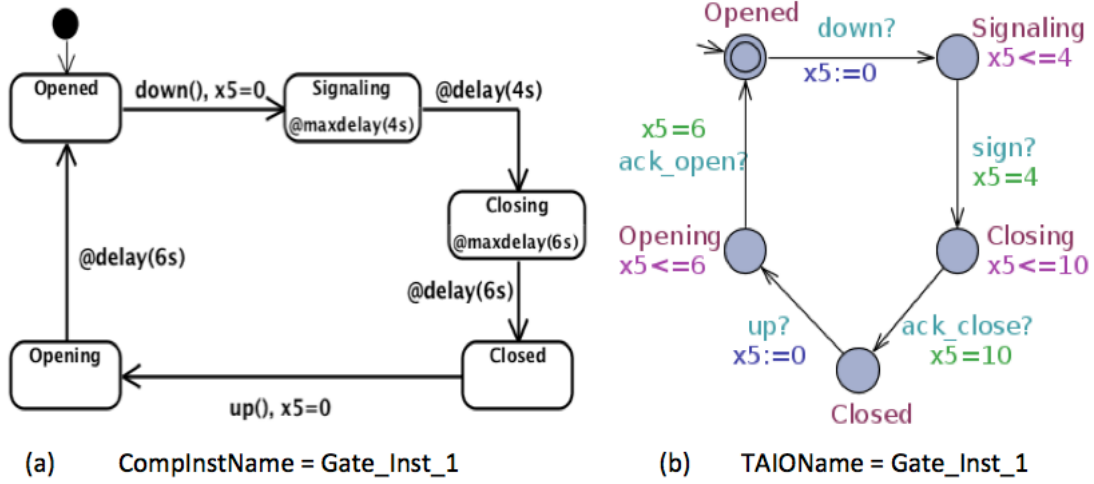


Figure 5.7: (a) Gate component UML SM, (b) The corresponding UPPAAL TAIO

Example 8. Figure 5.7 illustrates the transformation Rule 1.

Rule 2. (mapping of bindings). The objective of this rule is to transform component instance bindings. For each $\text{CompInstBinding} = (\text{CompInstName}_i, \text{Priority}, \text{CompConnect})$, where $\text{CompConnect} = (\text{IC}, \text{DC}, \text{OC})$, a TAIOModel is inserted in TASys by respecting the following rule parts:

- **Rule 2.1** For each $\text{IC}_n : \text{ModelName}.iv_j \rightarrow \text{CompInstName}_k.ip_l$, insert a TAIOModel $(L, l_0, V, \text{Act}, \text{Clock}, \text{Inv}, T)$ with name DC_n , where $T = \{(q, g, r, a, q')\}$, with $g = \{iv_j \text{ XOR } ip_l\}$, and $a = \{ip_l := iv_j\}$. XOR represents the inequality function between iv_j and ip_l to avoid modification during the transition. This leads to update ip_l and achieves the connection, as illustrated in IC_1 of Figure 5.8;
- **Rule 2.2** For each $\text{DC}_n : \text{CompInstName}_i.op_j \rightarrow \text{CompInstName}_k.ip_l$, insert a TAIOModel with name DC_n , where $g = \{op_j \text{ XOR } ip_l\}$ and $a = \{ip_l := op_j\}$, as illustrated in DC_1 of Figure 5.8;
- **Rule 2.3** For each $\text{OC}_n : \text{CompInstName}_k.op_i \rightarrow \text{ModelName}.ov_j$, insert a TAIOModel with name OC_n , where $g = \{op_i \text{ XOR } ov_j\}$, and $a = \{ov_j := op_i\}$, as illustrated in OC_1 of Figure 5.8.

Example 9. Figure 5.8 illustrates the transformation Rule 2 and Rule 3



Figure 5.8: Example of TAI0 connections for Figure 5.1

Rule 3. (Mapping of initial input and adapter). The objective of this rule is to initialize the model in order to check it with a TAsys tool, such as the UPPAAL-TRON tool. The adapter in UPPAAL-TRON is a driver which interprets the testing input for the system under observation, and in turn translates the output for TRON [Larsen et al., 2005]. For each $ip_j \in IP$ in SARAModel, insert TAIOModel with name $INIT_n$ and the action to update is $a = \{ip_j := ip_j?\}$, as illustrated in $INIT1$ of Figure 5.8. For each input $ip_j?$, an input/output adapter stub can be generated. For example, $ADAPT1$ in Figure 5.8 shows the delay adapter between the input port and the output port.

Rule 4. (mapping of declarations). The objective of this rule is to transform the declarations of a SARAModel into a TADeclaration = (GlobalDecl, ModelDecl, SysDecl). It is defined as follows:

- **Rule 4.1** For DataModel={IV, OV, LV, TV} in SARAModel, insert $iv \in IV$, $ov \in OV$ and $tv \in TV$ in GlobalDecl.
- **Rule 4.2** As mentioned in Definition 4, for technical reasons, we assume that all the local variables of model components occur somewhere in the top level component structure. The distinction of each component variable is ensured by prefixing each local variable of each component with the name of component. For each $lv \in LV$ with a prefix insert lv in ModelDecl.
- **Rule 4.3** For each declaration of $CompInst = (CompInstName_i, Priority, Behavior-Model)$ insert in SysDecl the $TAIName = CompInstName_i$. Based on the priorities of different component instances, arrange priorities on different TAIName of TAI0 models. This define the execution order between the different TAI0 models. The execution path begins at input ports and ends at output ports by following the priority order of component instances. For example, the execution order of Figure 5.1 is: $OC_i < Gate_Inst_1 < DC_i < Controller_Inst_1 < DC_j < Sensor_Inst_1 < IC_i$, where "<" is the execution order operator. This means that $Sensor_Inst_1$ has a highest priority than $Controller_Inst_1$ and $Controller_Inst_1$ has a highest priority than $Gate_Inst_1$.

Example 10. Figure 5.9 illustrates the transformation Rule 4 for the declaration part of LC-APS model of Figure 5.1.

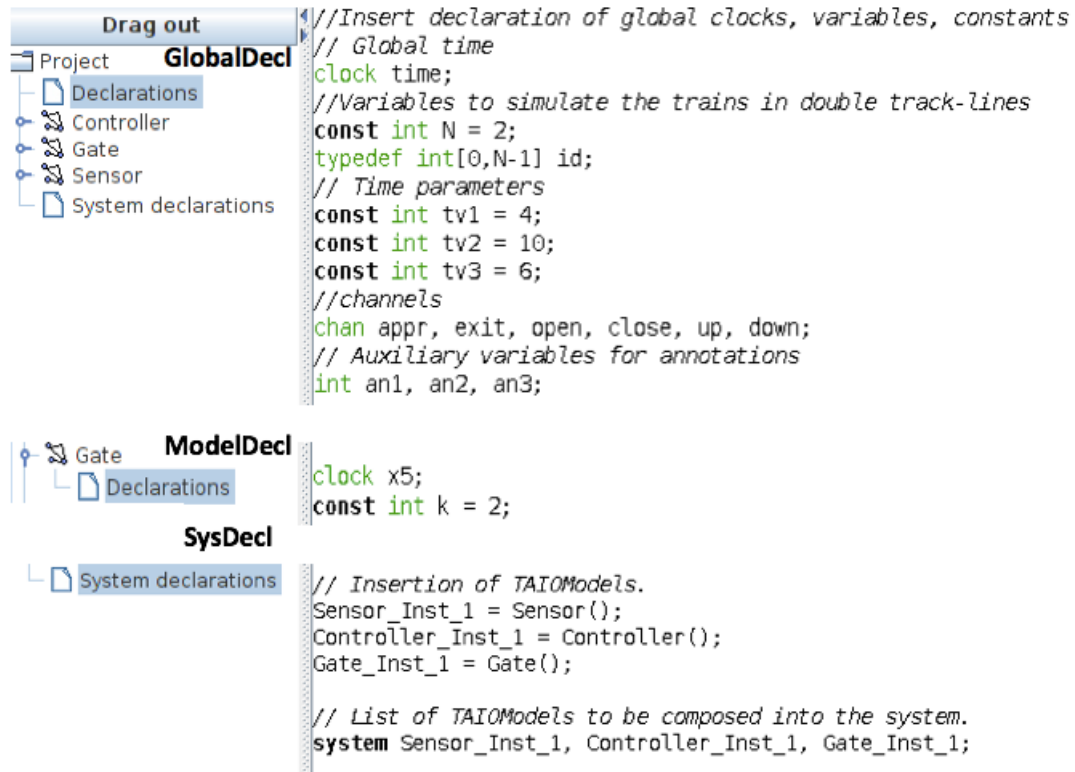


Figure 5.9: UPPAAL model declaration of Figure 5.1

Remark 7. Technically, it is important to note that the transformation links are maintained, such as Relational Database Management System, in order to use these links to trace the equivalence relations for the semantics preservation during model transformation.

Semantics preservation of model transformation

Many approaches for semantics preservation of model transformations exist [Giese et al., 2006]. Either specific correctness conditions are checked for both the original source model and its transformation model [Varró and Pataricza, 2003] or the semantics equivalence between both models is guaranteed by a bisimulation check [Narayanan and Karsai, 2008]. The former approach verifies if a transformation preserves certain dynamic consistency properties by model checking the source and target models for properties P and Q , where property P in the source language is transformed into property Q in the target language. This transformation requires validation by a human expert. The latter approach checks if the models are equivalent with respect to a particular property about the source model. Then, verifying the target model for the property will be equivalent to verifying the source model for this property. As our target model is used for observer-based verification, we study the second approach that provides a way to check if a *bisimulation* equivalence relation exists between an instance of the SARA model and an instance of the TAIOM model.

Bisimilarity

Two systems can be said to be bisimilar if one system simulates the other and vice-versa. A bisimulation relation can be defined formally as follows.

Definition 16. GENERAL DEFINITION OF BISIMILARITY

Given a labeled state transition system $(S, \Lambda, \rightarrow)$, a *bisimulation relation* is defined as an equivalence relation R over S , such that for all $p, q \in S$, if (p, q) is in R , and for all $p' \in S$ and $\alpha \in \Lambda$, $p \rightarrow^\alpha p'$ implies that there exists a $q' \in S$ such that $q \rightarrow^\alpha q'$ and (p', q') is in R , and conversely, for all $q' \in S$ and $q \rightarrow^\alpha q'$ implies $p \rightarrow^\alpha p'$ and (p', q') is in R .

This definition is given in terms of a single global set S . We can easily think of its equivalence for two transition systems where a global set contains both the source states and the target states. Based on Definition 6 and Definition 14, we provide its following equivalence definition for the bisimulation relation between the SARA source model and the TAIO target model.

Definition 17. ADAPTED DEFINITION OF BISIMILARITY

Given an equivalence relation R that matches a component configuration C_S in the SARA model and creates its equivalent state configuration C_T in the TAIO model, the equivalence is a bisimulation relation if for each transition t from C_S to C'_S in the SARA, there exists an equivalent transition t' in the TAIO from C_T to C'_T , and C'_T is equivalent to C'_S , and vice versa.

In this way, if there is a bisimulation relation between a SARA model and its corresponding TAIO model, then verifying the TAIO model for reachability will be equivalent to verifying the SARA source model for reachability. If the check fails, it means that there was an error in the transformation and the generated TAIO model does not truly represent the SARA source model.

Here, we check the behavioral equivalence of the SARA model and the TAIO model with respect to reachability. It should be noted that the proposition described below is not an attempt to prove the correctness of the transformation rules in general. This is a method to verify if a specific transformation from an instance of the SARA model into an instance of the TAIO model is valid. It must be executed for each transformation individually.

Proposition 2. CHECKING BISIMILARITY WITH RESPECT TO REACHABILITY

If a transformation relation R , which matches a component configuration C_S in the SARA model and creates its equivalent state configuration C_T in the TAIO model is a bisimulation, then verifying the TAIO model for reachability will be equivalent to verifying the SARA model for reachability.

Proof. The proof is realized by construction. Indeed, most of the TAIO model checking tools provide a means to check the reachability of a state configuration according to Definition

14. Alternatively, a claim can be made in the tool to say that the state S^A of TAO A is not reachable. If it is indeed reachable, the model checker refutes this claim and presents a counter-example, as a time trace $\gamma \in TTrace(S^A)$. This trace represents a valid series of transitions $s_0 \xrightarrow{\gamma} s$ in the TAO that leads to a state configuration C_T . Let $s_1 \xrightarrow{\gamma_i} s'_1$ be a transition in a series of transitions $s_0 \xrightarrow{\gamma} s$, such that $s_1 \in C_T$ and $s'_1 \in C'_T$. For each transition $s_1 \xrightarrow{\gamma_i} s'_1$ in a series of transitions $s_0 \xrightarrow{\gamma} s$, we search the equivalent transition in the SARA model configurations C_S and C'_S for bisimulation according to Definition 17. By construction, the transformation links created during the transformation are used to reproduce this trace in the SARA model. Rather than checking for all possible states in the SARA model, Definition 6 for input enabled and reachable configuration are used. In this way, verifying the TAO model for reachability will be equivalent to verify the SARA model for reachability. \square

Remark 8. The complexity of the transformation is not increased significantly by this method, as shown in the metrics of model transformation in Section 6.6. Indeed, as the transformation links are created every time the objects of the output model are created from the input model, the complexity of the transformation checking is proportional to the size of the input model and to the transformation engine.

5.4 A 3-Layer Approach for OBV

Figure 5.10 shows the overview of our component-based modeling and observer-based verification approach with separation of concerns. It is a 3-layer approach where the *upper layer*, the *middle layer* and the *lower layer* represent the requirement model, the solution model and the verification model, respectively.

5.4.1 Upper layer

The upper layer represents software requirement specification, where functional requirements are separated from non-functional requirements, such as safety requirements subjected to real-time constraints, called here temporal safety requirements. Our approach is based on common temporal requirement patterns shown in Section 2.5.1. However, the justification of pattern selection is a major challenge of pattern-based approach because generally, complex requirement specifications match more than one of patterns mentioned, as stated in Remark 1. For that purpose, we state the following assumption for our approach.

Assumption 2. DECOMPOSITION OF COMPLEX REQUIREMENTS

Complex requirements to be verified have to be decomposed into less complex requirements, which can be easily assigned to common pattern specifications.

We justify this assumption, by the fact that the assignment of requirements in the correct patterns is a first difficulty in the adoption of formal pattern-based approaches by non-

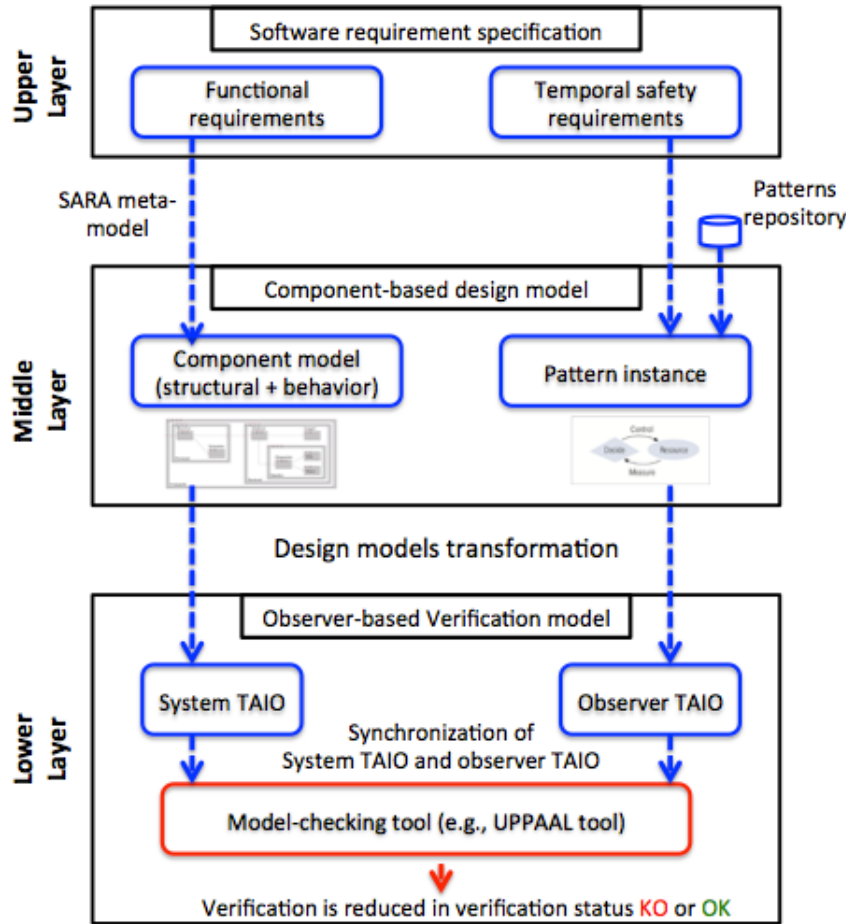


Figure 5.10: The overview of our OBV approach

experts. This assignment can be addressed by domain experts by using requirement diagrams, such as SysML requirement diagram [SysML, 2012]. With these methods, it is possible to simplify complex requirements in a greater number of less complex requirements. The simplified requirements, in turn, can be easily assigned to common pattern specifications or new pattern specifications. The consideration here is focused on the second difficulty, which consists in supporting formal verification non-experts with expert knowledge and experiences in the verification of pattern specifications. The main advantage of our approach in comparison to other approaches is that supporting non-experts with pattern specifications transfers expert knowledge. Therefore, it is an answer to the recurrent statement, common software engineers are not experts in logic. Once requirements to be verified have been assigned to the correct property specification patterns, our integrated approach can be processed as follows.

5.4.2 Middle layer

The middle layer describes the design model. Functional requirements are modeled according to the SARA model, which meta-model and formal model are introduced in Section 4.3 and Section 5.3.1, respectively. For time annotation properties to be verified, we select the appropriate observer patterns from the generic repository patterns presented in Section 2.5.1. For example, Figure 5.11 shows two observers. Figure 5.11(a) represents the absence before observer. The forbidden after observer can be easily derived from it. Figure 5.11(b) represents the response max delay pattern. In the same way, the response min delay can be easily derived from it.

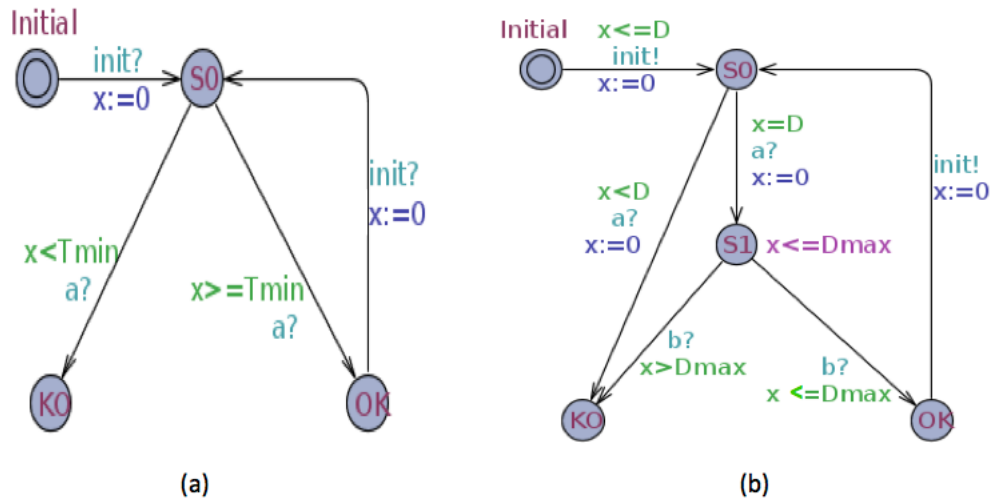


Figure 5.11: (a) Absence before observer, (b) Response max delay observer

Once identified, the patterns are instantiated with the appropriate parameters. For example, for requirement specification “when the gate is *opened* to road traffic, it must stay open at least T_{min} time units before closing”, from Example 5, the absence before observer in Figure 5.11 (a) is instantiated with states *opened* instead of *s0*, and with transition *down?* and *up?* instead of *a?* and *init?*, respectively. Here, the different types of observers are mainly informative, since nothing prohibits the mix of different types of observers. What we want really is to show that any execution trace of the observed system is preserved in the composition of the system and its observer. For this, we show that the well-defined observer does not obstruct a behavior of the system.

5.4.3 Lower layer

The lower layer represents the verification model, where the SARA design models are transformed into TAIIO models (see Section 5.3). The pattern instance is also represented into observer TAIIO model. The two TAIIO are synchronized both on time and on their shared common actions according to the parallel composition of TAIIO models.

We show that, from any trace of the composition system and its observer $S \parallel O$, we can obtain a trace of S by erasing the events from the well-defined observer O . This means that the well-defined observer does not add new behaviors to the observed system S . Let two traces σ_1 and σ_2 in $ObsTTrace(S)$ and $ObsTTrace(O)$, respectively. They are said to be *synchronizable* in $S \parallel O$ if there exists a run λ (see Definition 12) of $S \parallel O$ from which the two traces can be extracted. In other words, it can be stated in the following proposition.

Proposition 3. SOUNDNESS OF OBV

if $\sigma_1 \in ObsTTrace(S)$ and $\sigma_2 \in ObsTTrace(O)$ are the traces extracted from a run λ of $S \parallel O$, where $\sigma \in ObsTTrace(S \parallel O)$ then σ_1 and σ_2 are synchronizable in two composition of TIOLTS $S' \parallel O'$ with the same sets of input and output as S and O , and $\sigma \in ObsTTrace(S' \parallel O')$.

Proof. Let S' have the same sets of inputs Act_{in}^S and outputs Act_{out}^S as S and let O' have the same sets of inputs Act_{in}^O and outputs Act_{out}^O as O . Moreover, let S' and O' *synchronize* on the same set of actions Act_{syn}^S and Act_{syn}^O , respectively. Let $Act^S = Act_{in}^S \cup Act_{out}^S \cup Act_{syn}^S$ and $Act^O = Act_{in}^O \cup Act_{out}^O \cup Act_{syn}^O$. With no loss of generality, we also assume that the four TIOLTS have the same unobservable action τ .

Since $\sigma_1 \in ObsTTrace(S)$, according to Definition 13 there exists $\gamma_1 \in TTrace(S)$ such that σ_1 is the projection of γ_1 to Act^S , i.e., obtained by erasing from γ_1 all actions not in Act^S . Thus, there exist $[i_1, \dots, i_N]$ and $[j_1, \dots, j_N]$ such that $\gamma_1 = \tau^{j_1} a_{i_1} \dots \tau^{j_N} a_{i_N}$. Clearly, $\sigma_1 = a_{i_1} \dots a_{i_N}$.

In the same way, since $\sigma_2 \in ObsTTrace(O)$, there exists $\gamma_2 \in TTrace(O)$ such that σ_2 is the projection of γ_2 to Act^O . Thus, there exist $[k_1, \dots, k_N]$ and $[l_1, \dots, l_N]$ such that $\gamma_2 = \tau^{l_1} a_{k_1} \dots \tau^{l_N} a_{k_N}$. Clearly, $\sigma_2 = a_{k_1} \dots a_{k_N}$.

Since σ_1 and σ_2 are extracted from λ , it is possible to synchronize the traces γ_1 and γ_2 in $S \parallel O$ by considering the trace $\beta = \tau^{r_1} a_1 \dots \tau^{r_N} a_n \in ObsTTrace(S \parallel O)$, where (1) $a_1 \dots a_n = \gamma$ obtained by erasing from λ all actions not in $Act^S \cup Act^O$, (2) for $p = 1, \dots, n$, if $a_p = a_{i_s} = a_{k_t}$ appears in both γ_1 and γ_2 then $r_p = j_s + l_t$; if $a_p = a_{i_s}$ appears only in γ_1 then $r_p = j_s$; and if $a_p = a_{k_t}$ appears only in γ_2 then then $r_p = l_t$.

Then in the same way, it is possible to synchronize γ'_1 and γ'_2 in $S' \parallel O'$ by considering the trace $\beta' = \tau^{r'_1} a_1 \dots \tau^{r'_N} a_n \in ObsTTrace(S' \parallel O')$, where r'_p is defined similarly to r'_p . Hence, we are done. \square

At the end, from the composition of observed system and its observers, we use an appropriate verification tool, such as the UPPAAL model checker for the verification tasks. The verification tasks are reduced to a reachability search of error or no-error states (KO or OK states) on the TAIIO models, as illustrated in our case studies in Chapter 6.

5.5 Challenges Revisited and Lessons Learned

5.5.1 Challenges Revisited

We have presented in this chapter an observer-based verification approach for component-based models. Let us now revisit the specific challenges identified in Section 5.2.2 and discuss how our approach faces them.

- C4. **Separation of component interface contracts.** To face this challenge, we have distinguished the input and output contracts of component interface methods, expressed as pre- and post-conditions, from the real-time constraints of component interaction with its environment, expressed as interface contract protocol. The idea of pre- and postcondition is widely used in a variety of software techniques, such as design by contract [Meyer, 1997], and Uml 2.0 OCL specification [OMG, 2003]. The novel features of our approach is the definition of temporal annotation language for interaction protocol that imposes the order of the use of interface methods. This annotation language are built from real-time extensions of the Dwyer et al. classification shown in Section 2.5.1.
- C5. **Gray-box specification.** To face this challenge, the specification of SARA component is defined as a gray-box view, *i.e.*, a component is specified as an architecture that implements a behavior specification. Black-box component only defines its behavior specification, while gray-box component defines both Behavior specification and behavior body. The knowledge of behavior body is important to reason about component composition, *i.e.*, to predict the result of applying a composition mechanism.
- C6. **Assurance of model transformation.** To face this challenge, we have formally defined the SARA model and the TAI0 model in such a way it facilitate it transformation process. Based on these formal definitions, we define a simple transformation rules that map SARA elements to TAI0 elements. Then, we check a bisimilarity relation between the two models with respect to the reachability property. If there is a bisimulation relation between a SARA model and its corresponding TAI0 model, then verifying the TAI0 model for reachability will be equivalent to verifying the SARA source model for reachability. If the check fails, it means that there was an error in the transformation and the generated TAI0 model does not truly represent the SARA source model.

5.5.2 Lessons Learned

In above Section 5.5.1, we have discussed the *specific challenges* that we have faced in this chapter. In Section 2.6, *general challenges* of CBD are discussed and comparative analysis of the state of the art approaches are presented in Table 2.2 around four general requirements: traceability (R1), interoperability (R2), V&V (R3) and certification (R4).

In this section, we summarize in Table 5.1 the learning we got, the contributions and limitations of our SARA approach relative of these general requirements.

Categories	Sub-categories	Approaches	Discussion criteria			
			Traceability	Interoperability	V&V	Certification
General-Purpose	OOP-Based	EJB		(✓)	(✓)	
		Fractal		(✓)	(✓)	
	ADL-Based	AADL	(✓)	(✓)	✓	(✓)
		Pin		(✓)	✓	(✓)
Specialized-Purpose	OOP-Based	Think		(✓)	(✓)	
		CHESS		(✓)	(✓)	
	ADL-Based	ProCom		(✓)	(✓)	
		IEC-61499		(✓)	✓	(✓)
		SARA	✓	✓	✓	
		AUTOSAR	(✓)	✓	✓	(✓)

Table 5.1: Synthesis of comparison for some CBD-V approaches including ours

- R1. **Traceability.** The traceability support is generally weaker in academic component models or frameworks mostly due to extensive cost and time to deal with this requirement. Following some researches [Mader and Egyed, 2012, Dömges and Pohl, 1998] that show that system development practice with traceability can be cost benefit in long term if it is adapted to the project-specific needs, we have defined a lightweight traceability meta-model for domain component-based development. We have gathered the most relevant work related to traceability of concerns in MDE, and highlight their benefits for component-based model driving development.
- R2. **Interoperability.** The interoperability of components is a fundamental design desiderata of component-based development. As a consequence, with some exceptions, all of the CBD approaches provide a partial support interoperability. However, they do not focus on domain-specific knowledge, where a protocol of communication, is required for an effective interoperability. For this, in addition of two levels of interoperability, generally defined by common models *i.e.*, (i) *syntactic level* that covers the static aspects of component interoperation and (ii) *the semantic level* that covers the behavioral aspects of component interoperation, we have followed a theory of contract [He et al., 2006] and defined an additional level, *i.e.*, (iii) *the protocol level* to deal with the order in which a component expects its methods to be called.
- R3. **V&V.** are at the heart of the development process of system that require high dependability. As a consequence, most of component models dedicated to these systems offer a formal execution model. However, this does not imply that these formal models can be used without substantial additional effort to integrate them into a tool and a

development process. For this purpose, we have tried to use different several tools for verification tasks. However, an integrated tool-chain demonstrator is required for real-world development process.

- R4. **Certification.** As the traceability, certification support is generally weaker in academic component models or frameworks mostly due to extensive cost and time to deal with these requirements without industrial implication. CBSE is now an established software engineering research area. However, as it spread to safety-critical domains, it is crucial to establish to what extent individual components can be trusted and depended on, as well as developing a solid understanding of the impact from individual components on the overall dependability of the system for certification reason.

5.6 Summary

In this chapter, we have presented the formal aspects of our component-based modeling and observer-based verification. We have formalized our domain-specific SARA model and translated it into the TAO model for which we can use a time model checker tool for verification tasks. Common real-time extensions of Dwyer et al. patterns are intuitively represented as observers that react to the input and the output operations of the observed system model. This provides an intuitive way of expressing common real-time properties without requiring a significant knowledge of higher order logic and theorem proving. The correctness of a SARA model transformation into a TAO model is realized by applying the bisimulation relation based on transformation links. The correctness of our observer-based verification shows that any trace of the observed system is preserved in the composition of the system under observation and the observers.

In Chapter 6 of this dissertation, we evaluate the applicability of our approach through concrete case studies of railway safety-critical software.

Part IV

Validation

Validation Through Railway Safety-Critical Software

Contents

6.1 Introduction	118
6.2 Overview of SARA Process	118
6.3 A Brief Presentation of ERTMS/ETCS	120
6.3.1 A Structural presentation of ERTMS/ETCS	120
6.3.2 A Temporal QoS Ontology For ERTMS/ETCS	121
6.4 Rail-Road Level Crossing Case Study	122
6.4.1 The Case Study Motivation	122
6.4.2 Modeling, Verification and Tracing	124
6.5 RBC Handover Case Study	130
6.5.1 The Case Study Motivation	131
6.5.2 Modeling, Verification and Tracing	132
6.6 Metrics for Model Transformation and Component Reuse	137
6.6.1 Metrics for ATL and QVT Model Transformation	137
6.6.2 Metrics for Component Reuse Cost	138
6.7 Threats to Validity and Discussion	145
6.7.1 Scalability Concern	145
6.7.2 Safety Risk Assessment	147
6.8 Summary	147

6.1 Introduction

In this chapter, we evaluate the applicability of our approach, which is presented in contribution Part III, through some concrete case studies of railway safety-critical software. The case studies are derived from the System Requirement Specification (SRS) of ERTMS/ETCS. The aim of the case studies is to identify requirement concerns and architecture elements from which we can apply our traceability process shown in Figure 4.5 and our observer-based verification process shown in Figure 5.10.

The chapter is structured as follows. In Section 6.2, we give an overview of our approach. Section 6.3 briefly presents the ERTMS/ETCS. Section 6.4 and Section 6.5 describe the first and the second case studies, respectively. Section 6.6 shows some metrics related to experiments we conducted to evaluate our approach. Section 6.7 discusses the advantages and limitations of our approach. Finally, Section 6.8 summarizes and concludes the chapter.

6.2 Overview of SARA Process

As shown in Section 2.2, there are many component-based software development lifecycles. Most of them consider that the complete system should be completely constructed with component-based development. As a consequence, in the requirement analysis phase, the focus is put on component properties rather than user requirements. In this way, these lifecycles do not discussed in detail domain specific requirement issues.

However, as already discussed in Section 1.1, due to the railway control application market structure (many manufacturers and vendors), it is very difficult to build the complete system by only using these idealized CBD lifecycles. Indeed, there are always some requirements that are customer specific and vary from customer to customer, and still developed with traditional software development processes. One possible way to solve the user requirement problem is to develop reusable software components that meet the user requirements, and that can also be used in future projects. In this way, a hybrid development process is required to support both component-based software development as well as conventional software development process, such as V-Lifecycle.

Figure 6.1 shows an overview of our component-based model-driven development process named SARA development process, included in the V-Lifecycle prescribed by the CENELEC standard of railway safety-critical software for control and protection systems [EN-50128, 2011]. The entry point of the SARA development process is the Software Requirement Specification (SoRS), which provides the foundation for system comprehension, design, testing and maintenance. The SoRS is derived from the System Requirement Specification (SyRS). As already mentioned in Assumption 1, we assume that the SoRS is *traceable* if (i) the origin of each of its requirements is clearly related to the SyRS ones and if (ii) it facilitates the referencing of each requirement in future software development or enhancement artifacts.

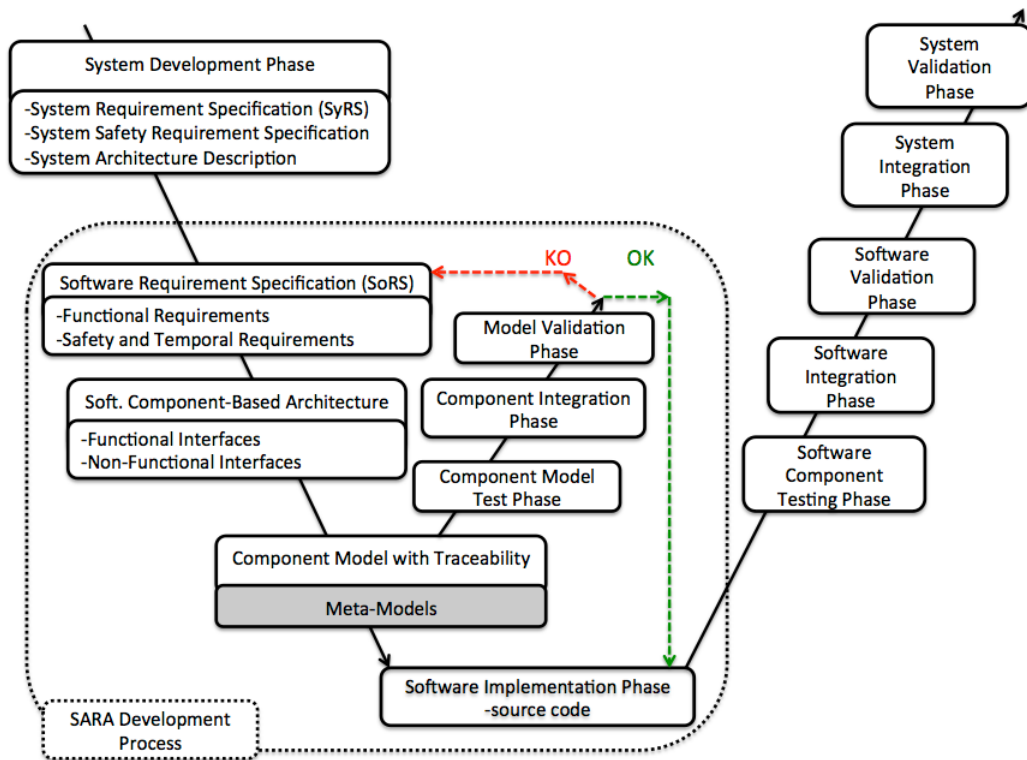


Figure 6.1: SARA component-based development and verification process included in CEN-ELEC prescribed V-Lifecycle

The key idea is that if scenario concerns are changed, model elements can be repeatedly and easily tracked in order to assess the change impact analysis whenever there is a change. As illustrated in the surrounded part of Figure 6.1, the SARA process includes two activities:

1. The software component-based design activities (the left branch of the surrounded part). From the phase of software requirement specifications, we want to separate functional requirements from safety and temporal requirements in software component-based architecture in order to facilitate the traceability of requirement concerns.
2. The verification activities performed at the model level (the right branch of the surrounded part). These verification activities performed at the model level are separated from other verification activities performed at the source code and the system level. They include both a test of individual components and their impact on the overall composite system. With these activities, we want to link back verification results ("ok" or "ko") to initial requirements or intermediate models to ensure that specifications are properly implemented.

The SARA process is based on core meta-models, which have been presented in Chapter 4. These meta-models are used for modeling software requirement concerns, component

elements and the traceability links among elements in different abstraction levels, from design level to implementation level and verification level. We have applied this process to evaluate our approach through some case studies of railway safety-critical software, derived from ERTMS/ETCS.

6.3 A Brief Presentation of ERTMS/ETCS

6.3.1 A Structural presentation of ERTMS/ETCS

The ERTMS is the European Rail Traffic Management System and the ETCS is the European Train Control Sub-system. Figure 6.2 shows the system architecture of ERTMS/ETCS and its interfaces with the Global System for Mobile communications - Railway (GSM-R) and other signaling systems, such as specific national systems, interlocking and centralized control center.

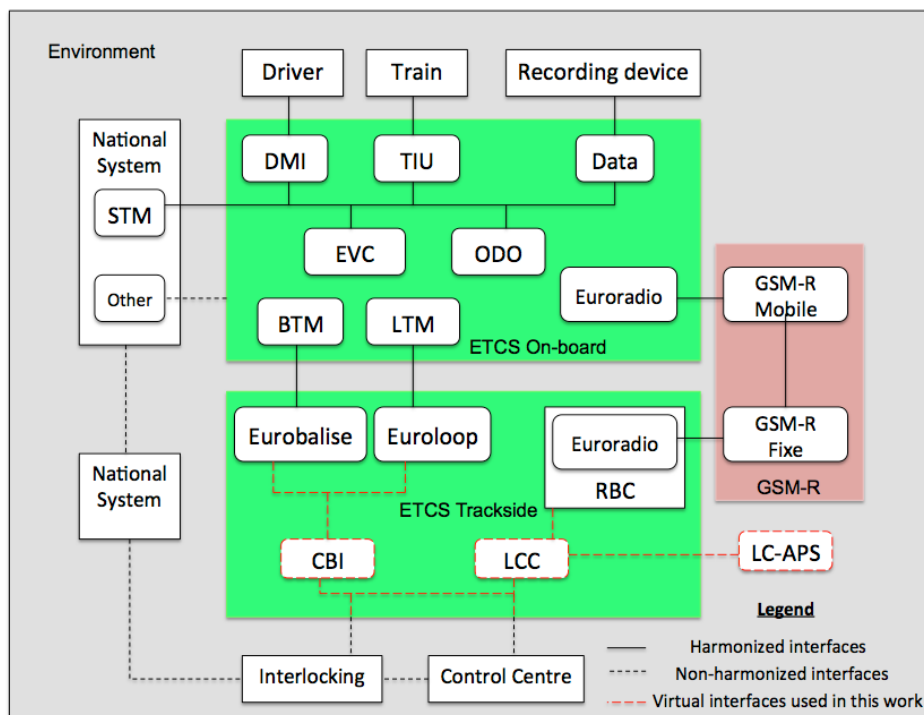


Figure 6.2: System architecture of the ERTMS/ETCS and its interfaces

The ETCS consists of two parts: the On-board part and the Trackside part. The detail of each part and components can be found in the specification [ERTMS/ETCS, 2014]. As shown in Figure 6.2, two types of interface specifications have been identified. The first category is the harmonized interfaces, such the interface between the Eurobalise and Eurobalise reader, which is called the Balise Transmission Module (BTM). This category ensures that

systems from different suppliers, which meet this specification, can work together without any further precautions. The second category is the non-harmonized interfaces, such as the interface between Radio Block Centre (RBC) and interlocking. This category will depend on the type of interlocking or national systems to be used on an ERTMS project or experimentation. For our experimentation, we add a virtual Computer-Based Interlocking (CBI), as shown in Figure 6.2, which communicates with interlocking, RBC, Eurobalise and Euroloop to provide the track occupancy information for the RBC handover case study. In addition, based on the new general architecture of rail-road Level Crossing Automatic Protection System (LC-APS) [Khouidour et al., 2009], we add a computer-based Local Control Centre (LCC) for the LC-APS case study.

6.3.2 A Temporal QoS Ontology For ERTMS/ETCS

Ontologies offer a means for representing and sharing information in many complex domains. For example, it can be used for representing and sharing information of System Requirement Specification (SRS) of complex systems like the SRS of ERTMS/ETCS written in natural language. Since this system is a real-time and critical system, generic ontologies, such as OWL [W3C, 2004] and generic ERTMS ontologies [Hoinaru et al., 2013, Hoinaru et al., 2014] provide minimal support for modeling temporal information omnipresent in these SRS documents.

To address this challenge, we have proposed a lightweight *3-layer* temporal Quality of Service (QoS) ontology [Sango et al., 2015a, Sango et al., 2015b] for representing, reasoning and querying over temporal and non-temporal information in a complex domain ontology. Figure 6.3 shows our 3-layer temporal QoS ontology for ERTMS.

The *upper layer* shows the generic ERTMS ontology that represents the main elements of ERTMS/ETCS SRS. The *intermediate layer* describes the generic QoS characteristics of these components. The *lower layer* represents the specific temporal QoS characteristics layer. The separation of the intermediate QoS layer from the lower QoS layer allows us to focus on specific QoS Characteristics, such as temporal or integrity characteristics. Here, we focus on temporal information that can be used to predict system run-time operation. In the generic ontology, the most significant component that can extend over time is the *Procedures* class. In the ERTMS/ETCS SRS documents, it is described by flowchart [ERTMS/ETCS, 2014], which is represented in our ontology by states, conditions and transitions. This class can be specialized in sub-classes, such as the Start of Mission, the Rail-Road level crossing, the RBC-RBC handover procedures and so on. To evaluate our approach presented in Chapter 4 and Chapter 5, the proposed domain ontology is instantiated and evaluated for the domain-specific Rail-Road level crossing and RBC-RBC handover case studies.

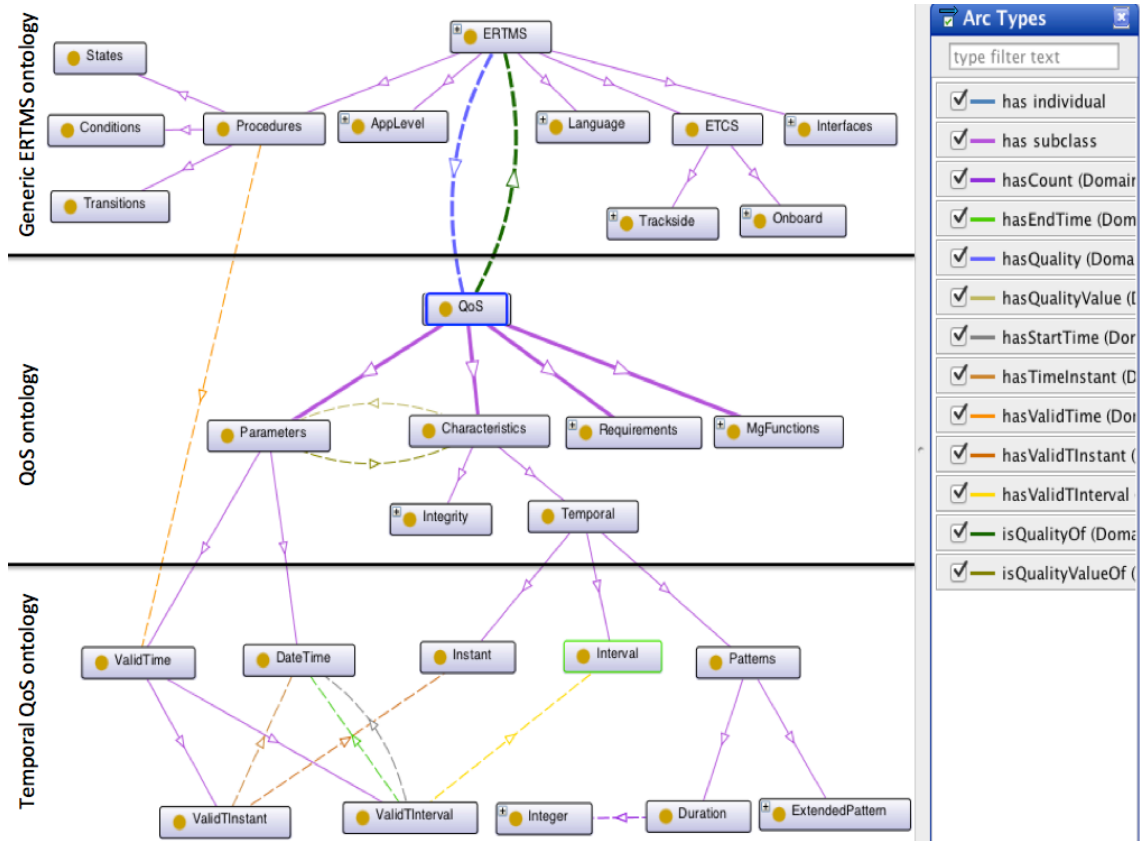


Figure 6.3: 3-Layer Temporal QoS Ontology for ERTMS/ETCS

6.4 Rail-Road Level Crossing Case Study

The aim of this case study description is to identify requirement concerns, from which we can evaluate our component-based modeling with traceability of concerns, presented in Chapter 4. We mainly identify requirement concerns, which could be changed, in order to investigate the impact of changes on software artifacts during the modeling, implementation and verification phases.

6.4.1 The Case Study Motivation

The safety of rail-road Level Crossing Automatic Protection System (LC-APS) has one of major concerns for railway and road stakeholders. Consequently, it has been used as a benchmark in several research vulgarization works, such as [El-Koursi et al., 2009, Khoudour et al., 2009, Bhatti et al., 2011, Mekki et al., 2012]. However, every year, more than 400 people continue to die in over 1,200 accidents involving road vehicles at rail-road LCs in the European Union [Eurostat, 2012]. Recently, the Coordination Action for the Sixth Framework Programme "Safer European Level Crossing Appraisal and Technology" (SELCAT) has

provided recommendations for further actions intended to improve safety at Level Crossings [El-Koursi et al., 2009]. However, there is no harmonized LC specification. In the recent version of ERTMS/ETCS specification [ERTMS/ETCS, 2014], a set of LC requirement concerns is specified. This specification is generic because it is not related to a specific LC topography or architecture view. To be more general and pragmatic, we use the architecture view of double-track railway lines proposed in [Mekki et al., 2012].

The Case study Architecture View

Figure 6.4 shows the LC topography considered. It is composed of the following features: (1) double-track railway lines (*UpLine* and *DownLine*); (2) roads with traffic in both directions; (3) traffic lights to manage the road traffic in the LC zone; (4) sound alarms to signal train arrival; (5) two half-barriers used to prevent road users from crossing while trains are passing; (6) three train sensors An_i , Ap_i and Ex_i in both track lines.

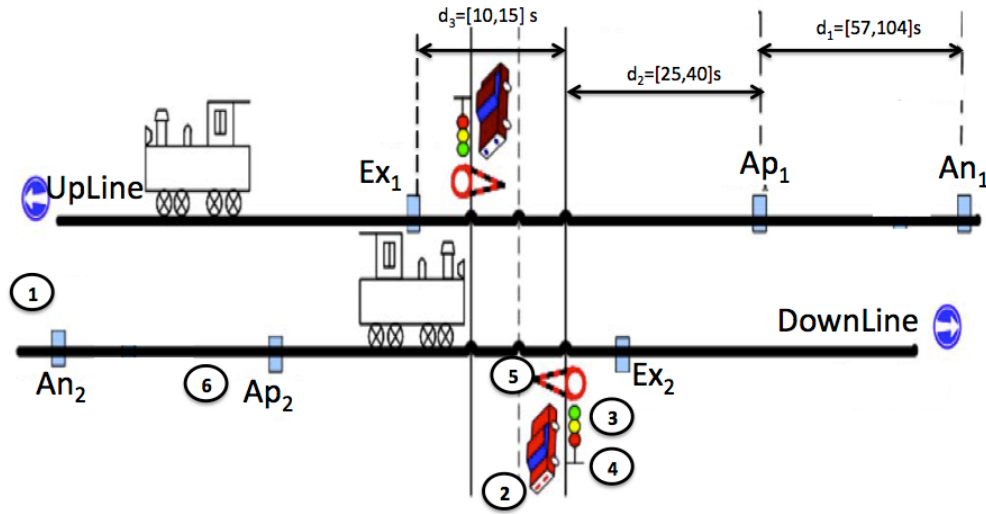


Figure 6.4: A level crossing topography

For example, in *DownLine*, the An_2 is the anticipation sensor, which allows the detection of the speed of an approaching train, necessary to alert road users with sound alarm and road lights. The approaching sensor Ap_2 is used to detect the arrival of trains in the LC zone and the exit sensor Ex_2 is used to announce the departure of trains after exiting the LC zone. Since several trains with different speeds (passenger or freight trains) can circulate on railway lines, the required durations between sensors are expressed with intervals, as shown in Figure 6.4. For example, $d_1 = [57, 104]$ second (s) is a required interval of durations between An_i and Ap_i . These interval requirements have to be respected by different trains circulating in these railway track lines.

Identification of Requirement Concerns

Here, we identify some requirements related to above LC-APS described architecture. We assume that requirement concerns, such as data and temporal parameters, can be changed for evolution reasons.

Functional requirement with a possible change of data format “Calculation of a dynamic speed profile taking into account the train running / braking characteristics which are known on-board and the track description data” (from paragraph §2.6.6.2.4 of [ERTMS/ETCS, 2014] and identified by FC_REQ_1).

Functional requirement with a possible change of MRSP “In the start location (D_{start}) of the LC, ETCS on-board system shall immediately include the LC speed restriction in the Most Restrictive Speed Restrictions (MRSP). The end of the LC speed restriction shall be the LC end location (D_{end})” (from §5.16.3 of [ERTMS/ETCS, 2014] and identified by FC_REQ_2).

Since the LC supervision involves train-passengers and road users safety, these functional requirements are enhanced with non-functional requirements in order to deal with the critical risky situations. In the following, we consider two non-functional requirements related to time constraints.

Non-Functional requirement with a possible change of minimum desired period of time “when the gate is open to road traffic, it must stay open at least T_{min} time units, where T_{min} represents the minimum desired period of time separating two successive closing cycles of gate” (adapted from [Mekki et al., 2012] and identified by NFC_REQ_1).

Non-Functional requirement with a possible change of time interval constraints “once closed and when there is no train approaching, the gate must be kept closed at least (T_{begin}) and at most (T_{end}), where T_{begin} and T_{end} are the time limits prescribed” (adapted from [Mekki et al., 2012] and identified by NFC_REQ_2).

Our objective is to trace these requirement concerns within or across these different abstraction levels. We focus on three main levels: the model design level, the model analysis level and the model implementation level.

6.4.2 Modeling, Verification and Tracing

► Modeling Phase:

As presented in the proposed process to apply the defined meta-models (see Figure 4.5), the modeling phase consists in modeling requirement concerns and architecture components. Listing 6.1 shows the model of requirement concerns and architecture components

identified in the previous section. It is a valid XML format in accordance with the concern meta-model DTD shown in Listing 4.9. The example shows only one functional requirement concern (id=FC_REQ_1, see line 5 of Listing 6.1) and one non-functional requirement concern (id=NFC_REQ_1, see line 8 of Listing 6.1).

```

1<?xml version="1.0" encoding="UTF-8"?>
2<!DOCTYPE ConcernModel PUBLIC "ConcernModel" "concern-model.dtd">
3<ConcernModel>
4  <ConcernGroup name="Rail-road Level Crossing (LC) application requirements">
5    <ReqConcern id="FC_REQ_1" name="LC-sensing-calculating-actuating" type="
      FR">
6      <Description></Description>
7    </ReqConcern>
8    <ReqConcern id="NFC_REQ_1" name="LC-time-controlling" type="NFR">
9      <Description></Description>
10   </ReqConcern>
11 </ConcernGroup>
12 <TraceableEltBetweenView />
13 <ComponentModel name="Rail-road Level Crossing (LC) application component">
14   <CompUnit id="COMP_UNIT_1" ref="LC-model-views.xml" type="component-model
      "></CompUnit>
15 </ComponentModel>
16</ConcernModel>

```

Listing 6.1: The LC-APS use case's concern modeling

By always keeping in mind the separation of concerns, we use a separated file `LC-model-views.xml` referenced in line 14 of Listing 6.1 to specify the component model of LC-APS component unit identified by `COMP_UNIT_1`.

The `LC-model-views.xml` file is shown in Listing 6.2, which represents the model abstraction levels of our LC-APS use case. We only focus on three model abstraction levels in Listing 6.2: SARA component modeling view for the model design level, its UPPAAL analysis view for the model analysis level and its ADA implementation view for the model implementation level. These three views are referenced respectively by identifier `Sara_View_ID`, `Uppaal_View_ID`, `Ada_View_ID`, shown respectively in the line 4, 5 and 6 of Listing 6.2. Each of them are separately defined in a file, as described bellow for SARA model.

LC-APS Modeling with SARA Model

The SARA model of LC-APS is the high level abstraction view among the three abstraction views of LC-APS, as referenced in line 4 of Listing 6.2.

Listing 6.3 shows the XML implementation of this component-based modeling. It is a valid XML format in accordance with the component-model DTD shown in Listing 4.8. In Listing 6.3, we only focus on one connection link, the `DC_1` links between *Sensor* component

```

1<?xml version="1.0" encoding="UTF-8"?>
2<!DOCTYPE ModelAbstractionLevel PUBLIC "ModelAbstractionLevel" "component-model.
  dtd">
3<ModelAbstractionLevel id="LC_USE_1" ref="" name="LC use case model">
4  <CompView id="Sara_View_ID" ref="LC-Sara-model-view.xml" type="model design
    view"></CompView>
5  <CompView id="Uppaal_View_ID" ref="LC-Uppaal-verification-view.xml" type="
    model analysis view"></CompView>
6  <CompView id="Ada_View_ID" ref="LC-Ada-implementation-view.xml" type="model
    implementation view"></CompView>
7</ModelAbstractionLevel>

```

Listing 6.2: The LC-APS use case's 3-layer abstraction levels: SARA component modeling view for the model design level, its UPPAAL analysis view for the model analysis level and its ADA implementation view for the model implementation level

and *Controller* as shown in Figure 5.1. The XML type attribute of this connection is identified by *DC_1* in line 16 of Listing 6.3. It includes the *in* and *out* attributes to denote the direction of variables that are exchanged. The other connection types are described in the same ways.

► Verification Phase:

The verification phase consists in translating the SARA design model into TAIIO models (see Section 5.3). The pattern of specification to be verified is also translated into observer TAIIO model (see Figure 5.10). Then, we use UPPAAL model checker for verification tasks.

Transformation into TAIIO Model

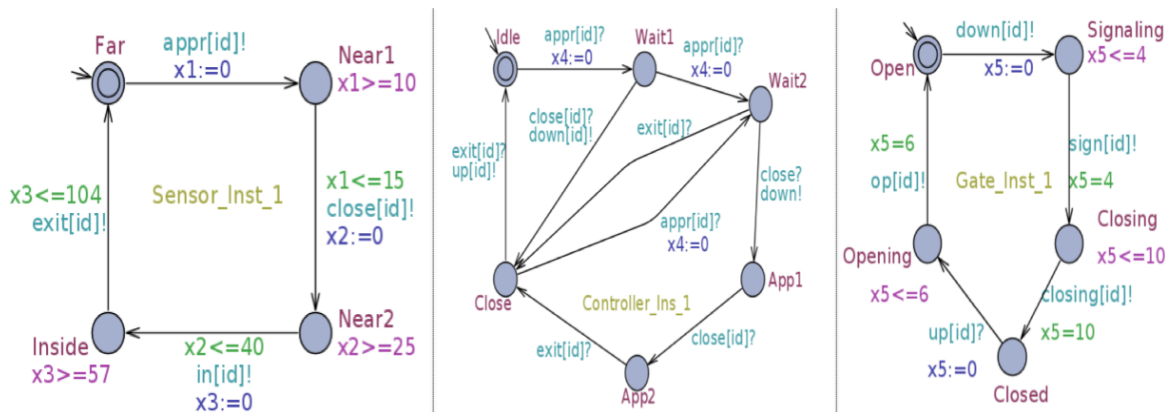


Figure 6.5: UPPAAL TAIIO model of LC-APS components


```

1<?xml version="1.0" encoding="UTF-8"?>
2<!DOCTYPE CompView PUBLIC "ModelAbstractionLevel" "component-model.dtd">
3<CompView id="Sara_View_ID" name="LC-ccv" type="component and port view">
4  <CompElement id="Sara_CP_ID" name="LC-cce" type="component and port element">
5    <CompEntity id="Sensor_Inst_1" name="sensor" type="component">
6      <CompEntity id="Sensor_Inst_1.op1" name="sensor-port" type="port"></
        CompEntity>
7      <CompConnection id="Sensor_Inst_1.LC_1" name="set-sensing" type="
        local connection">
8        <Parameter id="appr" name="out-var_1" type="output parameter
        variable">
9          <out>
10            <CompEntity id="Sensor_Inst_1.op1.appr" name="out-param_1
              " type="output parameter"></CompEntity>
11          </out>
12        </Parameter>
13      </CompConnection>
14    </CompEntity>
15    <CompConnection id="DC_1" name="train-approaching" type="direct connection">
16      <Parameter id="appr" name="var_1" type="transit parameter">
17        <out>
18          <CompEntity id="Sensor_Inst_1.op1.appr" name="out-param_1"
              type="output parameter"></CompEntity>
19        </out>
20        <in>
21          <CompEntity id="Controller_Inst_1.ip1.appr" name="in-param_1"
              type="input parameter"></CompEntity>
22        </in>
23      </Parameter>
24    </CompConnection>
25    ...
26  </CompElement>
27</CompView>
28</CompView>

```

Listing 6.3: The XML implementation of SARA model for the LC-APS

Based on our transformation approach presented in Section 5.3.3, the SARA model of LC-APS is translated into the TAO model. Figure 6.5 shows the UPPAAL TAO model generated for LC-APS.

Verifications tasks

The generated TAO model, is used for verification task with the UPPAAL tool. Here, we focus on the linking of verification results to TAO target model elements. We also focus on the manual linking of requirements to the SARA source elements. Then, based on Listing 4.7 that builds the traceability matrix, the SARA source elements is automatically link to the TAO target elements in the traceability matrix. Table 6.1 shows an example of a traceability

matrix for our use case. Before the requirement validation, each line contains requirement concern *ID* and user parameters (e.g., upper or lower limit). After performing the verification, the verification results with requirement validation status passed, failed, unchecked are manually inserted for each requirement.

<>	A	B	C	D	E	F	G	H	I	J
1	Requirement and Design Concerns					Verification and Validation Concerns				
2	Req ID	Red DES	Lower Limit	Upper Limit	SARA Elts	TAIO Elts	Valid Status	Verif Type	Detect Min	Detect Max
3	FC_-REQ_1	See Sec. 6.4.1			#S#C#G #opC()#C #opS()#S #opG()#G #null#	#S#C#G #opC()#C #opC()#C #null# #opG()#G	Unchecked			
4	FC_-REQ_2	See Sec. 6.4.1	Dstart = 5 km	Dend= 7.5 km MRSP= 70km/h	#S#C#G #opS()#S #opC()#C #opG()#G #mrsp#	#S#C#G #opS()#S #opC()#C #opG()#G #mrsp#	Passed	by simulation	32 km/h	34 km/h
5	NFC_-REQ_1	See Sec. 6.4.1	Tmin= 70 s		#S#C#G #opS()#S #opC()#C #opG()#G #open#G#	#S#C#G #opS()#S #opC()#C #opG()#G #open#delay#G#	Failed	by UP-PAAL	66 s	
6	NFC_-REQ_1	See Sec. 6.4.1	Tmin= 57 s		#S#C#G #opS()#S #opC()#C #opG()#G #open#G#	#S#C#G #opS()#S #opC()#C #opG()#G #open#delay#G#	Passed	by UP-PAAL	66 s	
7	NFC_-REQ_2	See Sec. 6.4.1	Tbegin= 29 s	Tend= 55 s	#S#C#G #opS()#S #opC()#C #close#G#	#S#C#G #opS()#S #opC()#C #close#G#	Passed	by UP-PAAL	32 s	32 s

Table 6.1: Validation results of requirements identified in Section 6.4.1

The mapping between the SARA source model elements and TAIO target model elements is automatically realized by Listing 4.7. If information is unchanged during the transformation of the source elements into the target elements, the SARA source model and the TAIO target model contain the same information. For instance in line 3 and column E and F of Listing 4.7, the mark #S#C#G in both columns mean that the source and the target models contain the same Sensor (S), the Controller (C) and the Gate (G) elements. If there are elements in the source model without corresponding ones in the target model (marked as #null#), then we can deduce that the element has been removed or renamed by the transformation. If there are elements in the target model without corresponding ones in the source model (marked as #null#), this implies the target element has been created or renamed by the transformation. If the relation between the source and target elements differs (e.g., #open#G and #open#delay#G), then the relationship should be re-tested. In this case, the designer have to make a decision. For example, in our use case, we modified the time parameters

(T_{min}) of requirement NFC_REQ_1 during subsequent experiments and the failure was no longer occurred, as illustrated in line 6 of Table 6.1.

► Tracing Phase:

As presented in the proposed process to apply the defined meta-models shown in Figure 4.5, the tracing phase consists in tracking concerns within or across model abstraction levels. Indeed, once the models and the mappings are defined, as illustrated in modeling phase, we can trace concerns within or across model abstraction levels. For this purpose, we can reuse either predefined reusable XQuery queries presented in Section 4.4.4, or write customized queries for tracing concerns.

For example, to identify the component elements that are impacted when the requirement concerns identified by NFC_REQ_1 is changed, we use the predefined function *ForwardTraceQuery*("*", NFC_REQ_1). This function is defined in line 24-30 of Figure 6.6 and the corresponding query statement is shown in lines 32-34.

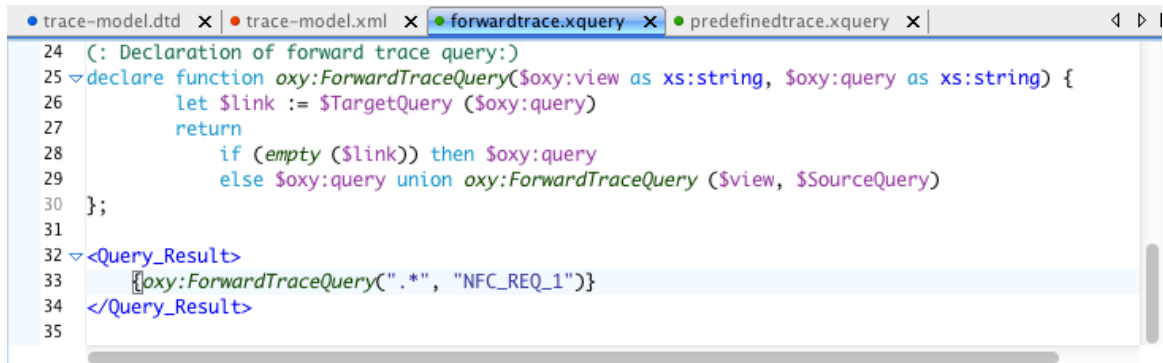
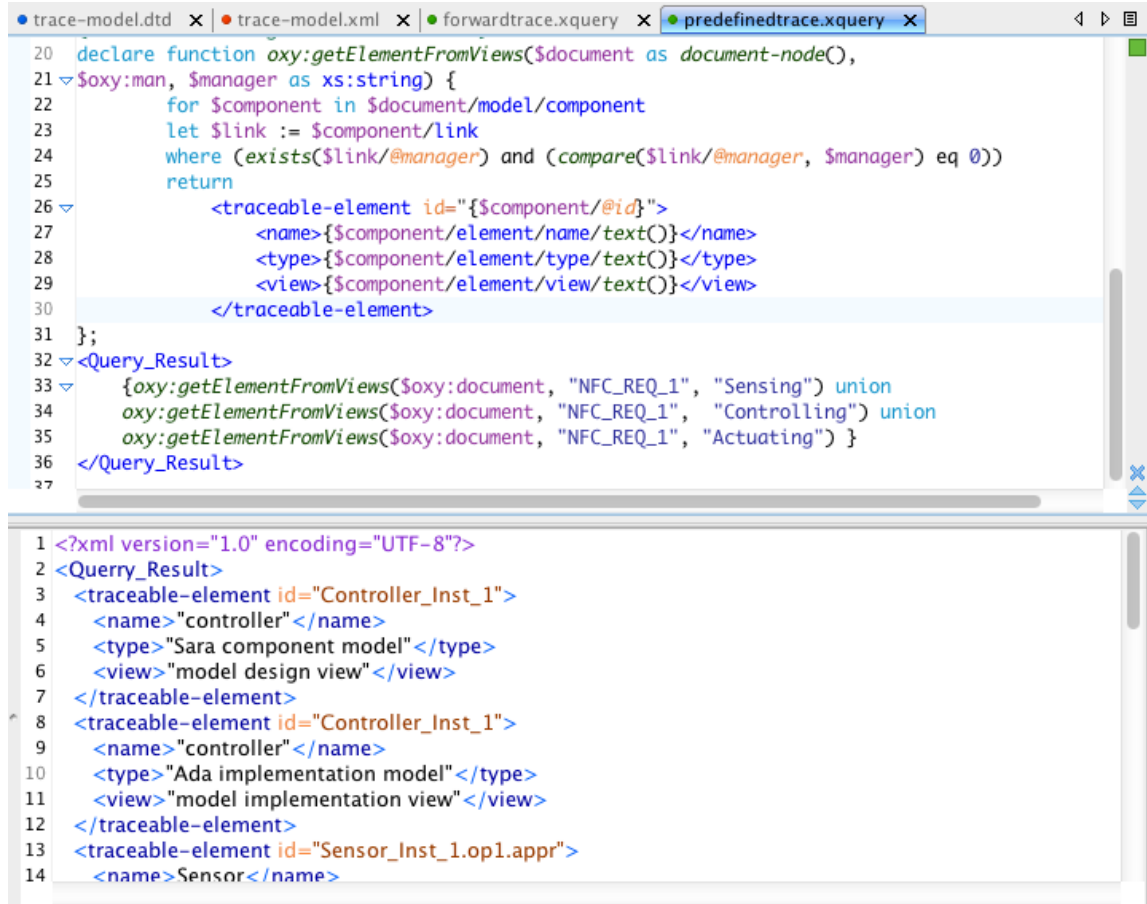


Figure 6.6: The forward trace query definition

However, it is not always possible to directly trace the concerns by providing generic concern like the LC requirement identified by NFC_REQ_1. In this situation, we apply the domain knowledge to characterize the requirement concerns. For example, for *ForwardTraceQuery*("*", NFC_REQ_1) query, we need to identify all the concerns that use sensing, controlling and actuating concerns of this LC requirement. Accordingly, the union set of the predefined queries (here *oxy:getElementFromViews*) is defined, as illustrated in line 33-35 of the top view of Figure 6.7. This union of queries determines elements in intra or inter views. The only difference between intra and inter view is that we have to refer to more than one view in the queries for inter view.

The result of the query is also an XML file, as shown in the bottom view of Figure 6.7. Of course the XML file can be transformed for human best visualization. We are mainly concentrated on providing minimal support in XML-based representations and XQuery queries to provide a quick feedback about the current research state. An effective tool support for a real development environment and graphical visualization is one of our perspectives.



```

20 declare function oxy:getElementFromViews($document as document-node(),
21   $oxy:man, $manager as xs:string) {
22   for $component in $document/model/component
23   let $link := $component/link
24   where (exists($link/@manager) and (compare($link/@manager, $manager) eq 0))
25   return
26   <traceable-element id="{ $component/@id }">
27     <name>{ $component/element/name/text() }</name>
28     <type>{ $component/element/type/text() }</type>
29     <view>{ $component/element/view/text() }</view>
30   </traceable-element>
31 };
32 <Query_Result>
33 {oxy:getElementFromViews($oxy:document, "NFC_REQ_1", "Sensing") union
34  oxy:getElementFromViews($oxy:document, "NFC_REQ_1", "Controlling") union
35  oxy:getElementFromViews($oxy:document, "NFC_REQ_1", "Actuating") }
36 </Query_Result>
37
1 <?xml version="1.0" encoding="UTF-8"?>
2 <Query_Result>
3   <traceable-element id="Controller_Inst_1">
4     <name>"controller"</name>
5     <type>"Sara component model"</type>
6     <view>"model design view"</view>
7   </traceable-element>
8   <traceable-element id="Controller_Inst_1">
9     <name>"controller"</name>
10    <type>"Ada implementation model"</type>
11    <view>"model implementation view"</view>
12  </traceable-element>
13  <traceable-element id="Sensor_Inst_1.op1.appr">
14    <name>Sensor</name>

```

Figure 6.7: A result of trace query

Summary

With the LC-APS case study, we have evaluated our component-based modeling with traceability of concerns. We have shown that by using the meta-model with traceability of concerns, we are able to model, implement and verify a concrete safety-critical scenario, in such a way as to facilitate a change impact analysis by using a query mechanism.

6.5 RBC Handover Case Study

In addition to the evaluation of the LC-APS case study presented in Section 6.4, with the second case study, we want to show that our approach is applicable in several case studies. Here, we focus on the evaluation of observer-based verification presented in Chapter 5.

6.5.1 The Case Study Motivation

The RBC (Radio Block Center) handover is an important part of modern train control systems, such as the European Train Control System (ETCS) [Liu et al., 2011] and the Chinese Train Control System (CTCS) [Yang et al., 2014]. The RBC handover procedure is called in order to transfer the communication from the Handing Over RBC (HO RBC for short) to an Accepting RBC (AC RBC for short) and to avoid communication termination when the train gets outside the range of the HO RBC, as illustrated in Figure 6.8.

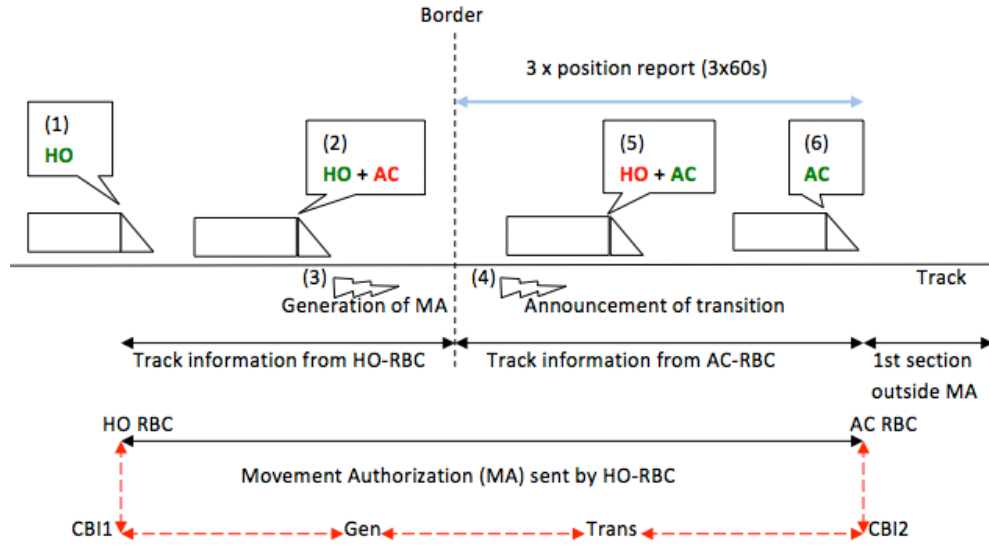


Figure 6.8: A RBC-RBC handover topography

In the specification, which can be found in paragraph §5.15 of [ERTMS/ETCS, 2014], there are two kinds of communication methods when the handover is announced: (i) two communication sessions can be handled simultaneously between the train and both the HO RBC and AC RBC, and (ii) only one communication session can be handled between the train and the HO RBC or the AC RBC, exclusively. Here, we consider the second method. Figure 6.8 shows the main functional steps needed to run from one RBC area to another one: (1) pre-announcement of the transition by the HO RBC; (2) HO RBC requests track information from AC RBC; (3) generation of Movement Authorization (MA) including the border information and track information from both HO RBC and AC RBC; (4) announcement of the transition; (5) transfer of train supervision to the AC RBC; and (6) termination of the session with HO RBC when HO RBC receives position report from on-board train.

Focus on non-deterministic time delay requirements

We focus on non-deterministic time delay performance. For this, we report in Figure 6.8 the delay scheme to deliver the track occupancy information. In the view of HO RBC, the mini-

minimum delivering time delay is $d_{min} = AC - CBI2 - Trans - Gen - CBI1 - HO$, see the dashed arrows in Figure 6.8. The maximum delivering time delay is $d_{max} = d_{min} + 3 * d_{position_report}$. In fact, "in case a communication session is established and no request to terminate the session is received from the HO RBC within a fixed waiting time after sending the position report, the position report shall be repeated with the fixed waiting time after each repetition" (§5.15.4 [ERTMS/ETCS, 2014]). For our evaluation, we allow 3 possible repetitions of position report and each position report takes 60 s, as illustrated in Figure 6.8. This non-deterministic time delay from d_{min} to d_{max} is what we focused on, and we want to check whether the response of the designed system can meet the time limit of the handover process at any time. We consider the following handover scenario requirements.

- R1: If an RBC receives track occupancy information within the boundary of the current Movement Authority (MA), it must send to the on-board system a Conditional EMergency stop (CEM) message immediately, and it does not need acknowledge;
- R2: If an RBC receives track occupancy information of the first section outside the boundary of the MA, the RBC determines within a certain time delay d_{wait} if the train (EVC) is a local vehicle or not. If not, RBC will send Shorten MA (SMA) message. This message needs to be answered by train in the delay d_{ack} ;
- R3: If an RBC receives a track occupancy information of another section (except the first section) outside the boundary of the MA, the RBC will send SMA to the train and this message needs to be answered in d_{ack} ;
- R4: In addition, if an RBC has not received the SMA response message ACK, but receives another track occupancy state information, it will no longer send CEM order to the train, but it will send an upgrade order Unconditional EMergency stop (UEM).

6.5.2 Modeling, Verification and Tracing

► Modeling Phase:

Based on the general ERTMS architecture, shown in Figure 6.2, including the RBC component, and the RBC handover scenario specified in above Section 6.5.1, the SARA model of this scenario contains three main components: CBI, EVC and RBC. As shown in Definition 4, the observable messages of each component is partitioned into periodic/aperiodic and input/output messages:

- RBC(periodic_message{train_pos?, req_MA?, MA!, Ack?},
aperiodic_message{TOcc?, SMA!, CEM!, UEM!, Ack?})
- EVC(periodic_message{pos_report!, Req_MA!, Ack!}
aperiodic_message{Ack!})

- CBI(aperiodic_message{TOcc!,T1Occ!})

Figure 6.9 shows the interaction between components through these messages. It is in accordance with handover scenario requirements (R1, R2 and R3) shown in Section 6.5.1.

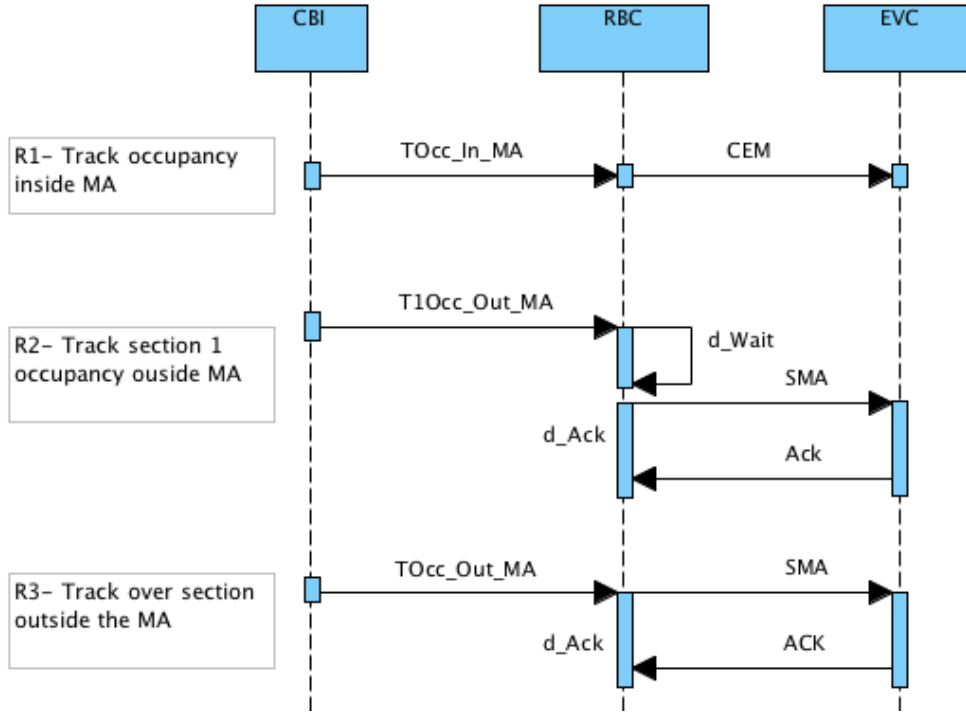


Figure 6.9: RBC-RBC handover scenario

► Verification Phase:

Transformation into TAIO Model

Based on the transformation process of Section 5.3.3, the TAIO models are generated from the SARA models. Figure 6.10 shows the generated TAIO models for RBC handover. Here, we consider the RBC component as the component under observation and the CBI and train EVC components as its environment part. As a consequence, in the following, we will focus on observer-based verification of the RBC component.

Verifications tasks

Based on the observer-based verification process (see Figure 5.10), the generated RBC TAIO model of Figure 6.10 is composed with the observer of requirement to be verified. Here,

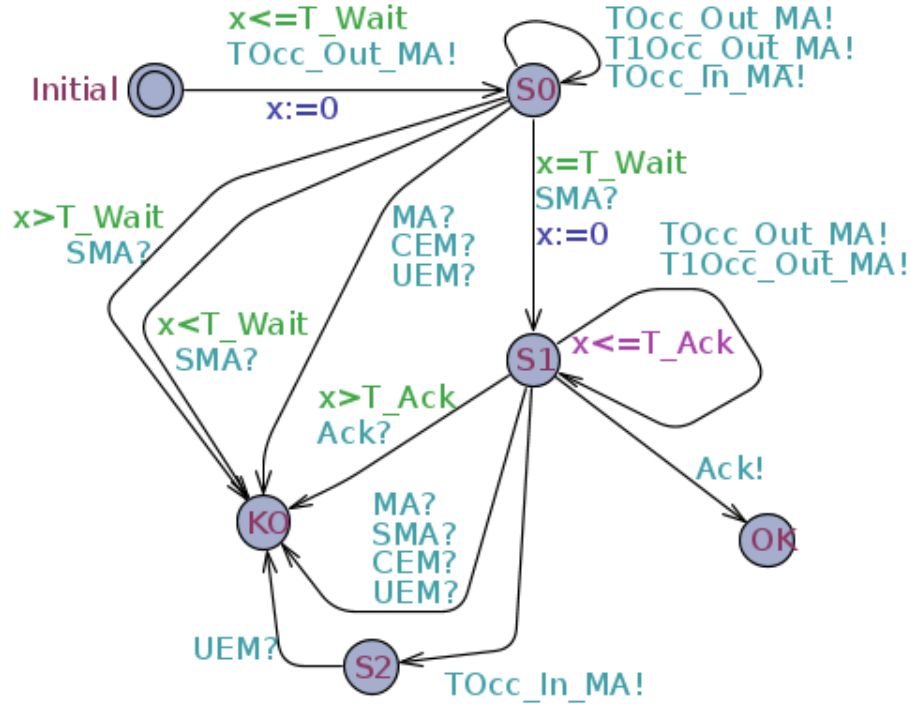


Figure 6.11: Composition of RBC TAIO model with response delay observer

ence of trans-boundary interlocking message delay. The verification fails for the description shown in Table 6.2. This means that with input action during the time window (148;171), the expected state is OK with the output action ACK, but we get a KO state with unexpected output UEM for Unconditional EMergency stop.

Table 6.2: Unacceptable output observed

RBC final state: IDLE.
Observable actions:
input : T1Occ_Out_MA@(148;171)
output : ACK
Internal actions annotations:
PDelay_Of_TOcc@(145;148)
QDelay_Of_TOcc@(146;150)
TDelay_Of_TOcc@(165;169)
PPDelay_Of_TOcc@(166;172)
Delay requirement: T_Delay_Max@200
Expected output: ACK@(165;172)
Got unacceptable output: UEM@(145;146)

Analysis of this result

The causes of failure may be various in different aspects. For example, in the logical function aspect, one of the components can receive exception message or an empty message. In

the real-time aspect, one of the components, for example the interlocking can transfer track occupancy information with a delay so long that it exceeds the time limit allowed in the specification. Based on the analysis of the last good states and the message delivering time sequence set before the failure, we conclude in Figure 6.11 the following failure path :

T1Occ_Out_MA! T_Wait.SMA? T_Ack.TOcc_In_MA! UEM?

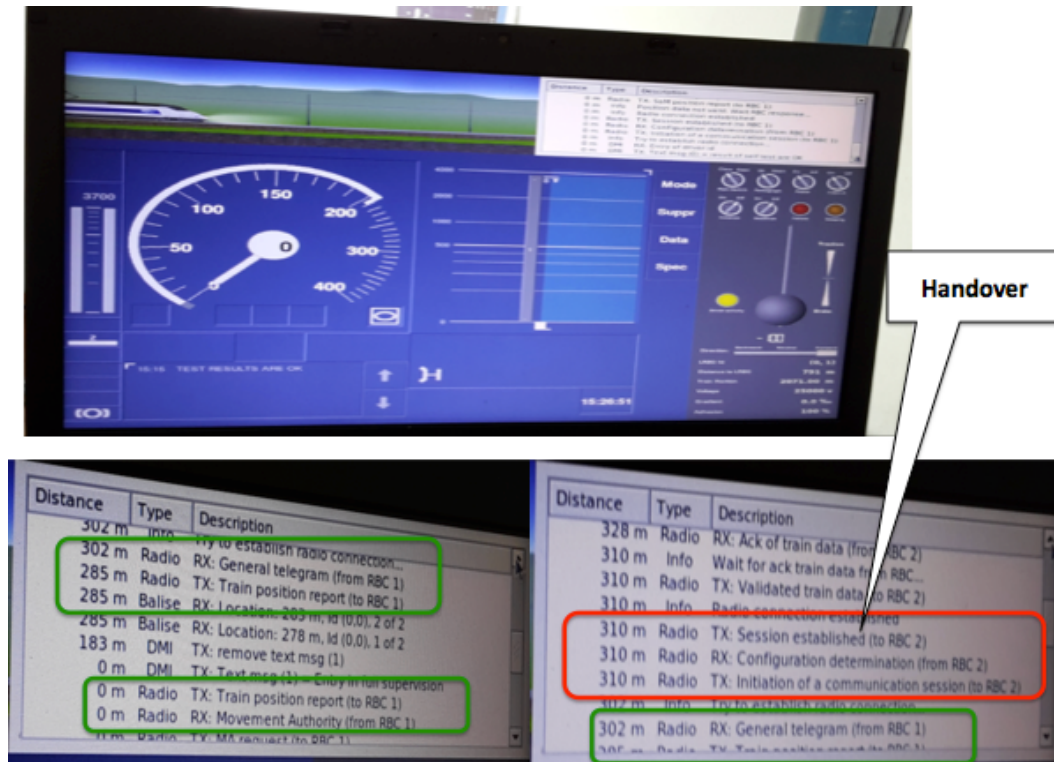


Figure 6.12: RBC handover simulation with ERSA Simulator

This failure path illustrated that the RBC receives a first track occupancy information of the first section outside the current MA, and sends an SMA order to the on-board equipment and needs to wait for an answer to this SMA order from the on-board equipment. However, the RBC received another second track occupancy information during this waiting time, thus leading the system logic to run into an exception state and to send out a UEM order.

The cause could be that the interlocking device delivered the first track occupancy information a little late. This could lead to the RBC for sending with a delay SMA order to on-board equipment and need to wait for an answer to this SMA order from on-board equipment. However, RBC received early another track occupancy information during this waiting time. Thus the system logic ran into exception state and sent out undesirable UEM order. This experiment showed that the time delay parameters in our initial RBC handover scenario specification were inconsistent with actual designed equipment.

In the subsequent experiments, we modified the time parameters in the requirement specification by widening the upper and lower limit value of the delay, and the failure no longer occurred. With the widening of the corresponding time interval, we limit the number of trains that can travel on the track section considered. But, the most important, here, is that it guarantees the robustness of the designed model. Indeed, as illustrated in Figure 6.12, to validate this robustness, we replayed the same scenario with the same time parameters on the ERTMS/ETCS Traffic Simulator developed by [ERSA, nd]. This Simulator is a real-time simulation demonstrating how trains can be run on tracks under ERTMS/ETCS supervision. The demonstration of our designed scenario with this real-time simulation confirms the robustness and is very encouraging for automating our software component-based modeling and observer-based verification approach.

Summary

With the RBC handover case study, we have evaluated our observer-based verification (OBV) with patterns of properties. We have shown that the OBV is robust by replaying with real-time ERTMS/ETCS simulator the same scenario validated with the OBV.

6.6 Metrics for Model Transformation and Component Reuse

Software metrics have been extensively studied in the last decades [Fenton and Pfleeger, 1998]. Metrics can be used to get quick insights or performance into the specific characteristics of a software artifact, among others. For example, in Section 6.6.1, we use metrics to get performance insights into our transformation rules implemented in two transformation languages. On the other hand, in Section 6.6.2, we also use metrics to analyze the reuse cost benefits of our component-based approach compared to an ad-hoc reuse strategy.

6.6.1 Metrics for ATL and QVT Model Transformation

Model transformations are increasingly being integrated in software development processes, as realized in our approach. However, when systems grow in size and complexity, the performance of the transformations tends to degrade. In this section, we investigate the factors that have an impact on our transformation rules defined in Section 5.3.3.

For this we have implemented our transformation rules with two languages: the ATLAS Transformation Language (ATL) [Jouault et al., 2006] and Query/View/Transformations Operational Mappings (QVT) [OMG, 2011]. The objective is to study the effect of different language constructs on the performance of the transformation. The two transformations are executed on the same platform: the Java Virtual Machine version 1.7.0_51, running on

Linux 64-bit version. The metrics have been extracted by using the metrics collection tool described in [van Amstel et al., 2011].

Figure 6.13 and Figure 6.14 show the time results of executing the transformation of LC-APS SARA model into TAIIO model. The results on Figure 6.13 show the effects of increasing the size of the input model, while the results on Figure 6.14 show the effect of increasing the complexity of the input model. Our assumption is that a model is more complex if it has more attributes, because attributes of components making the model elements more interconnected.

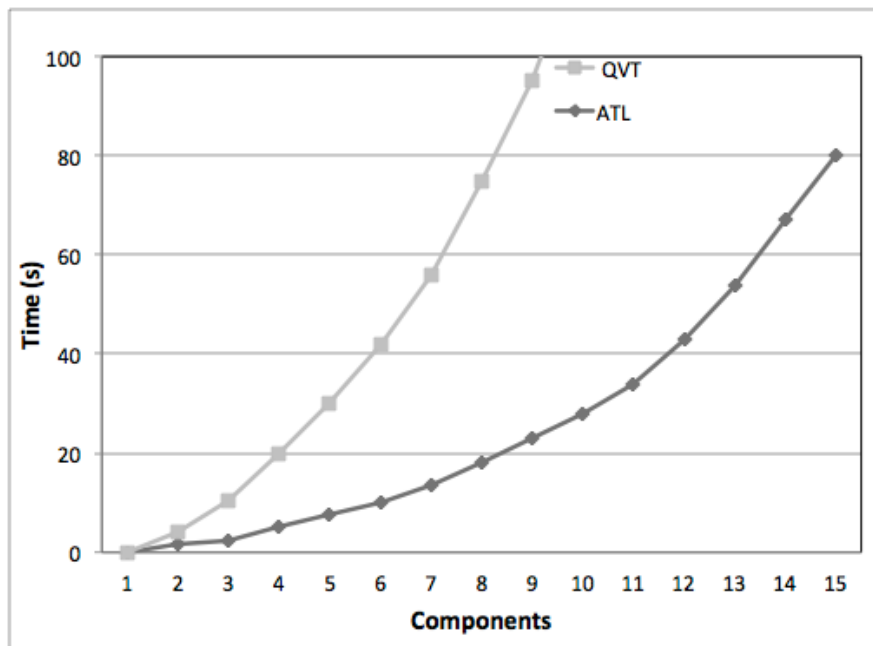


Figure 6.13: Effect of increasing the size of input model in model transformation

The comparison of ATL and QVT in both case, show that the ATL tool implementation is the fastest than the QVT implementation. However, when altering the size of the input models, the relative difference in performance of the transformations remains the same in Figure 6.13 and Figure 6.14

This shows that the complexity of the transformation is proportional to the size of and the complexity of the input model, and to the transformation engine. As a consequence, metrics can be used by the developers of transformation rules when estimating the expected performance of a transformation.

6.6.2 Metrics for Component Reuse Cost

The important point of component based software development is the reuse of existing software components. These reusable software components can arise from a variety of goals,

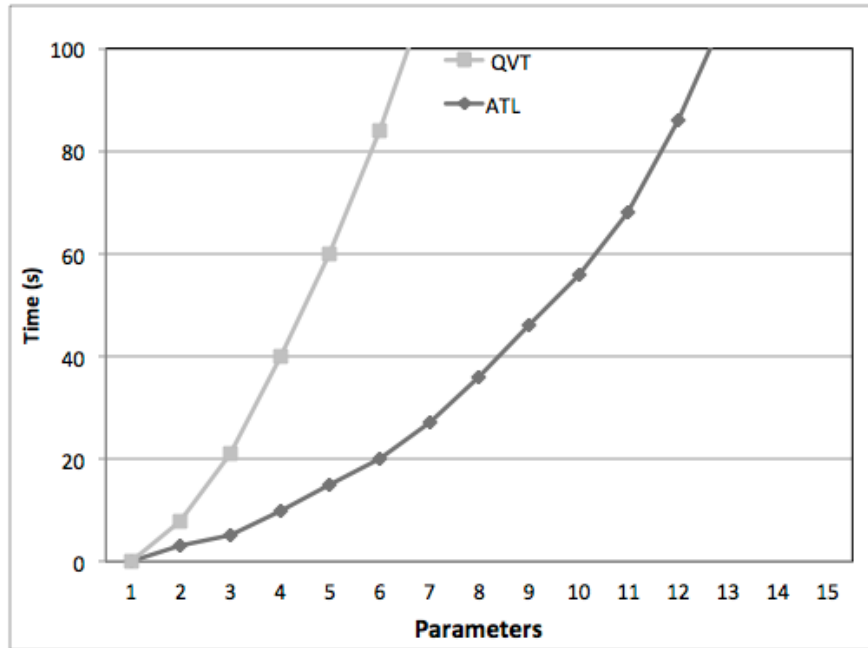


Figure 6.14: Effect of increasing the complexity of input model in model transformation

such as (i) Commercial Off The Shelf COTS, *i.e.*, components available in market just purchase them from third party, or (ii) identify the reusable software components from existing system [Gill, 2003]. Its main focus is to achieve the considerable software reuse and instead of building systems in-house from the scratch. For our best knowledge, there are not enough software COTS components available in the market that provide train control management functionality. So two options are available: whether components should be developed from the scratch or identify the reusable software components from existing systems. Identifying and modifying reusable software component from existing systems seems to be one of the more promising ways to obtain reusable assets.

As a consequence, we compare two software reuse strategy : (1) *ad-hoc reuse strategy* and (2) *systematic reuse strategy* [Rothenberger and Nazareth, 2002]. One approach of ad-hoc reuse strategy is a *Copy and Paste method (C&P)* used in the railway industries, such as Bombardier Transportation, as discussed in [Riaz, 2012]. One well-known approach of systematic reuse strategy is a *Component-Based Development (CBD)*. The CBD is a systematic way to achieve software reuse, as discussed in Section 2.2.

Reuse Cost Estimation for C&P approach

To analyze the cost benefits in C&P approach, we use Equation 6.1 derived from [Gill, 2003, Jasmine and Vasantha, 2008, Riaz, 2012]

$$\text{C\&P-COSTSAVING} = FS_{cost} - (S_{cost} + A_{cost} + D_{cost} + Q_{cost}) \quad (6.1)$$

- FS_{cost} is the cost of component when it will be developed from scratch without keeping reuse in mind.
- S_{cost} is the search cost that will have to perform for searching reusable component from old system. Search cost always will incur even in C&P because first similar component or code will have to search from existing system, and after searching it will be copy and paste.
- A_{cost} is the adaptation cost or modification cost because in real environment there must be some adaptation cost of component even though it is similar. There must be some changes and modifications that have to be performed in order to match the new requirements.
- D_{cost} is the documentation cost because new documentation will be required. Documents can be requirement specification, design specification, test specification, etc.
- Q_{cost} is the quality cost because V&V activities have to perform from scratch so there must also be quality cost in the case of copy paste technique. This quality include also testing cost because copy paste code must be retest again with new component. So testing will have to perform from scratch in the case of copy paste.

Reuse Cost Estimation for CBD approach

To analyze the cost benefits in CBD approach we use Equation 6.2.

$$\text{CBD-COSTSAVING} = FS_{cost} - (S_{cost} + R_{cost} + GD_{cost} + T_{cost} + A_{cost} + D_{cost} + Q_{cost}) \quad (6.2)$$

In addition to S_{cost} , A_{cost} , T_{cost} , D_{cost} , and Q_{cost} , shown in C&P Equation 6.1, one additional cost R_{cost} have been added in [Riaz, 2012] for CBD. R_{cost} refers to repository cost or library cost because in CBD, components will be stored in a repository. As a consequence, there must be some cost associated with it to manage repository or library of components.

In addition to this, we have also added two more cost GD_{cost} and T_{cost} for *general or domain-specific* component-based development with *traceability of concerns*. Indeed, T_{cost} refers to the cost of traceability management, while, GD_{cost} refers to the cost of generic or domain-specific component choice. It is accepted that more effort is required to make component generic or specific so there must be an addition of generic or specialized cost to make component generic or specific.

Cost Benefit Analysis for our Case Studies

Equation 6.1 and Equation 6.2 are applied to perform cost benefit analysis of each component of our case studies shown in Section 6.4 and Section 6.5. They are applied in C&P and CBD software reuse approaches respectively in order to compare the two approaches.

1. **Cost Benefit Analysis with C&P approach.** By using Equation 6.1, the cost benefit analysis process is applied in different steps :

- In step 1, we select components on which the analysis is to be performed. For our first case study (LC-APS), we do not have component available for reusability: all the components are developed from scratch without keeping reuse and traceability in mind.
- In step 2, we estimate the hours stand for implementing, documenting and testing. The total cost of components developed from scratch, i.e., FS_{cost} , is the sum of three costs of implementation, documentation and testing. For example, the FS_{cost} of Sensor component of LC-APS is evaluated to $FS_{cost}(Sensor) = 4 + 2 + 0,8 \text{ (hours)}$.
- In step 3, the overhead cost, i.e., $(S_{cost} + A_{cost} + D_{cost} + Q_{cost})$ of Equation 6.1 is included according to reusability goal, i.e., version evolution or new project:
- In step 3.1, during the first version of LC-APS development:

S_{cost} represents the number of hours required to search a component. Here, this cost does not matter because the component is not being searched from library, nor in market. It is our in house building from scratch during the initial development version.

A_{cost} represents the number of hours spend to adapt a component during the first version. In the copy paste case this adaptation is also incurred. For example, $A_{cost}(Sensor) = 0.25 \text{ (hours)}$.

D_{cost} represents the number of hours spends on the documentation when copy and paste method will be applied. When component code is copied from one version to another then documentation is also changed. We have taken the documentation cost 20% of adaptation cost.

Q_{cost} represents the number of hours spends on the quality activities such as V&V, which is estimated to 50% of from scratch cost, FS_{cost} .

Table 6.3 shows the original developing cost (FS_{cost}) of LC-APS components with the overhead costs for the first version development.

- In step 3.2, during the second version of LC-APS development, the overhead costs are relaxed because we have knowledge of components. For example the adaptation cost is now $A_{cost}(Sensor) = 0.125 \text{ (hours)}$.

- In step 4, we apply Equation 6.1, once the total effort of determining the different costs in previous step are done. Figure 6.15(a) shows the C&P reuse saving hours for all the LC-APS components in the version 1 and 2. It shows that we have saved a number hours in second version by using C&P reuse strategy. The partial conclusion at this

Table 6.3: Cost estimation for LC-APS V1 development with reuse C&P approach

LC-APS Comps	Impl	Test	Doc	FScost	Scost	Acost	Dcost	Qcost	Total C&P
Sensor	4	2	0,8	6,8	0	0,25	0,05	3,4	3,1
Controller	6	4	1,2	11,2	0	0,5	0,1	5,6	5
Gate	4	2	0,8	6,8	0	0,25	0,05	3,4	3,1
IC1	2	1	0,4	3,4	0	0,125	0,025	1,7	1,55
IC2	1	1	0,2	2,2	0	0,05	0,01	1,1	1,04
DC1	2	1	0,4	3,4	0	0,125	0,025	1,7	1,55
DC1	1	1	0,2	2,2	0	0,05	0,01	1,1	1,04
DC2	1	1	0,2	2,2	0	0,05	0,01	1,1	1,04
DC3	1	1	0,2	2,2	0	0,05	0,01	1,1	1,04
DC4	1	1	0,2	2,2	0	0,05	0,01	1,1	1,04
DC5	1	1	0,2	2,2	0	0,05	0,01	1,1	1,04
DC6	1	1	0,2	2,2	0	0,05	0,01	1,1	1,04
OC1	2	1	0,4	3,4	0	0,125	0,025	1,7	1,55
OC2	1	1	0,2	2,2	0	0,05	0,01	1,1	1,04

step is that C&P reuse strategy is not totally bad because it saves a number of hours from evolution of versions in the same project.

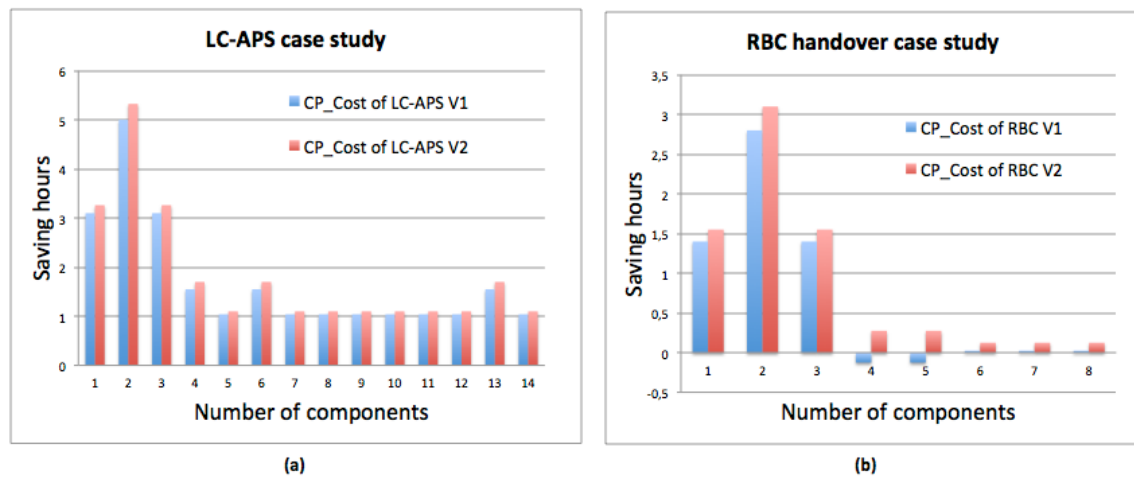


Figure 6.15: Cost benefit analysis with C&P approach

- In step 5, we repeat the above steps to evaluate the reuse of components in RBC case study, which is considered as a new project different from LC-APS project. The difference here is that we have to include S_{cost} and A_{cost} that represents the number of hours required to search components in the LC-APS project and to adapt them for RBC project. In this search only the buffer component has been reused between the

two project. Figure 6.15(b) shows the C&P reuse saving hours for the RBC handover case study.

Important thing that should be noticed here is that two components number 4 and 5 in Figure 6.15(b) have not reuse saving. It is because there are a lot of overhead costs, such as seaching, adaptation, re-documentation, re-testing, involved to reuse these components. A major disadvantage of C&P approach is that all activities (searching, adaptation, documentation and testing and so on) still have to be performed from the scratch to ensure that components will work in a new environment and have enough quality to meet the specified requirements.

2. **Cost Benefit Analysis with CBD approach.** Now we discuss the cost benefit analysis process for CBD on our two case study components and evaluate how much effort can be saved by using CBD. The same process is followed in CBD as that has been described above for C&P reuse strategy. The only difference is that here we use Equation 6.2. This equation has some more overhead costs: R_{cost} for repository cost, T_{cost} for traceability management and GD_{cost} for generic or domain-specific component development.

Figure 6.16(a) and Figure 6.16(b) shows the CBD reuse saving hours for the LC-APS and RBC handover case studies, respectively.

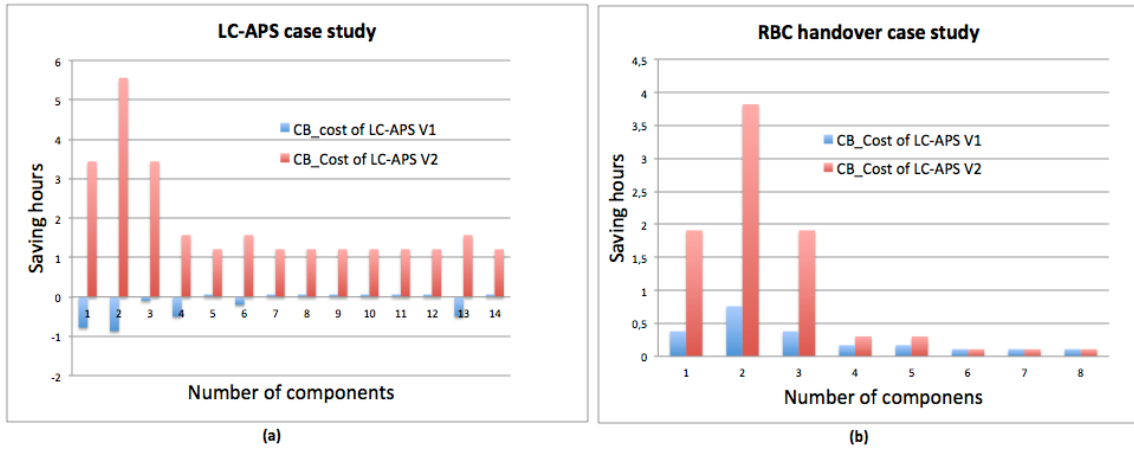


Figure 6.16: Cost benefit analysis with CBD approach

These histograms show that CBD reuse strategy while not delivering a benefit in saving hours for the initial application of CBD, see Figure 6.16(a), the CB_cost of LC-APS V1. This is not surprising because it is widely accepted that making components generic for reuse will require additional development time, while not delivering a benefit in savings for the initial projects. In contrast, here in domain specific component language with traceability, such as our SARA approach, this requires a first initialisation for the domain language and the traceability link definition in the model. However, as shown in Figure 6.16, when components have been reused in the second ver-

sion of the same project, *i.e.*, CB_cost of LC-APS V2 in Figure 6.16(a), or in another project *i.e.*, Figure 6.16(b), they recover their costs as well as they provide potential benefits.

3. **Comparison between C&P and CBD reuse approaches.** From the above analysis one can notice that the C&P reuse strategy is better than CBD in first version of our case studies. This is illustrated in Figure 6.17 by comparing C&P with CBD cost in the first version development of LC-APS case study.

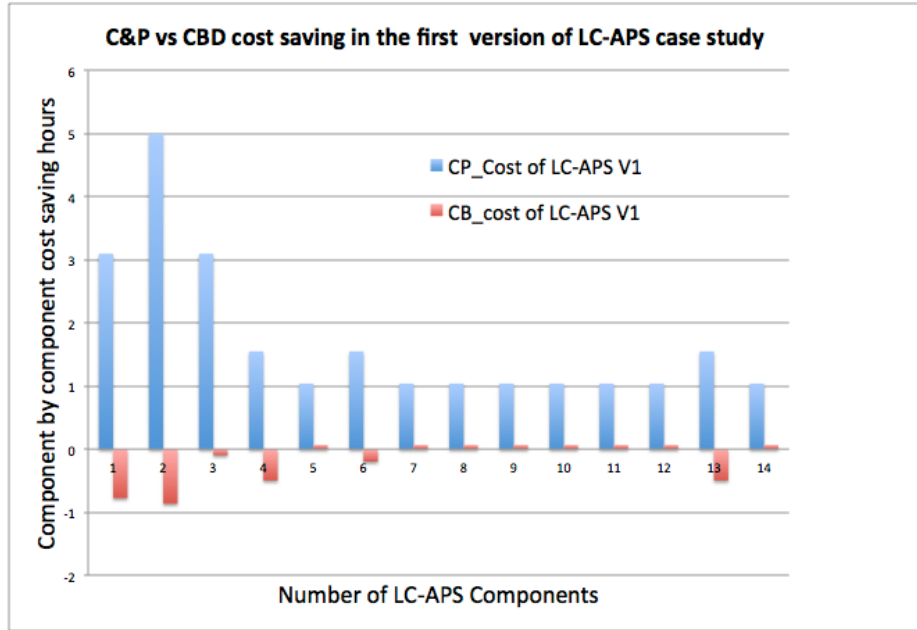


Figure 6.17: Component by component cost saving in both C&P and CBD approaches

Figure 6.17 shows that the C&P is an attractive ad-hoc reuse strategy for the users who are looking at reusability for rapid benefits. It can be a good short term option for those organizations that do not have enough resources to move on a systematic reuse strategy. However, it has no long term benefits associated with this technique. For example, Figure 6.18 shows the total saving hours curve in both approaches when all components have used in three versions of LC-APS project.

Generally, this result is not surprising because CBD is a systematic approach that reduces the cost of development, as discussed in the state-of-the art in Section 2.2.

Particularly, this show also that our approach with traceability of concerns and verification of patterns can be benefit, although it requires initial cost to define traceability links between components and property patterns or requirement specifications. Indeed, Figure 6.18 shows that our CBD approach saves $26,83 = -3,40 + 30,23$ hours when all components of LC-APS component will be used in the second version, while C&P strategy still save $25,75 = 24 + 1,75$ hours because same overhead costs (searching adaptation, documentation and testing without predicted traceability link) will

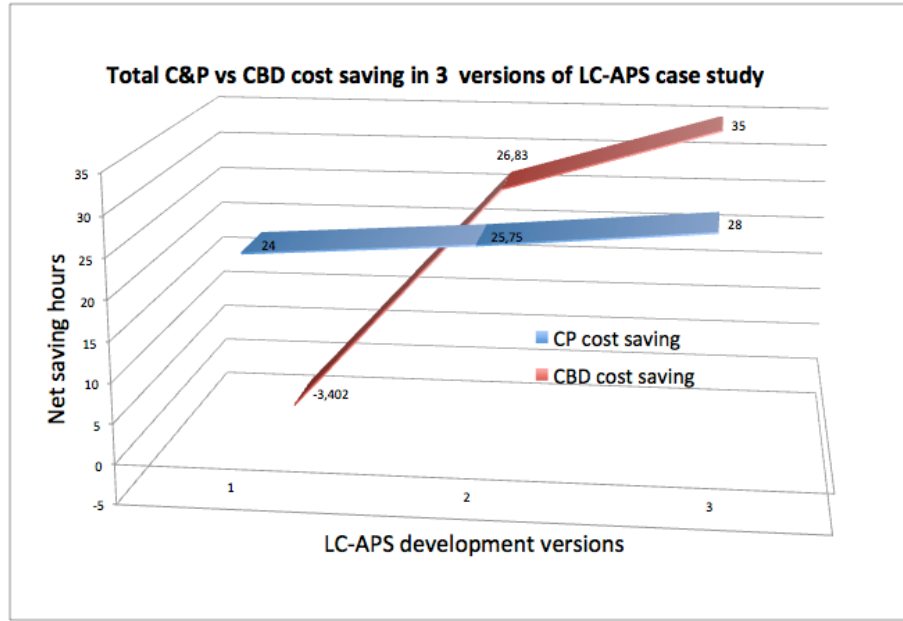


Figure 6.18: C&P versus CBD cost benefit analysis

keep on every time. In third time version, CBD save $35 = 28,83 + 6,17$ hours because verification and validation costs is reduced with pattern-based approaches.

6.7 Threats to Validity and Discussion

In this section, we present and discuss several aspects of our evaluation that may form threats to its validity.

6.7.1 Scalability Concern

In previous sections, we have evaluated the applicability of our approach through a rail-road level crossing automatic protection system (LC-APS) (see Section 6.4), and RBC-RBC handover case study (see Section 6.5). Although our approach is also applied in other case studies, such as the On-board ETCS Speed monitoring case study [Sango, 2013], the scalability of our approach is required to demonstrate it can be efficiently applied to large-scale safety-critical systems. This is the first threat to the validity of our approach. Indeed, in the large-scale ERTMS/ETCS system, we consider different use cases, such as the LC-APS and the RBC handover use cases, separately. However, in a real deployment, the LC-APS case study can depend on other use cases or other actors, such as the DMI actor, as illustrated in Figure 6.19.

For this reason, we have studied the empirical scalability of our approach by increasing the complexity of each case study and by raising the actors of each case study. As

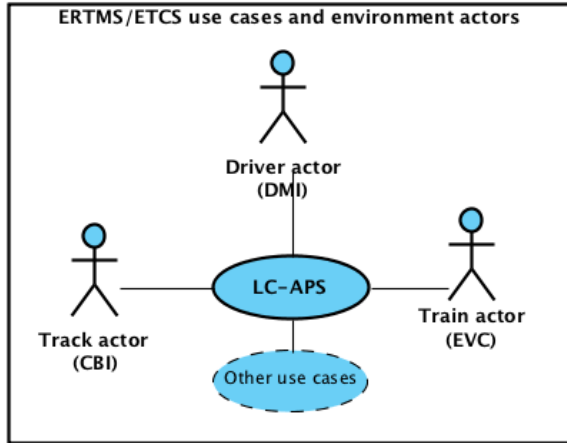


Figure 6.19: Use cases and its actors

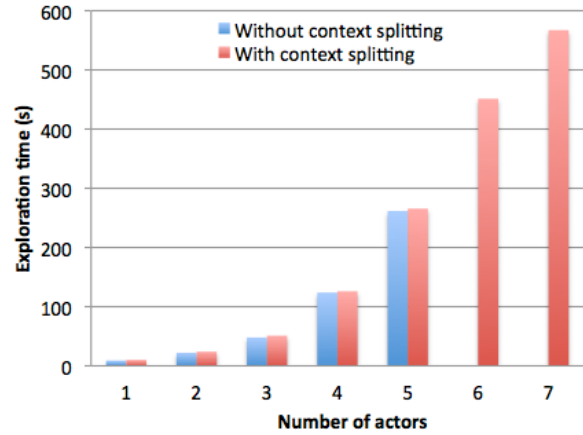


Figure 6.20: Exploration time

shown in Figure 6.20, without a context splitting of use case environment, for over five actors, the state explosion starts due to the memory limit of the computer that we used: Linux 32 bits - 2 Gb RAM. To reduce the state explosion, we adopt the environment context splitting approach [Dhaussy et al., 2012]. The main idea of this approach is to unroll the environment context into several scenarios and successively compose each scenario with the system under verification and verify the resulting composition. This approach was applied to several embedded applications in the avionic or electronic industrial domains [Dhaussy et al., 2012, Dhaussy and Teodorov, 2014]. It is based on the Context Description Language (CDL) and the Observer Based Prover (OBP) [Dhaussy, 2014]. For our case studies, we constructed several CDL models of our use case model with different complexities depending on the number of actors. The tests are successively performed on each CDL model composed with our use case model. We reported the results of OBP for these experiments and compared to our regular UPPAAL verification approach without context splitting [Sango et al., 2014a]. As illustrated in the histogram of Figure 6.20, with context splitting, the exploration is not limited but the time of exploration increases. Since the verification is realized offline, we can conclude that environment context awareness is one of the promising ways to deal with scalability in our modeling and verification approach.

However, during our experimentation of this approach, the major difficulty we faced was the identification of complete and coherent description of different interactions between the system under observation and its environment in the ERTMS/ETCS requirement documents, while the effective application of the context partitioning approach requires a strong assumption that consists in identifying all the possible interactions with the environment [Dhaussy et al., 2012]. This strong assumption is justified, particularly in the context of an embedded system, by the fact that the designer of a software component needs to know precisely and completely the perimeter (constraints, conditions) of its system in order to develop it properly. In addition, system designers can only take counter-measures for the errors that can be anticipated at specification and design time, *i.e.*, errors encountered at de-

velopment and verification time. However, in large and complex software, it is impossible to predict all error cases that will happen in real-world environments. As a consequence, to estimate the railway system's availability at runtime, the probability of influence brought by such uncertainties should also be taken into account. One area of improvement is software uncertainty analysis [Lehman and Ramil, 2002, Qiu et al., 2014].

6.7.2 Safety Risk Assessment

The second threat of validity is the safety risk assessment in requirement change. Indeed, when dealing with the traceability management and the requirement change impact, a rigorous safety risk assessment is required to meet the high Safety Integrity Level (SIL) [Beugin et al., 2007]. For example, in France, when railway functionality is replaced, modified or updated, the well-known safety principle, named GAME for "Globalement Au Moins Equivalent" in French, meaning "globally at least as good", has to be fulfilled.

6.8 Summary

In this chapter, we have presented a validation for our SARA component-based modeling with traceability of concerns and the observer-based verification with patterns of properties. We presented the implementation details of various concerns of our approach through two concrete case studies of ERTMS/ETCS specification, and then evaluated the traceability and observer-based verification concerns.

Evaluation of transformation rule implementation between the SARA model and the TAO model shows that the transformation rules defined do not increase significantly the complexity of the transformation. Moreover, as our cost empirical evaluation shows, we observe that development our SARA approach is well-suited to handle the modeling with traceability concerns while saving the cost for users who are looking at reusability for long term benefits. For larger-scale systems, an evaluation of an approach, which splits the environment context of the system under study, shows that context awareness is one of the promising ways to deal with scalability in our modeling and verification approach.

Globally, the major difficulties we faced during our evaluation are: (i) the identification of complete and coherent description of different interactions between the system or component under verification and its environment, and (ii) the anticipation of errors at design time, *i.e.*, errors encountered at development and verification time, because it is impossible to predict all error cases that will happen in real-world operation environment. One area to estimate such uncertainties in component development is to specify a target level of risk reduction, known as Safety integrity level (SIL).

This chapter concludes the third part of this document, which was dedicated to the validation of our approach. In the following chapter, we summarize the main contributions of this dissertation, present the conclusions of the research work, and define a set of perspectives for future work.

Part V

Conclusion

Conclusion and Perspectives

Contents

7.1	Summary of the Dissertation	151
7.2	Review of Research Questions	152
7.3	Perspectives	154
7.3.1	Short-term Perspectives	154
7.3.2	Long-term Perspectives	155

In this chapter, we summarize in Section 7.1 our thesis dissertation by discussing the challenges and goals addressed. Then, we outline our contributions in Section 7.2, regarding the research questions stated in Chapter 1. Finally, we discuss in Section 7.3 our short-term and long-term perspectives related to the work presented in this dissertation.

7.1 Summary of the Dissertation

Component-based software engineering is a systematic way to achieve software reuse. Several component based development approaches have been discussed and proposed in the literature and some of them have been discussed in this thesis. This is an evidence or a witness of a subject domain richness with technical and scientific challenges, and considerable potential. Unfortunately, however, the reported level of adoption has been comparatively low in industrial domains, particularly in safety-critical industrial domains.

The major difference between the traditional software development processes and Component-Based Development (CBD) processes is a separation of system development from component development in CBD. The goal of the CBD system development is to specify an ideal life cycle process model according to the fundamental CBD rules, such as reusability, composition, interoperability and so on.

But in the real industrial scenario, it is very rare to use CBD idealized lifecycle models as described in the literature. Most of the domains are still using traditional software development approaches by just adding to them some additional activities related to component based software development. Because these domains have been coupled with traditional software development for several reasons, it is very difficult to achieve the goals by using only component based software development.

Due to this, idealized approaches need adjustments. Mostly CBD will be successful in these critical domains compared to traditional software development if CDB decreases the cost of V&V activities, which is a sinew of war in these domains. Under this current situation, it is further needed to plan sequence of experiments, in which relative costs and benefits of choosing a component based software development can be weighed against the choice of a traditional software development.

It is in this context that we have described in this thesis a possible approach to merge component-based modeling with traceability of concerns and observer-based verification with patterns of properties, with the aim of providing a solution towards an efficient development of embedded safety-critical software, particularly in railway control software.

The solution we proposed is based on well-accepted and defined technologies and standards. In particular, our approach, called SARA, relies on the principles of CBSE. We bring together these principles from different CBD approaches to provide a simple and yet realistic CBD that can be applicable in train control applications.

7.2 Review of Research Questions

In this section, we outline and discuss our contributions regarding the research questions stated in Section 1.1:

Research Question 1. What is a suitable component model to build safety-critical control software, specially railway control and protection software, with traceability of concerns?

There are a lot of component models or framework available in the literature. Some of them have been discussed in Chapter 2. However, few of them explicitly introduced traceability of concerns in their underlining meta-model. Thus, we have studied in Chapter 3 some of the most relevant work related to traceability of concerns by highlighting the benefits for component based modeling. Our contribution for component-based modeling with traceability of concerns is presented in Chapter 4 as a set of meta-models. The component meta-model reifies relevant concepts and elements from existing component models with and an extension for traceability of concerns.

In railway control applications, it is still not guaranteed to be able to purchase trusted and certified Commercial Off The Shelf (COTS) components and build software from them. As a consequence it is difficult to apply an idealized component-based software development. For this we think that a suitable component model for railway control software,

should be a hybrid model to support both component based software development with component concerns as well as traditional software development with requirement concerns.

Research Question 2. How can safety-critical control software, specially railway control and protection software, be built in an efficient way by using CBD-V rules, such as interoperability and model verification?

A CBD-V cannot be completely put to use if the development and verification organizations have not adopted all its fundamental rules, such as interoperability, composition, reusability and model verification. In addition, the CBD-V is successful if the interoperable, reusable and certified components are available for component based software development, and if V&V activities are decreased in the CBD-V compared to traditional software development. As a consequence, formal modeling of interfaces that enables interoperability is necessary to reason and predict the effective application of component composition and substitution.

Three main levels of interoperability have been distinguished in the contracts of component interfaces: (i) *the signature level* covers the static aspects of component interoperation; (ii) *the semantic level* covers the behavioral aspects of component interoperation; and (iii) *the protocol level* deals with the order in which a component expects its methods to be called. Particularly, for the protocol level, we translated our formal model into a timed automata model, for which we can verify real-time constraints.

Research Question 3. Will CBD-V processes replace the need for traditional software development and verification processes? Particularly, what is the suitable development and verification process for railway control and protection software?

As already discussed in Section 2.3, the adoption of any new development process, particularly a CBD-V process, in a specific embedded industry, will depend on the industrial needs, its process, its technology and its market structures. In complex applications, such as railway control and protection applications, there are a lot of stakeholders, such as manufacturers and vendors. For example, various components developed by manufacturers can be supplied by different vendors, so the requirements are not stable. Since requirements change from customer to customer and are still implemented using traditional software development, it is very difficult to build the complete software by using only generic components. As a consequence, it is very difficult to replace the traditional life cycle completely with the CBD-V because there are always some requirements that are project specific or customer specific.

One possible way to avoid the mismatch is to negotiate the requirements and modify them to be able to use the existing components, but it is not always easy because customers may insist on some requirements and may not be ready to negotiate them. Another possible way to solve this problem is to develop new reusable software components or adapt existing reusable software components that meet the user requirements, and which can be used in future projects. This is the solution we adopt in this thesis. But this adoption raises

further challenges, such as version management and re-verification. As a consequence, we believe that the suitable development process for safety-critical software is the reusable development process, such as component-based development with traceability of concerns and observer-based verification with pattern of concerns, which should reduce the cost of V&V activities compared to traditional software development.

7.3 Perspectives

Although the work presented in this dissertation covers the needs of our research goal stated in Section 1.2, there is still some work that could be done to improve our research. In this section, we thus discuss some short-term and long-term perspectives that should be considered in the continuation of this work.

7.3.1 Short-term Perspectives

Integrated Tool-Chain Demonstrator

As explained in Chapter 6, to evaluate our approach we have used several tools. For meta-models, we mainly concentrated on providing minimal support in XML-based representations and XQuery queries to provide a quick feedback about the current state of our approach. For the formal model transformation we have implemented our transformation rules with two ATL and QVT transformation languages. The objective was to study the effect of different language constructs on the performance of the transformation. For verification tasks, we have used the SPARK tool for formal proof of functional input and output constraints and the UPPAAL tool for non-functional real-time constraints.

Although these different tools are powerful in their domain, their use separately does not provide an integrated tool-chain demonstrator. In this direction, one essential challenge when dealing with model transformation is to take advantage of feedback obtained in the verification low-level model to lead to the design high-level model. As a consequence, one short term direction is to take advantage of component-based model-driven techniques and traceability of concerns in tooling in order to have validation feedback on design models automatically.

Improve Scalability and Applicability

Although our approach has been shown to be scalable and applicable in several ERTM-S/ETCS case studies, our approach can be enhanced in different directions.

On the scalability side, currently, our approach is applied on each use case separately. However, as explained in Section 6.7, in a real scenario deployment, one use case can depend

on other use cases or other actors, which we consider as the environment of the system under study. To face this problem, we investigated the environment context partitioning approach, which consists in unrolling the environment context into several scenarios and successively composing each scenario with the use case under verification and verifying the resulting composition. The first results are encouraging, but this approach requires a domain knowledge in order to identify all the possible interactions with the use case environment. As a consequence, it requires more investigation to demonstrate an efficient scalability to large-scale railway control and protection systems.

On the pattern-based applicability side, experiments show that part of requirements found in the ERTMS/ETCS documents cannot be directly translated into real-time extensions of Dwyer et al. patterns used in our approach. One direction to face this problem is to extend current patterns and another is to define other patterns. In this direction, one theoretical issues is the use theorem proving techniques to support the formal validation of observers defined for railway requirement specification patterns. In addition, observers of patterns are actually manually defined and selected. A need to define an improved method to select best observers is another direction.

7.3.2 Long-term Perspectives

Mixed-Critical Components

The presumption of our approach is that if a lower critical component is running on the same secure compute platform with a higher critical component, it must be treated as the higher critical component. This presumption is set for the actual certification process. For example, the railway certification requirements require to produce the evidence and to demonstrate the required Safety Integrity Level (SIL), ranging from SIL 1 to SIL 4, which is the most dependable. However, this presumption, increases the Verification and Validation (V&V) effort. In fact, it requires to provide the same V&V effort for safety-critical, mission-critical and non-critical components when a single and secure compute platform is used. Another solution is the safety-bag approach, where the lower critical components and the higher critical components must be separately allocated to distinct processors or replicated processors. However, this absolute physical separation is sometimes too expensive to be applied. As a consequence, an efficient mixed-critical solution is required for the next generation of mixed-critical software, *i.e.*, mixed safety-critical, mission-critical, and non-critical components.

Towards Component-based Domain Engineering

Regarding Software Product Line (SPL) Processes, in this thesis we have dealt with component-based *application engineering*. Indeed SPL engineering is divided into two complementary processes: *Domain Engineering* and *Application Engineering*. The former usually refers to the development for reuse, while the latter is the development with reuse. In other

words, the domain engineering process is responsible for creating reusable assets, named *reference architectures* [Kang et al., 1998], while application engineering is the process of reusing those assets to build individual but similar software products.

In space and automotive software control domains, there are well-defined methods for constructing software reference architectures, such as [Panunzio and Vardanega, 2013] and [AUTOSAR, 2006], respectively. In the railway domain, [openETCS, 2012] project is recently driven by a European consortium in order to provide an Open Proofs Methodology for the European Train Control Onboard System. The purpose of the openETCS project is to develop an integrated modelling, development, validation and testing framework for leveraging the cost-efficient and reliable implementation of ETCS.

However, to the best of our knowledge, there are no well-defined methods for constructing software reference architectures in the railway control and protection systems. To go toward component-based domain engineering, *i.e.*, building a software reference architecture, one challenge is to allow explicit representation of variation points in the reference architecture, as defined in the feature model [Kang et al., 1998], where different features may be selected for a specific product. One research question which can be raised is: how to extend existing component-based meta-models, such as our defined meta-model, to allow explicit representation of variation points for a reference architecture?

Appendices

SARA Model Implementation in Ada Language

In this appendix, we report the minimal code archetypes written in Ada programming language to evaluate the SARA model.

```
1— This package is an implementation of SARA component model for Ada.
2— It is the root package of all packages of Sara component entities.
3package Sara is
4  pragma Pure;
5end Sara;
```

Listing A.1: Specification of SARA component model

```
1— Package Sara.Types_Root defines a minimal notion of type
2— that all types system should implement.
3package Sara.Types_Root is
4  type Root is Interface;
5  type Root_Ptr is access all Root'Class;
6  function Is_Subtype_Of_Root (Atype : Root) return Boolean is abstract;
7  — test if Atype is a subtype of Root
8end Sara.Types_Root;
```

Listing A.2: A minimal notion of type


```
1— Package Sara.Interfaces defines SARA component interfaces concept
2with Sara.Typeroot;
3package Sara.Interfaces is
4  type Itf_Type is Interface and Sara.Typeroot.Root;
5  type Itf_Type_Ptr is access all Itf_Type'Class;
6
7  Procedure Basic_Operation (This : in out Itf_Type) is abstract;
8  function Itf_Operations_Number (This : Itf_Type) return Natural is abstract;
9  function Is_Provided_Itf (This : Itf_Type) return Boolean is abstract;
10 function Get_Itf_Signature (This : Itf_Type) return String is abstract;
11end Sara.Interfaces;
```

Listing A.3: Specification of SARA component interfaces

```
1— Package Sara.Component defines Sara Component that lists
2— interfaces of a component.
3with Sara.Interfaces; use Sara.Interfaces;
4package Sara.Component is
5  type Comp_Type is abstract new Sara.Interfaces.Itf_Type with
6    record
7      Basic_Required_Itf : Itf_Type_Ptr;
8    end record;
9  type Comp_Type_Ptr is access all Comp_Type'Class;
10 type Comp_Array is array (Natural range <>) of Comp_Type_Ptr;
11
12 function Is_Basic_Component (This : in Comp_Type) return boolean is abstract;
13 — the component is basic or hierarchical component
14 function Get_Itf_Owner (This : Itf_Type) return Comp_Array is abstract;
15 — introspection that provides or not the inner components
16 function Get_Itf_Type (This : Itf_Type) return Itf_Type'Class is abstract;
17 — introspection that provides the interfaces type
18end Sara.Component;
```

Listing A.4: Specification of SARA basic component

```

1 pragma Profile (Ravenscar ) ;
2 with System;
3 with Ada.Real_Time;
4 with Sara.Errors;
5 generic
6   -----
7   — Mapping of component element —
8   -----
9   type Component_Type is private;
10  Component : in Component_Type;
11
12  -----
13  — Task parameters —
14  -----
15  Task_Period      : in Ada.Real_Time.Time_Span;
16  — Task minimal inter-arrival time of events
17  Task_Deadline    : in Ada.Real_Time.Time_Span;
18  — Task deadline
19  Task_Priority    : in System.Any_Priority;
20  — Task priority
21
22  -----
23  — Task Jobs —
24  -----
25  with procedure Wait (Component : Component_Type);
26  — Blocks the next triggering of the thread
27  with procedure Job (Component : in out Component_Type;
28                      Error : out Sara.Errors.Error_Kind);
29  — Procedure to call at each dispatch of the sporadic thread
30  with procedure Recover_Job;
31  — If given, the task runs Recover_Entrypoint when an error is
32  — detected.
33 package Sara.ComponentTask is
34   task Sporadic_Task is
35     pragma Priority (Task_Priority);
36   end Sporadic_Task;
37
38   -----
39   — return Task Jobs result —
40   -----
41   function Return_Job_Result return Sara.Errors.Error_Kind;
42 end Sara.ComponentTask;

```

Listing A.5: Specification of SARA component tasks

Bibliography

- [Abid et al., 2014] Abid, N., Zilio, S. D., and Botlan, D. L. (2014). A formal framework to specify and verify real-time properties on critical systems. *Int. J. Crit. Comput.-Based Syst.*, 5(1/2):4–30. 30, 32
- [Abrial, 1996] Abrial, J.-R. (1996). *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA. 28
- [Aceto et al., 2003] Aceto, L., Bouyer, P., Burgueno, A., and Larsen, K. G. (2003). The power of reachability testing for timed automata. *Theoretical Computer Science*, 300(1-3):411–475. 30
- [Adler et al., 2011] Adler, R., Schaefer, I., Trapp, M., and Poetzsch-Heffter, A. (2011). Component-based modeling and verification of dynamic adaptation in safety-critical embedded systems. *ACM Trans. Embed. Comput. Syst.*, 10(2):20:1–20:39. 92
- [Aizenbud-Reshef et al., 2006] Aizenbud-Reshef, N., Nolan, B. T., Rubin, J., and Shaham-Gafni, Y. (2006). Model traceability. *IBM Syst. J.*, 45(3):515–526. 54
- [Alanen and Porres, 2003] Alanen, M. and Porres, I. (2003). Difference and union of models. In Stevens, P., Whittle, J., and Bouch, G., editors, *The Unified Modeling Language. Modeling Languages and Applications*, volume 2863 of *Lecture Notes in Computer Science*, pages 2–17. Springer. 72
- [Allen and Garlan, 1997] Allen, R. and Garlan, D. (1997). A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249. 20
- [Alur and Dill, 1994] Alur, R. and Dill, D. L. (1994). A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235. 100
- [Anquetil et al., 2008] Anquetil, N., Grammel, B., Galvao Lourenco da Silva, I., Noppen, J., Shakil Khan, S., Arboleda, H., Rashid, A., and Garcia, A. (2008). Traceability for model driven, software product line engineering. In *ECMDA Traceability Workshop Proceedings*, pages 77–86, Norway. SINTEF. 46, 51, 55, 56

- [Anquetil et al., 2010] Anquetil, N., Kulesza, U., Mitschke, R., Moreira, A., Royer, J.-C., Rummmler, A., and Sousa, A. (2010). A model-driven traceability framework for software product lines. *Softw. Syst. Model.*, 9(4):427–451. 43, 46, 49, 51, 53, 54
- [Aponte et al., 2012] Aponte, M., Courtieu, P., Moy, Y., and Sango, M. (2012). Maximal and compositional pattern-based loop invariants. In *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, pages 37–51. 100
- [ARTIST, 2004] ARTIST (2004). Selected topics in Embedded Systems Design: Roadmaps for Research. Technical report, From EU IST ARTIST project. 22
- [AUTOSAR, 2006] AUTOSAR (2006). Automotive open system architecture. Official website of the AUTOSAR Partnership: www.autosar.org. 17, 34, 62, 156
- [AUTOSAR-Simulink, nd] AUTOSAR-Simulink (n.d.). Generating code for autosar software components using embedded coder. <http://www.mathworks.com/automotive/standards/autosar.html>. 36
- [Bachmann et al., 2000] Bachmann, F., Bass, L., Buhman, C., Dorda, S. C., Long, F., Robert, J., Seacord, R., and Wallnau, K. (2000). Volume II: Technical Concepts of Component-Based Software Engineering, 2nd Edition. Technical report, CMU/SEI - Carnegie Mellon University/Software Engineering Institute. 16
- [Bayse et al., 2005] Bayse, E., Cavalli, A., Nunez, M., and Zaidi, F. (2005). A passive testing approach based on invariants: application to the {WAP}. *Computer Networks*, 48(2):247–266. 30
- [Becker et al., 2009] Becker, S., Koziolok, H., and Reussner, R. (2009). The palladio component model for model-driven performance prediction. *J. Syst. Softw.*, 82(1):3–22. 70
- [Behm et al., 1998] Behm, P., Desforges, P., and Meynadier, J.-M. (1998). METEOR: An Industrial Success in Formal Development. In *Proceedings of the Second International B Conference on Recent Advances in the Development and Use of the B Method*, B '98, pages 26–, London, UK, UK. Springer-Verlag. 28
- [Beizer, 1990] Beizer, B. (1990). *Software Testing Techniques (2Nd Ed.)*. Van Nostrand Reinhold Co., New York, NY, USA. 27
- [Berthomieu et al., 2009] Berthomieu, B., Bodeveix, J.-P., Chaudet, C., Zilio, S., Filali, M., and Vernadat, F. (2009). Formal verification of aadl specifications in the topcased environment. In *Proceedings of the 14th Ada-Europe International Conference on Reliable Software Technologies*, Ada-Europe'09, pages 207–221, Berlin, Heidelberg. Springer-Verlag. 36
- [Bertolino, 2007] Bertolino, A. (2007). Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering, FOSE '07*, pages 85–103, Washington, DC, USA. IEEE Computer Society. 27

-
- [Beugin et al., 2007] Beugin, J., Renaux, D., and Cauffriez, L. (2007). A SIL quantification approach based on an operating situation model for safety evaluation in complex guided transportation systems. *Reliability Engineering and System Safety*, 92(12):1686 – 1700. 147
- [Bhatti et al., 2011] Bhatti, Z., Sinha, R., and Roop, P. (2011). Observer based verification of IEC 61499 function blocks. In *Industrial Informatics*, pages 609–614. 84, 122
- [Bianculi et al., 2012] Bianculi, D., Ghezzi, C., Pautasso, C., and Senti, P. (2012). Specification patterns from research to industry: A case study in service-based applications. In *ICSE’12*, pages 968–976. IEEE Press. 32
- [Bitsch, 2001] Bitsch, F. (2001). Safety patterns - the key to formal specification of safety requirements. In *Proceedings of the 20th International Conference on Computer Safety, Reliability and Security*, pages 176–189, London, UK. 32, 33
- [Bruneton et al., 2006] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J.-B. (2006). The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284. 17, 19, 34
- [Bures et al., 2008] Bures, T., Carlson, J., Crnkovic, I., Sentilles, S., and Vulgarakis, A. (2008). Progress Component Model Reference Manual-version 1.0. Technical report, Malardalen University. 36
- [Capretz, 2005] Capretz, L. F. (2005). Y: A new component-based software life cycle model. *Journal of Computer Science*, 1(1):76–82. 20
- [CCM, 2006] CCM (2006). Corba component model, version 4.0. <http://www.omg.org/spec/CCM/4.0/>. 18
- [Chaki et al., 2007] Chaki, S., Ivers, J., Lee, P., Wallnau, K., and Zeilberger, N. (2007). Model-driven construction of certified binaries. In Engels, G., Opdyke, B., Schmidt, D., and Weil, F., editors, *Model Driven Engineering Languages and Systems*, volume 4735 of *Lecture Notes in Computer Science*, pages 666–681. Springer Berlin Heidelberg. 37
- [Chapman, 2006] Chapman, R. (2006). Correctness by construction: A manifesto for high integrity software. In *Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software - Volume 55, SCS ’05*, pages 43–46, Darlinghurst, Australia, Australia. Australian Computer Society, Inc. 4, 28
- [Chechik and Paun, 1999] Chechik, M. and Paun, D. (1999). Events in property patterns. In *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *Lecture Notes in Computer Science*, pages 154–167. Springer Berlin Heidelberg. 32
- [Chen et al., 2007a] Chen, X., He, J., Liu, Z., and Zhan, N. (2007a). A model of component-based programming. In Arbab, F. and Sirjani, M., editors, *International Symposium on Fundamentals of Software Engineering*, volume 4767 of *Lecture Notes in Computer Science*, pages 191–206. Springer Berlin Heidelberg. 86

- [Chen et al., 2007b] Chen, X., Liu, Z., and Mencl, V. (2007b). Separation of concerns and consistent integration in requirements modelling. In van Leeuwen, J., Italiano, G., van der Hoek, W., Meinel, C., Sack, H., and Plášil, F., editors, *SOFSEM 2007: Theory and Practice of Computer Science*, volume 4362 of *Lecture Notes in Computer Science*, pages 819–831. Springer Berlin Heidelberg. 4
- [Chen et al., 2009] Chen, Z., Liu, Z., Ravn, A. P., Stolz, V., and Zhan, N. (2009). Refinement and verification in component-based model-driven design. *Sci. Comput. Program.*, 74(4):168–196. 4, 18, 21, 61
- [CHESS, 2012] CHESS (2012). <http://www.chess-project.org/>. 62
- [Chung and do Prado Leite, 2009] Chung, L. and do Prado Leite, J. C. S. (2009). On non-functional requirements in software engineering. In *Conceptual Modeling: Foundations and Applications*, volume 5600 of *Lecture Notes in Computer Science*, pages 363–379. Springer Berlin Heidelberg. 45, 66, 67
- [Clarke et al., 1986] Clarke, E. M., Emerson, E. A., and Sistla, A. P. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263. 30
- [Cleland-Huang et al., 2003] Cleland-Huang, J., Chang, C., and Christensen, M. (2003). Event-based traceability for managing evolutionary change. *Software Engineering, IEEE Transactions on*, 29(9):796–810. 46, 50, 52, 53, 54
- [Cleland-Huang and Schmelzer, 2003] Cleland-Huang, J. and Schmelzer, D. (2003). Dynamically tracing non-functional requirements through design pattern invariants. In *Workshop on Traceability in Emerging Forms of Software Engineering*, page 10. 46, 50, 51, 52, 53, 54
- [Cloutier and Verma, 2007] Cloutier, R. J. and Verma, D. (2007). Applying the concept of patterns to systems architecture. *Systems Engineering*, 10(2):138–154. 67
- [Collart-Dutilleul et al., 2014] Collart-Dutilleul, S., Bon, P., El Koursi, E. M., and Lemaire, E. (2014). Study of the implementation of ERTMS with respect to French national – "non on board rules" using a collaborative methodology based on formal methods and simulation. In *TRA - Transport Research Arena*, page 8p, France. 24
- [Collofello and Institute, 1988] Collofello, J. and Institute, C.-M. U. S. E. (1988). *Introduction to Software Verification and Validation*. Technical report (Carnegie-Mellon University. Software Engineering Institute). Carnegie Mellon University, Software Engineering Institute. 25
- [Corin et al., 2005] Corin, R., Etalle, S., Hartel, P., and Durante, A. (2005). A trace logic for local security properties. *Electronic Notes in Theoretical Computer Science*, 118(0):129 – 143. Proceedings of the International Workshop on Software Verification and Validation (SVV 2003) Software Verification and Validation 2003. 93

-
- [Crnkovic, 2002] Crnkovic, I. (2002). *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA. 19
- [Crnkovic, 2005] Crnkovic, I. (2005). Component-based software engineering for embedded systems. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 712–713. 22
- [Crnkovic et al., 2006] Crnkovic, I., Chaudron, M., and Larsson, S. (2006). Component-based development process and component lifecycle. In *Proceedings of the International Conference on Software Engineering Advances, ICSEA '06*, pages 44–50, Washington, DC, USA. IEEE Computer Society. 20
- [Crnkovic et al., 2003] Crnkovic, I., Schmidt, H., Stafford, J., and Wallnau, K. (2003). 6th icse workshop on component-based software engineering: Automated reasoning and prediction. In *Proceedings of the 25th International Conference on Software Engineering, ICSE '03*, pages 775–776, Washington, DC, USA. IEEE Computer Society. 19
- [Crnkovic et al., 2011] Crnkovic, I., Sentilles, S., Vulgarakis, A., and Chaudron, M. R. V. (2011). A classification framework for software component models. *IEEE Trans. Software Eng.*, 37(5):593–615. 18, 19, 33, 62, 91
- [Dhaussy, 2014] Dhaussy, P. (access june 2014). A language : Context description language (cdl) – a toolset : Observer based prover (obp). Technical report, ENSTA-Bretagne, UMR CNRS 6285. <http://www.obpcdl.org>. 146
- [Dhaussy et al., 2012] Dhaussy, P., Boniol, F., Roger, J.-C., and Leroux, L. (2012). Improving model checking with context modelling. *Adv. Soft. Eng.*, 2012:9:9–9:9. 146
- [Dhaussy and Teodorov, 2014] Dhaussy, P. and Teodorov, C. (2014). Context-aware verification of a landing gear system. In *ABZ 2014: The Landing Gear Case Study*, volume 433 of *Communications in Computer and Information Science*, pages 52–65. 146
- [Dijkstra, 1982a] Dijkstra, E. (1982a). On the role of scientific thought. In *Selected Writings on Computing: A personal Perspective*, Texts and Monographs in Computer Science, pages 60–66. Springer New York. 55
- [Dijkstra, 1972] Dijkstra, E. W. (1972). The humble programmer. *Commun. ACM*, 15(10):859–866. 28
- [Dijkstra, 1982b] Dijkstra, E. W. (1982b). On the role of scientific thought. In *Selected Writings on Computing: A personal Perspective*, Texts and Monographs in Computer Science, pages 60–66. Springer New York. 4
- [DO-331, 2011] DO-331 (2011). *Model-Based Development and Verification Supplement to DO-178C and DO-278A*. RTCA. 3
- [Dömges and Pohl, 1998] Dömges, R. and Pohl, K. (1998). Adapting traceability environments to project-specific needs. *Commun. ACM*, 41(12):54–62. 35, 54, 55, 113

- [Dromey, 2003] Dromey, R. (2003). From requirements to design: formalizing the key steps. In *Software Engineering and Formal Methods, 2003.Proceedings. First International Conference on*, pages 2–11. 51
- [Dwyer, nd] Dwyer, M. B. (n.d.). Specification patterns web site. <http://patterns.projects.cis.ksu.edu/documentation/patterns.shtml>. 31
- [Dwyer et al., 1998] Dwyer, M. B., Avrunin, G. S., and Corbett, J. C. (1998). Property specification patterns for finite-state verification. In *Proceedings of the Second Workshop on Formal Methods in Software Practice, March 4-5, 1998, Clearwater Beach, Florida, USA*, pages 7–15. 30
- [Dwyer et al., 1999] Dwyer, M. B., Avrunin, G. S., and Corbett, J. C. (1999). Patterns in property specifications for finite-state verification. In *Proceedings of ICSE'99*, pages 411–420. 30
- [Egyed, 2003] Egyed, A. (2003). A scenario-driven approach to trace dependency analysis. *IEEE Trans. Softw. Eng.*, 29(2):116–132. 46
- [El-Koursi et al., 2009] El-Koursi, E.-M., Khoudour, L., Impastato, S., Malavasi, G., and Ricci, S. (2009). *Safer European level crossing appraisal and technology*. Collection Actes INRETS. 122, 123
- [Emmelmann, 2003] Emmelmann, M. (2003). An integrated prototyping and simulation architecture for space specific protocol developments and verifications. In *Proceedings of the 1rd NASA Space Internet Workshop*, Cleveland, OH, USA. 26
- [EN-50128, 2011] EN-50128 (2011). *Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems*. CENELEC, Brussels. 3, 4, 6, 20, 24, 40, 55, 70, 118
- [ERSA, nd] ERSA (nd). Ertms/etcs operational simulator. <http://www.ersa-france.com/index.php/en/menu-products>. 137
- [ERTMS/ETCS, 2006] ERTMS/ETCS (2006). European rail traffic management system/european train control system. System Requirements Specification, Version 2.3.0. <http://www.era.europa.eu/Document-Register/Pages/UNISIGSUBSET-026.aspx>. 63
- [ERTMS/ETCS, 2014] ERTMS/ETCS (2014). European rail traffic management system/european train control system. System Requirements Specification, Version 3.4.0. [http://www.era.europa.eu/Document-Register/Pages/SystemRequirementsSpecification\(Recommendation\).aspx](http://www.era.europa.eu/Document-Register/Pages/SystemRequirementsSpecification(Recommendation).aspx). 24, 63, 68, 120, 121, 123, 124, 131, 132
- [Eurostat, 2012] Eurostat (access june 2012). Railway safety statistics, available from. http://epp.eurostat.ec.europa.eu/statistics_explained/index.php/Railway_safety_statistics. 122

-
- [Falleri et al., 2006] Falleri, J.-R., Huchard, M., and Nebut, C. (2006). Towards a Traceability Framework for Model Transformations in Kermeta. In *ECMDA'06 Traceability Workshop*, pages 31–40, Bilbao (Spain). 46, 50, 51, 52, 53, 54, 72, 73
- [Fassino et al., 2002] Fassino, J.-P., Stefani, J.-B., Lawall, J. L., and Muller, G. (2002). Think: A software framework for component-based operating system kernels. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference, ATEC '02*, pages 73–86, Berkeley, CA, USA. USENIX Association. 34
- [Favaro and Sartori, 2014] Favaro, J. and Sartori, G. (2014). Model-based software design in rail - a european perspective. *EURAILmag - THE MAGAZINE FOR EUROPEAN RAIL DECISION MAKERS*, 30(2):224–227. 4, 62
- [Feiler and Gluch, 2012] Feiler, P. H. and Gluch, D. P. (2012). *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, 1st edition. 34
- [Fenton and Pfleeger, 1998] Fenton, N. E. and Pfleeger, S. L. (1998). *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 2nd edition. 137
- [Ferrari et al., 2013] Ferrari, A., Gnesi, S., and Tolomei, G. (2013). Using clustering to improve the structure of natural language requirements documents. In *Proceedings of the 19th International Conference on Requirements Engineering: Foundation for Software Quality, REFSQ'13*, pages 34–49, Berlin, Heidelberg. Springer-Verlag. 26
- [Galvao and Goknil, 2007] Galvao, I. and Goknil, A. (2007). Survey of traceability approaches in model-driven engineering. In *Enterprise Distributed Object Computing Conference*, pages 313–313. 43, 49
- [Giese et al., 2006] Giese, H., Glesner, S., Leitner, J., Schafer, W., and Wagner, R. (2006). Towards verified model transformations. In *Workshop associated to MODELS'06*, pages 78–93. 106
- [Gill, 2003] Gill, N. S. (2003). Reusability issues in component-based development. *SIGSOFT Softw. Eng. Notes*, 28(4):4–4. 139, 140
- [Glinz, 2007] Glinz, M. (2007). On non-functional requirements. In *15th IEEE International Requirements Engineering Conference*, pages 21–26. 45, 66, 67
- [Gotel and Finkelstein, 1994] Gotel, O. C. Z. and Finkelstein, A. C. W. (1994). An analysis of the requirements traceability problem. In *Proceedings of the First IEEE International Conference on Requirements Engineering*, pages 94–101. 40, 41
- [Groß, 2005] Groß, H. (2005). *Component-based software testing with UML*. Springer. 27
- [Gruhn and Laue, 2006] Gruhn, V. and Laue, R. (2006). Patterns for timed property specifications. *Electronic Notes in Theoretical Computer Science*, 153(2):117–133. 30, 32

- [Guiho and Hennebert, 1990] Guiho, G. and Hennebert, C. (1990). Sacem software validation. In *Proceedings of the 12th International Conference on Software Engineering, ICSE '90*, pages 186–191, Los Alamitos, CA, USA. IEEE Computer Society Press. 28
- [Hall and Chapman, 2002] Hall, A. and Chapman, R. (2002). Correctness by construction: developing a commercial secure system. *Software, IEEE*, 19(1):18–25. 28
- [He et al., 2006] He, J., Li, X., and Liu, Z. (2006). A theory of reactive components. *Electronic Notes in Theoretical Computer Science*, 160(0):173 – 195. Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005) Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005). 4, 20, 84, 87, 94, 98, 113
- [Heineman and Councill, 2001] Heineman, G. T. and Councill, W. T., editors (2001). *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 16
- [Henzinger, 1998] Henzinger, T. (1998). It’s about time: Real-time logics reviewed. In *Concurrency Theory*, volume 1466 of *LNCS*, pages 439–454. Springer Berlin Heidelberg. 29, 84
- [Hissam et al., 2005] Hissam, S., Ivers, J., Plakosh, D., and Wallnau, K. (2005). Pin component technology (v1.0) and its c interface (cmu/sei-2005-tn-00). Technical report, Carnegie Mellon Software Engineering Institute (SEI). 34, 36
- [Hoare and He, 1998] Hoare, C. A. R. and He, J. (1998). Unifying theories of programming. In *Prentice-Hall*. 94
- [Hoinaru et al., 2014] Hoinaru, O., Gransart, C., Mariano, G., and Lemaire, E. (2014). An ontology for the ERTMS/ETCS. In *Transport Research Arena*, page 10p, Paris. 121
- [Hoinaru et al., 2013] Hoinaru, O., Mariano, G., and Gransart, C. (2013). Ontology for complex railway systems application to ERTMS/ETCS system. In *FM-RAIL-BOK Workshop in SEFM’2013 11th International Conference on Software Engineering and Formal Methods*, page 6p, Espagne. 121
- [Huang et al., 2011] Huang, C.-Y., Yin, Y.-F., Hsu, C.-J., Huang, T., and Chang, T.-M. (2011). Soc hw/sw verification and validation. In *Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific*, pages 297–300. 26
- [Hirsch and Lopes, 1995] Hirsch, W. L. and Lopes, C. V. (1995). Separation of concerns. Technical report, Technical report by the College of Computer Science, Northeastern University, Boston, MA 02115, USA. 54
- [IEC-61499, 2005] IEC-61499 (2005). *IEC 61499 function blocks for industrial-process measurement and control systems*. Geneva, Switzerland. 34

-
- [IEEE-Std-610, 1990] IEEE-Std-610 (1990). Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84. 25, 40
- [ISO-26262, 2009] ISO-26262 (2009). *Road vehicles, Functional safety, Part 6: Product development: software level*. 3, 24
- [Jackson, 2006] Jackson, D. (2006). *Software Abstractions: Logic, Language, and Analysis*. The MIT Press. 27, 56
- [Jarke et al., 1995] Jarke, M., Gallersdörfer, R., Jeusfeld, M. A., Staudt, M., and Eherer, S. (1995). Conceptbase—a deductive object base for meta data management. *J. Intell. Inf. Syst.*, 4(2):167–192. 54
- [Jasmine and Vasantha, 2008] Jasmine, K. and Vasantha, D. R. (2008). Cost estimation model for reuse based software products. *Proceedings of the International MultiConference of Engineers and Computer Scientists*, 1. 140
- [Johansson, 2001] Johansson, R. (2001). Dependability characteristics and safety criteria for an embedded distributed brake control system in railway freight trains. Technical report, Department of Electrical and Computer Engineering Chalmers Lindholmen University College. 23, 24
- [Jouault, 2005] Jouault, F. (2005). Loosely Coupled Traceability for ATL. In *ECMDA’05 Traceability Workshop*, pages 29–37. 72
- [Jouault et al., 2006] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., and Valduriez, P. (2006). Atl: A qvt-like transformation language. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications, OOPSLA ’06*, pages 719–720, New York, NY, USA. ACM. 137
- [Kang et al., 1998] Kang, K. C., Kim, S., Lee, J., Kim, K., Shin, E., and Huh, M. (1998). Form: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.*, 5:143–168. 156
- [Kassab and Ormandjieva, 2006] Kassab, M. and Ormandjieva, O. (2006). Towards an aspect oriented software development model with tractability mechanism. In *Proceedings of Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design. Bonn, Germany*. 43
- [Kaur and Singh, 2010] Kaur, K. and Singh, H. (2010). Candidate process models for component based software development. *ournal of Software Engineering*, 4(1):16–29. 21
- [Khoudour et al., 2009] Khoudour, L., Ghazel, M., Boukour, F., Heddebaut, M., and El-Kourssi, E.-M. (2009). Towards safer level crossings: existing recommendations, new applicable technologies and a proposed simulation model. *European Transport Research Review*, 1(1):35–45. 121, 122

- [Kim et al., 2012] Kim, S.-K., Myers, T., Wendland, M.-F., and Lindsay, P. A. (2012). Execution of natural language requirements using state machines synthesised from behavior trees. *J. Syst. Softw.*, 85(11):2652–2664. 51
- [Knethen and Paech, 2002] Knethen, A. v. and Paech, B. (2002). A survey on tracing approaches in practice and research. Technical report, IESE-Report, 095.01/E, Fraunhofer IESE - an institute of the Fraunhofer Gesellschaft., Boston, MA 02115, USA. 49
- [Knight, 2012] Knight, J. (2012). *Fundamentals of Dependable Computing for Software Engineers*. Chapman & Hall/CRC, 1st edition. 25, 27, 28, 29
- [Kolovos et al., 2006] Kolovos, D. S., Paige, R. F., and Polack, F. A. C. (2006). Merging models with the epsilon merging language (eml). In *MoDELS'06*, pages 215–229. 46, 50, 51, 52, 53, 54
- [Konigs et al., 2012] Konigs, S. F., Beier, G., Figge, A., and Stark, R. (2012). Traceability in systems engineering - review of industrial practices, state-of-the-art technologies and new research solutions. *Advanced Eng. Informatics*, 26(4):924–940. 46, 49, 50, 51, 55
- [Konrad and Cheng, 2005] Konrad, S. and Cheng, B. H. C. (2005). Real-time specification patterns. In *Proceedings of the 27th ICSE*, pages 372–381. 30, 31
- [Krichen and Tripakis, 2009] Krichen, M. and Tripakis, S. (2009). Conformance testing for real-time systems. *Form. Methods Syst. Des.*, 34(3):238–304. 84, 100, 102
- [Larsen et al., 2005] Larsen, K. G., Mikucionis, M., Nielsen, B., and Skou, A. (2005). Testing real-time embedded software using uppaal-tron: An industrial case study. In *Proceedings of the 5th ACM International Conference on Embedded Software*, pages 299–306. ACM. 100, 105
- [Larsen et al., 1997] Larsen, K. G., Pettersson, P., and Yi, W. (1997). Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152. 29, 100
- [Lau et al., 2011] Lau, K.-K., Taweel, F., and Tran, C. (2011). The w model for component-based software development. In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*, pages 47–50. 20
- [Lau and Wang, 2007] Lau, K.-K. and Wang, Z. (2007). Software component models. *Software Engineering, IEEE Transactions on*, 33(10):709–724. 16, 17, 18, 19, 34, 62
- [Lehman and Ramil, 2002] Lehman, M. and Ramil, J. (2002). Software uncertainty. In Bustard, D., Liu, W., and Sterritt, R., editors, *Soft-Ware 2002: Computing in an Imperfect World*, volume 2311 of *Lecture Notes in Computer Science*, pages 174–190. Springer Berlin Heidelberg. 55, 147
- [Lindvall and Sandahl, 1996] Lindvall, M. and Sandahl, K. (1996). Practical implications of traceability. *Softw. Pract. Exper.*, 26(10):1161–1180. 45

-
- [Liu et al., 2011] Liu, Y., Tang, T., Liu, J., Zhao, L., and Xu, T. (2011). Formal modeling and verification of RBC handover of ETCS using differential dynamic logic. In *Autonomous Decentralized Systems (ISADS), 2011 10th International Symposium on*, pages 67–72. 131
- [Liu et al., 2009] Liu, Z., Morisset, C., and Stolz, V. (2009). rCOS: theory and tool for component-based model driven development. In *Proceedings of the Third IPM international conference on Fundamentals of Software Engineering, FSEN'09*, pages 62–80, Berlin, Heidelberg. Springer-Verlag. 18
- [Lumpe et al., 2003] Lumpe, M., Schneider, J., Schönhage, B., Bauer, M., and Genssler, T. (2003). Composition languages. In *Object-Oriented Technology: ECOOP 2003 Workshop Reader, ECOOP 2003 Workshops, Darmstadt, Germany, July 21-25, 2003, Final Reports*, pages 107–118. 20
- [Mader and Egyed, 2012] Mader, P. and Egyed, A. (2012). Assessing the effect of requirements traceability for software maintenance. In *28th IEEE International Conference on Software Maintenance*, pages 171–180. 35, 55, 80, 113
- [Manna and Pnueli, 1992] Manna, Z. and Pnueli, A. (1992). *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA. 30
- [Manna and Pnueli, 1995] Manna, Z. and Pnueli, A. (1995). *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag New York, Inc., New York, NY, USA. 92
- [Marwedel, 2011] Marwedel, P. (2011). *Embedded Systems Design - Embedded Systems Foundations of Cyber-Physical Systems*. Springer, 2nd edition. ISBN 978-94-007-0256-1. 22
- [Mekki, 2012] Mekki, A. (2012). *Contribution for the Specification and the Verification of Temporal Requirements : Proposal of an extension for the ERTMS-Level 2 specifications*. Theses, Ecole Centrale de Lille. 103
- [Mekki et al., 2011] Mekki, A., Ghazel, M., and Toguyeni, A. (2011). Patterns-Based Assistance for Temporal Requirement Specification. In *Proceeding of SERP'2011*, pages 1–7, Las Vegas, États-Unis. 32
- [Mekki et al., 2012] Mekki, A., Ghazel, M., and Toguyeni, A. (2012). Validation of a new functional design of automatic protection systems at level crossings with model-checking techniques. *IEEE Trans. on ITS*, 13(2):714–723. 33, 84, 122, 123, 124
- [Merle and Stefani, 2008] Merle, P. and Stefani, J.-B. (2008). A formal specification of the Fractal component model in Alloy. Research Report RR-6721, INRIA. 36
- [Merlin and Farber, 1976] Merlin, P. and Farber, D. J. (1976). Recoverability of communication protocols—implications of a theoretical study. *Communications, IEEE Transactions on*, 24(9):1036–1043. 100
- [Meyer, 1997] Meyer, B. (1997). *Object-oriented Software Construction (2Nd Ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. 112

- [Myers and Sandler, 2004] Myers, G. J. and Sandler, C. (2004). *The Art of Software Testing*. John Wiley & Sons. 27
- [Mylopoulos et al., 1992] Mylopoulos, J., Chung, L., and Nixon, B. (1992). Representing and using nonfunctional requirements: a process-oriented approach. *Software Engineering, IEEE Transactions on*, 18(6):483–497. 45, 46
- [Narayanan and Karsai, 2008] Narayanan, A. and Karsai, G. (2008). Towards verifying model transformations. *Electron. Notes Theor. Comput. Sci.*, 211:191–200. 106
- [NIST, 2002] NIST (2002). The economic impacts of inadequate infrastructure for software testing. <http://www.nist.gov/director/planning/upload/report02-3.pdf>. 27
- [OMG, 2003] OMG (2003). Uml 2.0 OCL specification. <http://www.omg.org/cgi-bin/doc?ptc/2003-10-14>. 66, 112
- [OMG, 2005] OMG (2005). Omg unified modeling language specification. <http://www.omg.org/spec/UML/>. 66
- [OMG, 2011] OMG (2011). Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. <http://www.omg.org/spec/QVT/>. 72, 137
- [openETCS, 2012] openETCS (2012). Itea2 openetcs consortium. <http://openetcs.org>. 62, 156
- [Panunzio and Vardanega, 2010] Panunzio, M. and Vardanega, T. (2010). A component model for on-board software applications. In *36th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2010, Lille, France, September 1-3, 2010*, pages 57–64. 22, 34
- [Panunzio and Vardanega, 2013] Panunzio, M. and Vardanega, T. (2013). On software reference architectures and their application to the space domain. In Favaro, J. and Morisio, M., editors, *Safe and Secure Software Reuse*, volume 7925 of *Lecture Notes in Computer Science*, pages 144–159. Springer Berlin Heidelberg. 156
- [Panunzio and Vardanega, 2014] Panunzio, M. and Vardanega, T. (2014). A component-based process with separation of concerns for the development of embedded real-time software systems. *Journal of Systems and Software*, 96(0):105 – 121. 4, 18, 35, 36, 55, 62
- [Peleska, 2013] Peleska, J. (2013). Industrial-strength model-based testing - state of the art and current challenges. In *Proceedings Eighth Workshop on Model-Based Testing, MBT 2013, Rome, Italy, 17th March 2013.*, pages 3–28. 3
- [Pfister et al., 2012] Pfister, F., Chapurlat, V., Huchard, M., Nebut, C., and Wippler, J.-L. (2012). A proposed meta-model for formalizing systems engineering knowledge, based on functional architectural patterns. *Syst. Eng.*, 15(3):321–332. 66, 67

-
- [Pinheiro, 2004] Pinheiro, F. A. (2004). Requirements traceability. In do Prado Leite, J. and Doorn, J., editors, *Perspectives on Software Requirements*, volume 753 of *The Springer International Series in Engineering and Computer Science*, pages 91–113. Springer US. 25, 40, 41, 42, 44, 45, 46, 63, 64
- [Pop et al., 2014] Pop, T., Hnetyuka, P., Hosek, P., Malohlava, M., and Bureš, T. (2014). Comparison of component frameworks for real-time embedded systems. *Knowledge and Information Systems*, 40(1):127–170. 17, 33, 62
- [Qiu et al., 2014] Qiu, S., Sallak, M., SchÄ¶n, W., and Cherfi-Boulanger, Z. (2014). Availability assessment of railway signalling systems with uncertainty analysis using statecharts. *Simulation Modelling Practice and Theory*, 47(0):1 – 18. 147
- [Ramakrishna et al., 1996] Ramakrishna, Y., Melliar-Smith, P., Moser, L., Dillon, L., and Kutty, G. (1996). Interval logics and their decision procedures: Part i: An interval logic. *Theoretical Computer Science*, 166(1-2):1–47. 30
- [Ramesh and Jarke, 2001] Ramesh, B. and Jarke, M. (2001). Toward reference models for requirements traceability. *IEEE Trans. Softw. Eng.*, 27(1):58–93. 46, 50, 51, 52, 53, 54
- [Renault et al., 2009] Renault, X., Kordon, F., and Hugues, J. (2009). Adapting models to model checkers, a case study : Analysing aadl using time or colored petri nets. In *Rapid System Prototyping, 2009. IEEE/IFIP International Symposium on*, pages 26–33. 36
- [Riaz, 2012] Riaz, S. (2012). *Moving Towards Component Based Software Engineering in Train Control Applications*. Master thesis, Linkoping University Department of Computer and Information Science. 4, 139, 140
- [Roscoe et al., 1997] Roscoe, A. W., Hoare, C. A. R., and Bird, R. (1997). *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA. 98
- [Rothenberger and Nazareth, 2002] Rothenberger, M. A. and Nazareth, D. L. (2002). A cost-benefit-model for systematic software reuse. In *Proceedings of the 10th European Conference on Information Systems, Information Systems and the Future of the Digital Economy, ECIS 2002, Gdansk, Poland, June 6-8, 2002*, pages 371–378. 139
- [Rubinger and Burke, 2010] Rubinger, A. L. and Burke, B. (2010). *Enterprise JavaBeans 3.1 - Developing Enterprise Java Components: Covers JavanBeans 3.1 (6. ed.)*. O’Reilly. 18, 34
- [Rychly, 2011] Rychly, M. (2011). A metamodel for modelling of component-based systems with mobile architecture. In Pokorny, J., Repa, V., Richta, K., Wojtkowski, W., Linger, H., Barry, C., and Lang, M., editors, *Information Systems Development*, pages 635–646. Springer New York. 70
- [Sadani et al., 2005] Sadani, T., Courtiat, J.-P., and De Saqui-Sannes, P. (2005). From RT-LOTOS to time petri nets new foundations for a verification platform. In *Third IEEE Int. Conference on Software Engineering and Formal Methods*, pages 250–259. 32

- [Sango, 2013] Sango, M. (2013). Application of sara approach to ertms/etcs on-board train speed control software. Technical Report, IFSTTAR. <http://urls.fr/sara>. 145
- [Sango et al., 2014a] Sango, M., Duchien, L., and Gransart, C. (2014a). Component-based modeling and observer-based verification for railway safety-critical applications. In *11th International Symposium on Formal Aspects of Component Software*, pages 248–266, Bertinoro, Italy. 62, 72, 146
- [Sango et al., 2014b] Sango, M., Gransart, C., and Duchien, L. (2014b). Safety component-based approach and its application to ERTMS/ETCS on-board train control system. In *Transport Research Arena (TRA2014)*, pages 648–653, Paris, France. 62
- [Sango et al., 2015a] Sango, M., Hoinaru, O., Gransart, C., and Duchien, L. (2015a). A temporal qos ontology for ertms/etcs. In *ICKEOE 2015: International Conference on Knowledge Engineering and Ontological Engineering, London, United Kingdom, (Jan 19-20, 2015)*. 121
- [Sango et al., 2015b] Sango, M., Hoinaru, O., Gransart, C., and Duchien, L. (2015b). A temporal qos ontology for ertms/etcs. *International Journal of Computer, Control, Quantum and Information Engineering*, 9(1):95 – 101. 121
- [Santiago et al., 2012] Santiago, I., Jiménez, A., Vara, J. M., De Castro, V., Bollati, V. A., and Marcos, E. (2012). Model-driven engineering as a new landscape for traceability management: A systematic literature review. *Inf. Softw. Tech.*, 54(12):1340–1356. 21
- [SCADE, nd] SCADE (n.d.). Model-based development environment for critical embedded software. <http://www.esterel-technologies.com/products/scade-suite/>. 29
- [Schmidt, 2006] Schmidt, D. C. (2006). Model-driven engineering. *IEEE Computer*, 39(2):25–31. 4, 61
- [Schneider et al., 1992] Schneider, G. M., Martin, J., and Tsai, W. T. (1992). An experimental study of fault detection in user requirements documents. *ACM Trans. Softw. Eng. Methodol.*, 1(2):188–204. 26
- [Sentilles et al., 2008] Sentilles, S., Vulgarakis, A., Bures, T., Carlson, J., and Crnkovic, I. (2008). A component model for control-intensive distributed embedded systems. In *Component-Based Software Engineering, 11th International Symposium, CBSE 2008, Karlsruhe, Germany, October 14-17, 2008. Proceedings*, pages 310–317. 34
- [Simulink, nd] Simulink (n.d.). Simulation et model-based design. <http://fr.mathworks.com/products/simulink/>. 29
- [Soliman et al., 2012] Soliman, D., Thramboulidis, K., and Frey, G. (2012). Transformation of function block diagrams to uppaal timed automata for the verification of safety applications. *Annual Reviews in Control*, 36(2):338 – 345. 36, 84

-
- [Sousa and Garlan, 1999] Sousa, J. P. and Garlan, D. (1999). Formal modeling of the enterprise javabeans component integration framework. In Wing, J., Woodcock, J., and Davies, J., editors, *Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg. 36
- [SPARK, 2014] SPARK (2014). Expanding the boundaries of safe and secure programming. <http://www.spark-2014.org/about>. 28, 87, 100
- [SysML, 2012] SysML (2012). System modeling language specification, version 1.3, 2012. <http://www.omg.org/spec/SysML/1.3>. 109
- [Szyperski et al., 2002] Szyperski, C., Gruntz, D., and Murer, S. (2002). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition. 16, 62
- [Tansalarak and Claypool, 2005] Tansalarak, N. and Claypool, K. T. (2005). COCO: composition model and composition model implementation. In *ICEIS 2005, Proceedings of the Seventh International Conference on Enterprise Information Systems, Miami, USA, May 25-28, 2005*, pages 340–345. 20
- [Thomas and Barry, 2003] Thomas, D. and Barry, B. M. (2003). Model driven development: The case for domain oriented programming. In *Companion of the 18th Annual ACM SIGPLAN Conference on OOPSLA'03*, pages 2–7, New York, NY, USA. ACM. 21
- [Tomar and Gill, 2010] Tomar, P. and Gill, N. (2010). Verification and validation of components with new x component-based model. In *Software Technology and Engineering (ICSTE), 2010 2nd International Conference on*, volume 2, pages V2–365–V2–371. 20
- [Triskel project (IRISA), nd] Triskel project (IRISA) (nd). The Metamodeling Language Kernel. <http://www.kermeta.org/>. 54
- [UML2.0, 2005] UML2.0 (2005). Unified modeling language: Superstructure, version 2.0. <http://www.omg.org/cgi-bin/doc?formal/05-07-04>. 17, 19
- [van Amstel et al., 2011] van Amstel, M., Bosems, S., Kurtev, I., and Pires, L. F. (2011). Performance in model transformations: Experiments with ATL and QVT. In *Theory and Practice of Model Transformations - 4th International Conference, ICMT 2011, Zurich, Switzerland, June 27-28, 2011. Proceedings*, pages 198–212. 138
- [Vanhooff and Berbers, 2005] Vanhooff, B. and Berbers, Y. (2005). Supporting modular transformation units with precise transformation traceability metadata. In *ECMDA'06 Traceability Workshop*, pages 15–27. 46, 50, 51, 52, 53, 54
- [Varró, 2002] Varró, D. (2002). A formal semantics of uml statecharts by model transition systems. In *Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 378–392. Springer Berlin Heidelberg. 103

- [Varró and Pataricza, 2003] Varró, D. and Pataricza, A. (2003). Automated formal verification of model transformations. In *Critical Systems Development in UML*, page 63–78. 106
- [W3C, 2004] W3C (2004). Owl web ontology language overview. 121
- [Wen et al., 2007] Wen, L., Colvin, R., Lin, K., Seagrott, J., Yatapanage, N., and Dromey, R. G. (2007). Integrare, a collaborative environment for behavior-oriented design. In *4th International Conference on Cooperative Design, Visualization, and Engineering*, pages 122–131. 54
- [Wen and Dromey, 2004] Wen, L. and Dromey, R. (2004). From requirements change to design change: a formal path. In *Proceedings of the Second International Conference on Software Engineering and Formal Methods, 2004. SEFM 2004.*, pages 104–113. 46, 50, 51
- [Wen et al., 2014] Wen, L., Tuffley, D., and Dromey, R. G. (2014). Formalizing the transition from requirements’ change to design change using an evolutionary traceability model. *ISSE*, 10(3):181–202. 46, 49, 50, 51, 52, 53, 54
- [Wieringa, 1995] Wieringa, R. (1995). An introduction to requirements traceability. In *Faculty of Mathematics and Computer Science, Vrije Universiteit, Tech. Rep. IR-389*, page 24. 41, 47, 72
- [Winkler and Pilgrim, 2010] Winkler, S. and Pilgrim, J. (2010). A survey of traceability in requirements engineering and model-driven development. *Softw. Syst. Model.*, 9(4):529–565. 49
- [Woodcock and Morgan, 1990] Woodcock, J. and Morgan, C. (1990). Refinement of state-based concurrent systems. In Bjorner, D., Hoare, C., and Langmaack, H., editors, *VDM ’90 VDM and Z - Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 340–351. Springer Berlin Heidelberg. 98
- [Yang et al., 2014] Yang, K., Duan, Z., and Tian, C. (2014). Modeling and verification of RBC handover protocol. *Electronic Notes in Theoretical Computer Science*, 309:51 – 62. 131
- [Yie and Wagelaar, 2009] Yie, A. and Wagelaar, D. (2009). Advanced Traceability for ATL. In *Workshop on Model Transformation with ATL*, page 10. 72
- [Yovine, 1997] Yovine, S. (1997). A verification tool for real-time systems. *Springer International Journal of Software Tools for Technology Transfer*, 1(1-2):123–133. 29, 100
- [Yu et al., 2006] Yu, J., Manh, T., Han, J., Jin, Y., Han, Y., and Wang, J. (2006). Pattern based property specification and verification for service composition. In *Web Information Systems*, volume 4255 of *LNCS*, pages 156–168. Springer Berlin Heidelberg. 32
- [Zhao et al., 2002] Zhao, J., Yang, H., Xiang, L., and Xu, B. (2002). Change impact analysis to support architectural evolution. *Journal of Software Maintenance*, 14(5):317–333. 53