Architecture-Driven Integration of Modeling Languages for the Design of Software-Intensive Systems

Architecture-Driven Integration of Modeling Languages for the Design of Software-Intensive Systems

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universtiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.C.A.M. Luyben,
voorzitter van het College van Promoties,
in het openbaar te verdedigen op woensdag 10 maart 2010 om 12:30 uur
door

Michel dos Santos SOARES

Mestre em Ciência da Computação
Universidade Federal de Uberlândia, Uberlândia, Brazil
geboren te Rio de Janeiro, Brazil.

Dit proefschrift is goedgekeurd door de promotor:
Prof.dr.ir. A. Verbraeck

Copromotor: Dr. J.L.M. Vrancken

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
Prof.dr.ir. A. Verbraeck,	Technische Universiteit Delft, promotor
Dr. J.L.M. Vrancken,	Technische Universiteit Delft, copromotor
Prof.dr. Y-H. Tan,	Technische Universiteit Delft,
Prof.dr. F.M.T. Brazier,	Technische Universiteit Delft,
Prof.dr. G. Muller,	Buskerud University College,
Prof.dr. P. Klint,	University of Amsterdam,
Prof.dr. M. Boasson,	University of Amsterdam,
Reservelid:	
Prof.dr.ir. S.P. Hoogendoorn,	Technische Universiteit Delft

Published and distributed by:
Next Generation Infrastructures Foundation
P.O. Box 5015 2600 GA Delft The Netherlands
T: + 31 15 2782564
F: + 31 15 2782563
E-mail: info@nginfra.nl

This research was funded by the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. INFSO-ICT-223844, and by the Next Generation Infrastructures Foundation.

keywords: Software-Intensive Systems, Distributed Real-Time Systems, Software Architecture, Formal Methods, Road Traffic Management Systems, Requirements Engineering, Object-Oriented Modeling

ISBN 978-90-79787-24-1

Printed in the Netherlands by Gildeprint Drukkerijen BV, Enschede.
E-mail: mics.soares@gmail.com

Acknowledgements

The present thesis is an effort of almost 4 years of work. During this period I benefited from the support, guidance and encouragement of many people.

First of all I thank God for all support, forgiveness, and for letting me perform my own choices.

I am indebted to my co-promotor Jos Vrancken. During the period that this thesis was developed, Jos was both my daily supervisor and roommate. As a supervisor, I learned a lot, not only about Road Traffic Management Systems and Software Engineering, but also about doing research. I still remember him asking those "silly questions", for which I had no good answers, but helped me in finding solutions for my research problems. We also had the opportunity to discuss themes other than our research, such as economy and politics. I am very grateful for all the freedom in choosing my research topics that he provided during my research.

I was lucky to have Alexander Verbraeck as my promotor. He accepted to be my promotor despite I had already performed large parts of my research, and still manage to help me in creating a coherent thesis. I admire his way to supervise, not only pointing out problems but also always proposing solutions. Alexander, thank you for teaching me that I should ask more "why" questions when doing research.

I would like to thank the group of PhD students supervised by Jos. Yufei and Dijkstra, keep doing the good work. Mohsen, thank you for our talks and for always being in a good mood. Yubin, thank you for all the help during my internship at Trinité, and for the road pictures of Chapter 2. Now the token is with you guys.

I would like to thank all (former) members of the ICT Section at TBM, including Harry, Jan, Tineke, Marijn, Jolien, Mark, Jeffrey, Marnix, Bram, Sietse, Anne Fleur, Yiwei, Jaro, Sam, Luuk, Ralph, Andreas, Martijn, Potchara, Janneke, Virginia, Jo-Ann and Eveline. Nitesh, thank you for reading my initial chapters and for your feedback.

Many thanks to all members of my thesis committee for the good discussions

and for “opening my eyes” to problems in previous versions of the thesis.

I would like to thank all members of the NGI Intelligent Infrastructures subprogram, in special Zofia, Rudy, Koen, and Behzad.

The employees I met at Trinité provided an excellent environment during my internship. I could learn a lot about how Road Traffic Management Systems are designed and implemented in practice. I admire the way systems are developed at Trinité. Thank you Frank, Marcel Mossink, Marcel Valé, and Ying. And thanks to all employees who survived my course on Software Architecture and Modeling Languages and provided all the feedback I needed.

Thanks to all my friends from Brazil and Europe, including Elizângela, who was also my guide in my first steps in The Netherlands, Stéphane, my Master supervisor and co-author of one article included in this thesis, and Antonio, for the discussions about our projects. At TU Delft I had also the pleasure of working with Joseph Barjis, from the SK Section at TBM, during and after the supervision of Itamar Sharon.

To all my family a special gratitude. My parents, Ib and Elcia, for all support, my brother Marcelo and my sister Michele, all my uncles, in special Naldinho, aunts, and cousins, and my grandpa Arnaldo. I am also grateful for my parents-in-law, José Antônio and Elza for coming to my thesis defense. A vocês todos meus sinceros agradecimentos.

Finally, I can never thank enough my wife Taís. She gave up on her career to come with me to The Netherlands. Therefore, my dream became our dream. Taís, thank you for all the support, encouragement, and love.

Michel dos Santos Soares
Delft, February 2010.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Software Development as an Engineering Discipline

The importance of software in modern society is increasing because of the ubiquitous use of computers in virtually every human activity. Software helps in improving productivity and augmenting efficiency in manufacturing and agriculture. Banking and financial institutions rely heavily on software in their daily activities. Software is frequently used to control infrastructures systems such as electricity, gas, communication lines, road transportation and water networks, making modern life easier and more comfortable.

The development of software is a difficult, time consuming and complex activity, in which creativity and rigor have to be balanced. The complexity of developing, deploying and maintaining software is well-recognized and has been widely studied (Dijkstra, 1972; Boehm, 1973; Dijkstra, 1979; Brooks, 1987, 1995; Glass, 1999; Berry, 2004; Pressman, 2005; Sommerville, 2007; van Vliet, 2008). Software failures have been responsible for financial losses and disasters (Flowers, 1996; Bar-Yam, 2003; Charette, 2005). With the purpose of developing and maintaining software, a specific engineering discipline was proposed.

The term *Software Engineering* was probably introduced during a NATO Conference in 1968 (Buxton and Randell, 1970). The conference was a response to what was recognized as a crisis in software development due to high costs, late deliveries and poor product quality. The term was both a vision and a provocation. A vision because it was widely recognized by that time that the software crisis could only be solved with a disciplined engineering approach to developing software. A provocation because at that time the average software development practices were chaotic and could not be called an engineering disci-

pline. But why should software development be considered engineering (Parnas, 1997)? *Engineering* is about applying defined techniques, methods, processes, scientific and mathematical knowledge to the design, construction and implementation of artifacts such as buildings, bridges, planes, ships and machines (ABET, 1941). Measuring quality and other characteristics, such as usefulness and reliability, is possible and desirable not only for the engineering process but also for the final product. These notions are true for traditional engineering disciplines, such as Civil Engineering, and should also be true for Software Engineering (Pressman, 2005; Sommerville, 2007).

A number of principles of Software Engineering were proposed in the literature (Boehm, 1983; Parnas and Clements, 1986; Brooks, 1987; Bourque et al., 2002; Wang, 2007). These principles are useful to address the complexity of software development. Some of these principles relate directly to software design. According to (Parnas and Clements, 1986), the complexity of software development can be tackled by the divide and conquer strategy (Jackson, 2003), and techniques such as information hiding, functional decomposition, and modularization. The central idea of modularization is based on the basic assumption that software can be decomposed into smaller functional pieces during design. However, deciding how to divide a system into modules (subsystems, components) is itself a hard task (Parnas, 1972). Another fundamental principle to reduce complexity in software development is abstraction, which is concerned with eliciting essential properties of a set of objects while omitting unnecessary details of them (Booch et al., 2007). Models and diagrams are key elements to represent abstract designs of software.

Software Engineering has grown to become a discipline that is concerned with all aspects of software production, from the early stages of system specification to maintaining the system after it has gone into use (see SWEBOK in Chapter 2). After more than 40 years of creation of the term, there is no doubt that Software Engineering has made tremendous progress in many aspects. Programming languages created since then are more abstract and less machine-oriented, compilers are faster and more reliable, and methodologies are more systematic, just to cite a few achievements. Nevertheless, designing, implementing and deploying large, complex software-intensive systems remains an unresolved challenge (Broy, 2006; Tiako, 2008; Wirsing et al., 2008).

1.2 Software-Intensive Systems

Software is the general term used to describe not only the programs that are executed in hardware, but also configuration files and associated documentation such as requirements and design models (Sommerville, 2007). The collection of

software, configuration files, and the associated technical and user documentation is a software system.

Software-intensive systems (Wirsing and Holzl, 2006; Tiako, 2008; Hinchey et al., 2008) are large, complex systems in which software is an essential component, interacting with other software, systems, devices, actuators, sensors and with people. Being an essential component, software influences the design, construction, deployment, and evolution of the system as a whole (ANSI/IEEE, 2000). These systems are in widespread use and their impact on society is still increasing. Developments in engineering software-intensive systems have a large influence on the gains in productivity and prosperity that society has seen in recent years (Dedrick et al., 2003). Examples of software-intensive systems can be found in many sectors, such as manufacturing plants, transportation, military, telecommunication and health care.

More specifically, the type of software-intensive systems that are investigated in this thesis are the Distributed Real-Time Systems. The term Real-Time System usually refers to systems with timing constraints (Gomaa, 2000). Dijkstra (1965) recognized that some applications were concurrent in nature, in which several activities were logically occurring in parallel. In concurrent problems, there is no way of predicting which system component will provide the next input, which increases design complexity. Moreover, system components, such as sensors and actuators, are often geographically distributed in a network and need to communicate according to specific timing constraints described in requirements documents.

1.2.1 Development and Evolving Difficulties

The development of software has always been challenging (Wirth, 2008). Society's demands and improved hardware has vastly widened the area of computer applications (Boehm, 2006). As the focus of software switches from simple tasks and calculations automatization to modern control systems and systems-of-systems, the number and type of problems has risen tremendously. It is expected that future generations of software-intensive systems will become even more complex, highly distributed, and exhibiting adaptive behavior (Wirsing and Holzl, 2006).

The increase of hardware capacity in terms of performance and storage came together with a decrease in prices. As a result, the main cost of a system switched from hardware to software (Boehm, 1973). One clear example is related to the difficulties of discovering the problems to be solved and the functionalities to be implemented in the future system. Any errors introduced during these initial phases, in which the requirements of the stakeholders are gathered, documented

and analyzed, will have a high impact on the final cost of the system (Boehm and Papaccio, 1988). There is no doubt that measures were taken by researchers and practitioners in order to understand and decrease development costs, such as metrics for software (Boehm et al., 2000). However, software development is still costly and risky (Sommerville, 2007).

A great challenge in modern society is to develop successful software-intensive systems respecting constraints such as costs and deadlines, and being able to maintain and evolve these systems (Broy, 2006). This challenge is associated with another important one: developing practically useful and theoretically well-founded principles, methods, algorithms and tools for programming and engineering reliable, secure, cost-effective, and efficient software-intensive systems throughout their whole life-cycle (Wirsing and Holzl, 2006).

The proper environment in which software-intensive systems act poses great challenges. Software-intensive systems are frequently used to control critical infrastructures in which any error, non-conformance or even response delays may cause enormous financial damage or even jeopardize human life (Vrancken et al., 2008). These systems must provide high reliability. Depending on the system, having a correct response but with a small delay can cause a disaster (Johnson, 2003). As a matter of fact, in many cases the system must provide correct responses respecting a strict interval.

An interesting fact that can not be overlooked is that systems have to evolve in order to still be considered useful (Glass, 2006). In fact, software is hardly developed from scratch. Green field projects, in which software systems are developed without any constraints imposed by prior work, are rare (Boehm and Turner, 2003). A combination of existing systems is common, as is the adaptation in order to add new functionality or change the system to a new hardware architecture. Thus, systems must be flexible in order to facilitate their change. Estimates show that more than 75% of software development personnel in the United States alone work most of the time with maintenance (fixing errors and dealing with mass updates to aging legacy applications) (Jones, 2006). Most of the time, legacy systems can not simply be turned-off. These legacy systems are common in domains such as defense, transportation, air traffic and banking, and may keep existing for decades (Sommerville, 2007). They are maintained because it is too risky and too costly to replace them.

1.3 A Motivating Case

Road transportation is an important economic force that faces many problems. Traffic congestion, environmental pollution and safety are becoming increasingly unaccepted by society. The introduction of new infrastructure is impor-

tant but not sufficient (McQueen and McQueen, 1999; PIARC, 1999). Traffic demand is increasing, while constructing new road infrastructure is limited due to environmental, social and financial constraints (McDonald et al., 2006). In order to cope with these challenges, a possible solution is to manage and to control road traffic by developing Road Traffic Management Systems (RTMS) (see Chapter 3). RTMS make use of real-time data acquired from the road network in order to reduce traffic congestion and accidents, and to save energy and preserve the environment (PIARC, 1999). RTMS are Distributed Real-Time Systems, which means that all characteristics and difficulties previously discussed are valid (Vrancken and Soares, 2010). For instance, the complexity of the integration of information systems in transportation systems requires a formal statement of the structure of the system, the subsystems, and their interfaces.

The HARS project was the main motivating case for this thesis. HARS (*Het Alkmaar Regelsysteem*, the Alkmaar Control System in Dutch) is a RTMS installed on the belt road of the Dutch city of Alkmaar. HARS operation is challenging in the sense that it uses a large system to control complex behavior (road traffic). The controlled surface includes dozens of actuators and sensors which are distributed in the network. The development of HARS took 4400 man hours, and further maintenance developments took 3500 man hours, according to the company that developed HARS. It is important to note that actually many software components were reused. Therefore, developing such system from scratch would take much more time.

From the Software Engineering point of view, the development and maintenance of HARS was also a challenge due to many factors. The most important ones are mentioned as follows. First, requirements were frequently changed. The major reason is that the area of road network control is still largely uncharted territory (Vrancken and Soares, 2010). Thus, algorithms, techniques and methods are frequently being developed by traffic engineers. In addition, policies for transportation are often being changed as well (Eurostat, 2006). Second, legacy systems such as traffic signals control systems had to be integrated into the HARS system. These systems were not designed for network-level control and, due to their legacy properties, are hard to adapt to this new way of using them (Vrancken and Soares, 2009a). Third, the physical network to be controlled is constantly changing, with the addition of new sensors, actuators and even extensions of the road network. Whenever the physical network changes, or a physical component of the system (for instance, a sensor such as an inductive loop) is damaged and need to be substituted, the system has to be adapted and reconfigured, which means that flexibility is a concern. And fourth, the sensors and actuators of the system need to communicate with the control system by receiving and sending commands. The major difficulty is to cope with real-time constraints, as these elements are physically distributed in

the network.

HARS was installed in 2006 and is currently in operation. A new version has been developed and is planned to be deployed in the near future. The new system will have to be maintained for many years to come, and will have to cooperate with new systems, the same way the current system cooperates with many legacy systems. This case will return in Chapters 4, 5 and 6.

1.4 The Importance of Modeling

Modeling is fundamental for Software Engineering (Ludewig, 2003; Bézivin, 2005; Booch et al., 2005, 2007). There is a gap between the needs and constraints of a software-intensive system, which usually are expressed in natural language by the stakeholders, and the specifications needed to actually build software. Modeling can fill this gap, improving communication between teams and significantly diminishing natural language ambiguities.

Models are abstractions of physical systems that allow one to reason and understand the system by ignoring irrelevant details while focusing on the relevant ones (Booch et al., 2005; Brown et al., 2005). This simplification (or abstraction) is the essence of modeling (Booch et al., 2007). Models are used in many activities, such as to predict system behavior, as technical specifications and to communicate design decisions to various stakeholders. Although source code can also be considered a model, as it abstracts lower-level machine instructions, in practice source code and models are often considered to be different types of artifacts. Typically, in Systems and Software Engineering, an artifact is considered to be a model if it has a graphical, formal or mathematical representation instead of only a textual one as in the case of source code (Bézivin, 2006). This comes as no surprise, as UML (see Section 2.3) and its profiles are the current dominant graphical modeling languages.

Design for software-intensive systems requires adequate methodology in order to support the development of these systems (Tiako, 2008). In addition, there is a need to increase the level of abstraction, hiding whenever possible unnecessary complexity by the intense use of models (Booch et al., 2007). However, the theory of modeling for software-intensive systems remains incomplete, and methodologies for specifying and verifying software-intensive systems pose a grand challenge that a broad stream of research must address (Broy, 2006).

1.5 Research Objective and Questions

In order to develop software-intensive systems, modeling tasks have to cover different development phases such as requirements analysis, architectural design, and detailed design. Other phases, such as implementation, testing and integration are direct consequences of the modeling phases (Wirsing and Holzl, 2006). From the case previously mentioned and literature review, the main research objective was proposed as follows.

Improve Systems and Software Engineering methodology (*what*), by using, adapting, extending and combining modeling languages (*how*), to be used by systems and software developers (*to whom*), for designing software-intensive systems (*for which purpose*) that are in line with requirements, flexible, and reliable (*with which quality factors*).

In order to achieve this objective, three main research questions (RQ), each one with its focus on one core phase of Software Engineering methodologies (Requirements, Architecture and Design - see Chapter 2) were formulated:

RQ1 - How to improve requirements specification and analysis for Software-Intensive Systems?

This question is mainly answered in Chapter 4, in which, through the early introduction of graphical models, requirements are documented and analyzed. The identification and graphical representation of requirements relationships facilitate that traces are made. This helps in uncovering the impact that changes in requirements have in the system design. Requirements are important to determine the architecture (high-level design). When designing the architecture, at least part of the functional requirements should be known. In addition, the non-functional requirements that the architecture has to conform with should be made explicit.

RQ2 - How to specify a Software-Intensive System's Architecture that enables reusability?

This question is answered by the application of domain and software architectures, respectively topics of Chapters 5 and 6 of this thesis. These architectures have to conform with and be based on requirements and constraints. Another important characteristic of the proposed software architecture is that it is the basis for the design of a line of related software products. Thus, it must be able to accommodate related software-intensive systems.

RQ3 - How to model and verify reliability of Software-Intensive Systems?

This question is mainly answered in Chapter 7 by using formal methods. The basic approach is to create formal models that are further verified, checking whether they exhibit a set of desirable properties. For instance, whether unsafe states are reached due to an incorrect design can be detected.

1.6 Research Approach

In order to fulfill the research objective and answer the research questions, the research approach used is described in this section.

1.6.1 Research Philosophy

Four dominant research philosophies can be characterized (Creswell, 2008):

Positivism states that all knowledge must be based on logical inference from a set of basic observable facts. Positivists are reductionists, in that they study things by breaking them into simpler components. This corresponds to their belief that scientific knowledge is built up incrementally from verifiable observations and inferences based on them. Positivists prefer methods that start with precise theories from which verifiable hypotheses can be extracted, and tested in isolation. Hence, positivism is most closely associated with the controlled experiment; however, survey research and case studies are also frequently conducted with a positivist paradigm.

Interpretivism rejects the idea that scientific knowledge can be separated from its human context (Klein and Myers, 1999). Interpretivists concentrate less on verifying theories, and more on understanding how different people make sense of the world, and how they assign meaning to actions. Interpretivists prefer methods that collect rich qualitative data about human activities, from which theories might emerge. Interpretivism is most closely associated with ethnographies, although interpretivists often use exploratory case studies and survey research too (Easterbrook et al., 2007).

Critical Theory judges scientific knowledge by its ability to free people from restrictive systems of thought (Calhoun, 1995). Critical theorists prefer participatory approaches in which the groups they are trying to help are engaged in the research, including helping to set its goals. In Software Engineering, it includes research that actively seeks to challenge existing perceptions about software practice, most notably the open source movement, the process improvement community and the agile community (Easterbrook et al., 2007). Critical theorists often use case studies

to draw attention to things that need change. However, it is Action Research (Baskerville, 1999) that most closely reflects the philosophy of critical theorists.

Pragmatism acknowledges that all knowledge is approximate and incomplete, and its value depends on the methods by which it was obtained (Menand, 1997). Pragmatism is less dogmatic than the other three paradigms described above, as pragmatists tend to think the researcher should be free to use whatever research methods will help with the research problem. In essence, pragmatism adopts an engineering approach to research. It values practical knowledge over abstract knowledge, and uses whatever methods are appropriate to obtain it. Pragmatists use any available methods, and strongly prefer mixed methods research where several methods are used.

This thesis is based on an engineering approach to research, in the sense that it makes use of techniques and methods to create artifacts such as designs for software-intensive systems. The motivating case is a large distributed real-time system that can only be studied by reasoning on its parts. Thus, reducing the system into smaller, more comprehensible components, is of fundamental importance. Because of this, positivism, with its reductionism property, is a feasible choice for research philosophy. Another important aspect of this research is that it opens discussions for different people, who have different opinions, about the added value of Software Engineering methods and languages. Part of this research was performed in an environment in which the users could manifest their opinions, which makes the social context important as well. As a result, interpretivism also has a minor role in this research.

1.6.2 Research Strategy

Two paradigms characterize much of the research in Information Systems: behavioral science and design science (Hevner et al., 2004). In terms of objective, the behavioral science paradigm seeks to find "what is true", while the design science paradigm seeks to create "what is effective".

The behavioral science paradigm seeks to develop and verify theories that explain or predict human or organizational behavior. It has its roots in natural science research methods. The main objectives are to develop and justify theories that explain or predict organizational and human phenomena surrounding the analysis, design, implementation, management, and use of information systems.

The design science paradigm, which is followed in this thesis, has its roots in engineering (Tsichritzis, 1997; Denning, 1997). As a matter of fact, it is

fundamentally a problem solving paradigm with emphasis on products and solutions (Rossi and Sein, 2003).

There is lack of consensus as to the precise desired outputs of design research. In many cases, a solution such as a Software Engineering model is evaluated as "good enough", or in terms of its acceptance, and not in terms of "the best" solution. The resulting products, according to (Hevner et al., 2004), include 1) constructs or concepts, i.e., abstractions and representations that define the terms used when describing an artifact; 2) models, i.e., abstractions and representations, that are used to describe and represent the relationship among concepts; 3) methods, i.e., sets of steps and practices that are used to represent algorithms, processes or approaches on how to perform a certain task; and 4) instantiations, i.e., implemented or prototyped systems, that are used to realize the artifact. A fifth product, better theory, is included by (Rossi and Sein, 2003), as the practice may help in better understanding relationships, concepts and processes.

1.6.3 Research Instruments

Research instruments are used to collect and later analyze data in order to create models, methods or theories of a research strategy. Which research instrument to use depends on many factors such as the type of research questions, the research objective, and the amount of existing theories available. A set of research methods was applied during this research. Each one is briefly described as follows.

Carrying out a literature review research phase is strongly recommended to gain background knowledge of a subject as well as providing useful leads that will help to get the maximum from a research (Sjoberg et al., 2007). Initially an extensive literature review was performed, in which initial data was gathered by evaluating publications. After extensive literature research on problems and possible solutions for designing and analyzing software-intensive systems, the next step was to gain a deeper understanding of the domains (Software Engineering and Traffic Engineering) using case studies as research instrument.

A case study is defined as "an empirical inquiry that investigates a contemporary phenomenon within its real-life context, especially when the boundaries between phenomenon and context are not clearly evident" (Yin, 2003). Case study research can be characterized as qualitative and observatory (Yin, 2003). Rather than using samples, case study methods involve an in-depth examination of a single instance or event. When selected with care, even a single case study may be successful in terms of theory formulation and testing (Yin, 2003). This might be because it is a critical case for testing a well-formulated theory,

and if the theory holds for this case, it is likely to be true for many others. Case study research is most appropriate for cases where the reductionism of controlled experiments is inappropriate (Easterbrook et al., 2007). This includes situations where effects are expected to be wide ranging, or take a long time (e.g. weeks, months, years) to appear, which is the case in this research.

Case studies are a preferred research method when "how" and "why' questions are being investigated. The "how to" type of question is better approached with Action Research (AR) (Baskerville and Wood-Harper, 1996; Sjoberg et al., 2007). In order to make academic research relevant, researches should try out their theories with practitioners in real situations and real organizations (Avison et al., 1999). From a social organization viewpoint, the study of a newly invented technique/method is impossible without intervening in some way to inject the new approach into the practitioner's environment (Baskerville, 1999). AR can be described as a technique characterized by intervention experiments that operate on problems or questions perceived by practitioners within a particular context. The research is performed in the context of focused efforts to improve the quality of an organization. AR encourages researchers to experiment through intervention and to reflect on the effects of their intervention and the implication of their theories.

The ideal domain for AR application is satisfied by three conditions: 1) the researcher is actively involved, with expected benefit for both researcher and organization; 2) the knowledge obtained can be immediately applied, there is not the sense of the detached observer, but of an active participant, wishing to utilize any new knowledge based on an explicit, clear, conceptual framework; and 3) the research is a process linking theory and practice. These conditions were fulfilled during the performed research.

AR is an important method used in the research that led to this thesis. One important reason is related to the domain of the research. One clear area in the ideal domain of AR is new or changed software development methodologies. Studying new or changed methodologies implicitly involves the introduction of such changes, and is necessarily interventionist. AR is one of the few valid research approaches that we can legitimately employ to study the effects of specific alterations in software development methodologies in organizations. As part of the research was performed in a real organization, AR is a feasible research instrument.

The actual intervention must be evaluated. The methods chosen in this research for evaluation were surveys and interviews. Survey research is used to identify the characteristics of a broad population of individuals (Easterbrook et al., 2007). The defining characteristic of survey research is the selection of a representative sample from a well-defined population, and the data analysis techniques used to generalize from that sample to the population, usually to

answer base-rate questions. A hard challenge is to ensure that the questions of the survey are designed in a way that yields useful and valid data. It can be difficult to phrase the questions such that all participants understand them in the same way. Also, it is possible that what people say they do in response to survey questions bears no relationship to what they actually do (Easterbrook et al., 2007).

The results of the survey and interviews are shown in Chapter 8.

1.7 Thesis Outline

This thesis contributes to Software Engineering research and practice by proposing the extension and integration of formal and semi-formal modeling languages in a multiple-view software architecture, combined with domain architecture, which are used in practice to develop a family of systems in the Road Traffic Management Systems domain.

The thesis outline is depicted in Fig. 1.1. Chapter 2 limits the scope in terms of which areas of Software Engineering are investigated. Together with Chapter 3, which is about Intelligent Transportation Systems definition, approaches and examples, these chapters give a theoretical background for the remainder of the thesis. Chapter 4 covers Requirements Engineering, and Chapters 5 and 6 are about Architecture. The first deals with Domain Architecture, and the later with Software Architecture.

It is important to understand that Chapters 4, 5 and 6 are fully dependent on each other. The reason has to do with how software development occurs in practice. In reality, one needs to know about the requirements in order to establish an architecture. However, to design an architecture, one needs to have at least some ideas of the system to be built, i.e., at least part of the requirements must be known in advance. Chapters 4, 5 and 6 are important inputs for Chapter 7, about formal design and verification of system components and software objects.

Although the thesis follows a structure inspired by the Waterfall model for Software Engineering, this is just a formality. Modern software development processes are iterative instead of linear, which is exactly the approach that the reader of this thesis must have in mind. Chapter 8 is about evaluation in practice, and Chapter 9 is about conclusions, lessons learned, limitations and future research.

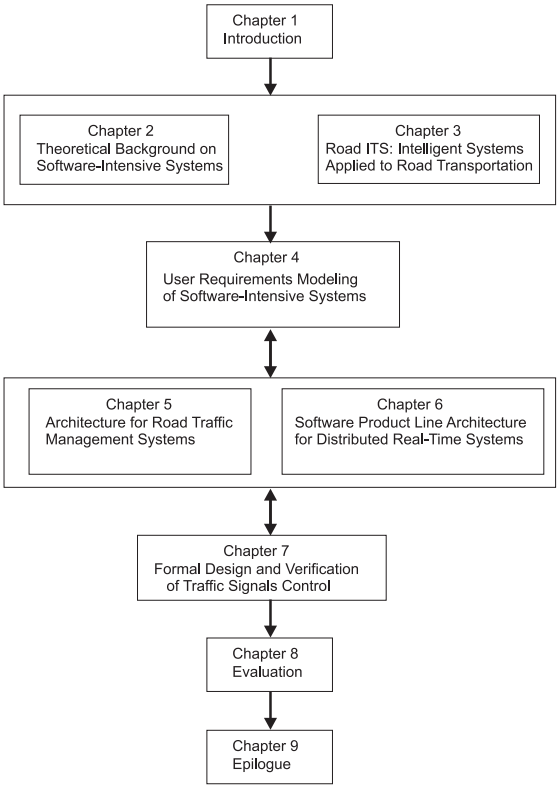

Figure 1.1: Thesis outline

1.8 Origins of the Chapters

The origin of the other chapters is as follows.

Chapters 2 and 3 are both about theoretical background, written after extensive literature research on, respectively, Software-Intensive Systems and Intelligent Transportation Systems. These chapters are also based on the following papers:

- (Soares and Vrancken, 2007b)
- (Soares and Vrancken, 2008a)
- (Vrancken et al., 2008)

Chapter 4

- (Soares and Vrancken, 2007c)

- (Soares and Vrancken, 2008b)
- (Soares and Vrancken, 2008c)

Chapter 5

- (Soares and Vrancken, 2007a)
- (Vrancken and Soares, 2009b)
- (Vrancken and Soares, 2010)
- (Soares et al., 2010)

Chapter 6

- (Soares et al., 2009b)
- (Soares and Vrancken, 2009b)
- (Soares et al., 2009a)
- (Soares et al., 2009c)

Chapter 7

- (Soares and Vrancken, 2007d)
- (Soares and Vrancken, 2007e)
- (Soares and Vrancken, 2008d)
- (Soares et al., 2008)

Chapter 8

- (Soares and Vrancken, 2009a)

Chapter 2

Theoretical Background on Software-Intensive Systems

This chapter relates the scope of this thesis with the knowledge areas of the SWEBOK (Section 2.1). Then, a classification of modeling languages and methods is given in Section 2.2. UML and SysML are briefly introduced in Section 2.3. The chapter ends with approaches in which modeling languages and methods are integrated (Section 2.4).

2.1 Thesis Scope

The main focus of this thesis is on the modeling phases, as already discussed in Chapter 1 and further explained in this section. The SWEBOK (Software Engineering Body of Knowledge) (Abran et al., 2004) is a document created by software research experts and practitioners with the purpose of establishing a baseline for the body of knowledge for the field of Software Engineering. The document is considered as a guide that contributes to a consensually validated characterization of the bounds of the Software Engineering discipline and provides a topical access to the body of knowledge supporting that discipline. The SWEBOK is subdivided into ten Software Engineering Knowledge Areas (KA) plus an additional chapter providing an overview of the KAs of strongly related disciplines. Each KA is subdivided into subareas, topics and sub-topics. Figs. 2.1 and 2.2 depict the current version of the SWEBOK structure with the KAs and the subareas.

The main KAs covered in this thesis are: Software Requirements, Software Design, and Software Engineering Tools and Methods.

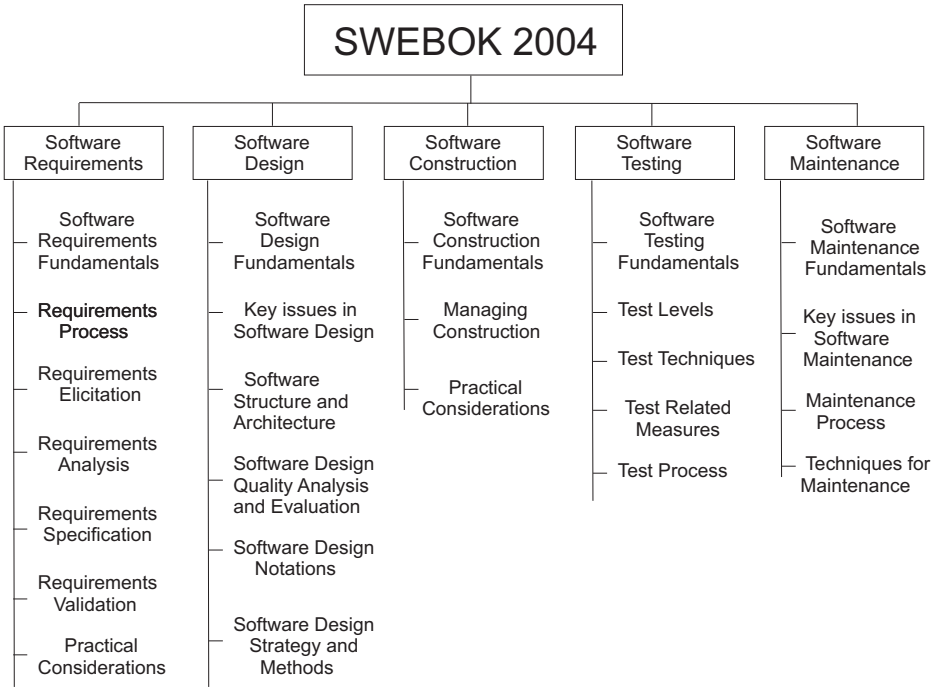

Figure 2.1: SWEBOK Knowledge Areas - Part 1

In the KA Software Requirements, the covered subareas are:

- Software Requirements Fundamentals;
- Requirements Process;
- Requirements Analysis;
- Requirements Specification.

In the KA Software Design, the covered subareas are:

- Software Design Fundamentals;
- Software Structure and Architecture;
- Software Design Notations;
- Software Design Strategies and Methods.

In the KA Software Engineering Tools and Methods, the subarea is:

- Software Engineering Methods;

For each of the main KAs, a brief definition of the area according to the SWEBOK document and the main reasons why the area was selected are given in the next subsections.

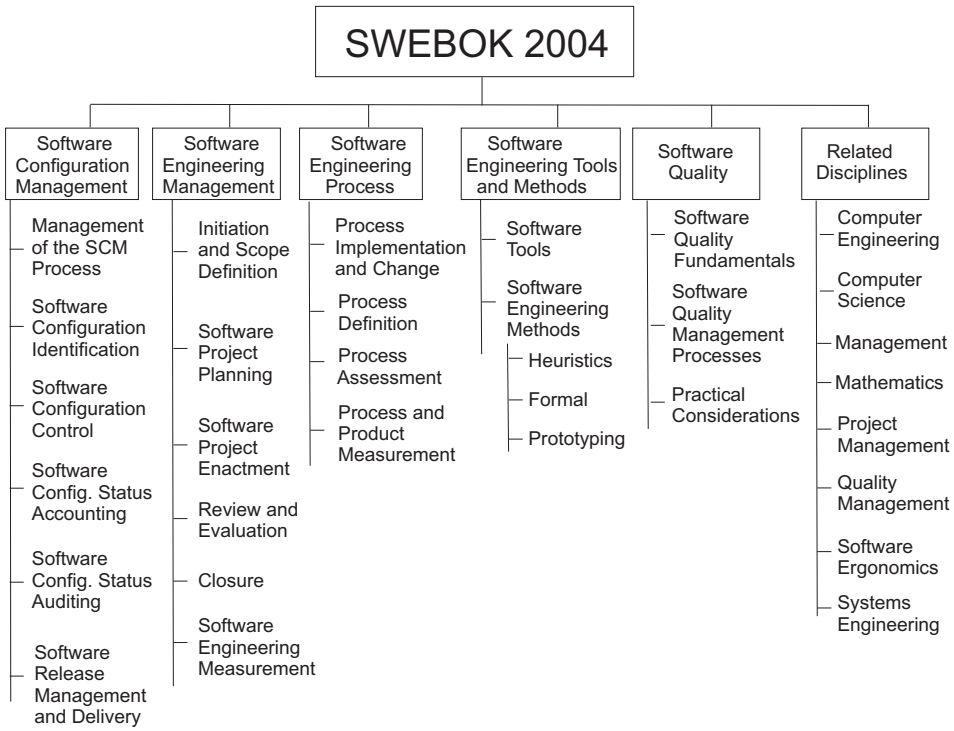

Figure 2.2: SWEBOK Knowledge Areas - Part 2

2.1.1 Software Requirements

The Software Requirements Knowledge Area (called *Requirements Engineering* in this thesis) is concerned with the elicitation, analysis, specification, and validation of software requirements. When any of these activities are poorly performed, software projects are critically vulnerable and have more chances to fail (Abran et al., 2004).

Requirements Engineering is a very influential phase in the life cycle. According to the SWEBOK, it concerns Software Design, Software Testing, Software Maintenance, Software Configuration Management, Software Engineering Management, Software Engineering Process, and Software Quality KAs.

Requirements Engineering is generally considered in the literature as the most

critical phase within the development of software (Juristo et al., 2002; Komi-Sirviö and Tihinen, 2003; Damian et al., 2004; Minor and Armarego, 2005). Dealing with ever-changing requirements is considered the real problem of Software Engineering (Berry, 2004). Already in 1973, Boehm suggested that errors in requirements could be up to 100 times more expensive to fix than errors introduced during implementation (Boehm, 1973). According to Brooks (Brooks, 1987), knowing what to build, which includes requirements elicitation and technical specification, is the most difficult phase in the design of software. Lutz (Lutz, 1993) showed that 60% of errors in critical systems were the results of requirements errors. Studies conducted by the Standish Group (The Standish Group, 2003) and other researchers (van Genuchten, 1991; Hofmann and Lehner, 2001) found that the main factors for problems with software projects (cost overruns, delays, user dissatisfaction) are related to requirements issues, such as lack of user input, incomplete requirements specifications, uncontrolled requirements changing, and unclear objectives. In an empirical study with 12 companies (Hall et al., 2002), it was discovered that, out of a total of 268 development problems cited, 48% (128) were requirements problems. In another empirical study (Luisa et al., 2004), the activities of identify user requirements and later model these requirements were considered as priorities by a large percentage of respondents.

2.1.2 Software Design

Software design is the activity of the Software Engineering life cycle in which requirements are analyzed in order to produce a description of the software's internal structure that will serve as the basis for its construction (Abran et al., 2004). The result of this activity is the software architecture, the interfaces between components, and also the components at a level of detail that enable their construction.

Software design plays an important role in developing software: it allows software engineers to produce various models that form a kind of blueprint of the solution to be implemented. These models can be analyzed and evaluated to determine whether or not they will fulfill the various requirements. The result of this activity can be used to plan the subsequent development activities, in addition to using them as input and the starting point of construction and testing.

According to (IEEE/EIA, 1996), software design consists of two activities that fit between requirements engineering and software construction:

1. Software architectural design (also known as high-level design): describing software's top-level structure and organization and identifying the various

components;

2. Software detailed design: describing each component sufficiently to make its construction possible.

The main focus of Chapter 5 is on architectural design for the domain, i.e., architectures for Road Traffic Management Systems, and the focus of Chapter 6 is on a software product line architecture for implementing Road Traffic Management Systems. Detailed design using formal methods is the topic of Chapter 7.

Basically, architecture refers to the organization of the system, such as its components, subsystems, interfaces, and how these elements collaborate and are composed to form the system (Garlan and Shaw, 1993; ANSI/IEEE, 2000). Only relevant decisions are important at this level, i.e., those that have a high impact on cost, reliability, maintainability, performance and resilience of the future system.

A proven concept to build software-intensive systems and to avoid problems and failure causes, such as poor communication among stakeholders and sloppy and immature development practices, is to have a well-defined software architecture (Bass et al., 2003). Already in 1969 software architecture was recognized as fundamental for large-scale software, positioned at a higher level than design (Buxton and Randell, 1970). According to (Booch, 2007), having an architecture allows the development of systems that are better and more resilient to change when compared to systems developed without a clear architectural definition. Software architecture is also considered one of the most significant technical factors in ensuring project success (Brown and McDermid, 2007). The software architecture affects the performance, robustness, and maintainability of a system (Bosch, 2000). The architecture style and structure may both depend on and influence these requirements, which are fundamental for software-intensive systems.

2.1.3 Software Engineering Tools and Methods

The Software Engineering Tools and Methods KA covers the complete life cycle processes, and is therefore related to every KA in the SWEBOK. The focus of Chapter 7 of this thesis is on the Software Engineering Methods subarea, which goal is to make Software Engineering activities systematic and ultimately more likely to be successful. Methods vary widely in scope, from a single life cycle phase to the complete life cycle. The SWEBOK classifies the Software Engineering Methods subarea in three topics: heuristic, formal, and prototyping methods. Nevertheless, it is clear in the document that these three topics are

not disjoint, but represent distinct concerns. One example is Z++ (Lano, 1991), which is both formal and object-oriented.

In this thesis, the covered topics of the Software Engineering Methods subarea are *heuristic methods*, dealing with semi-formal approaches, such as object-oriented methods, and *formal methods*, dealing with mathematics based approaches. These topics are applied in the activity of detailed design in this thesis (Chapter 7). Object-oriented methods (notation, specification, refinement) are also used in Chapter 6. The following section describes the classification for Software Engineering Methods used in this thesis.

2.2 Classification of Software and Systems Engineering languages and methods

In order to support the specification of software systems, modeling languages and methods were created based on many paradigms. The classification proposed is the one relevant for this thesis, in which modeling languages and methods are categorized as formal or object-oriented.

2.2.1 Formal Methods

The term formal method (Wing, 1990; Saiedian, 1996; Sommerville, 2007) is used in literature to refer to any language or method that relies on a mathematical theory, such as algebra, logic or set theory. A formal specification is expressed in a language whose vocabulary, syntax and semantics are formally defined. This essential characteristic allows specifications expressed in such a language to be proven with mathematical rigor. Examples of well-known formal specification languages are LOTOS, Z, CSP and Petri nets.

When designing software-intensive systems that are going to be used in a critical environment, such as energy or transportation networks, emergent properties including safety and reliability are fundamental. Discovering possible design flaws in the specifications, such as a deadlock, is highly desirable.

The applicability and suitability of formal methods in practice is highly debated in the Software and Systems Engineering community (Bowen and Hinchey, 1995; Larsen et al., 1996; Abrial, 2007; Woodcock et al., 2009). In economic terms, the cost/benefit ratio is quickly becoming affordable (Davis, 2005). Some research studies were published arguing that there are so many advantages in using formal methods that it is worth trying (Larsen et al., 1996; Davis, 2005; Bowen and Hinchey, 2005). Dependability (the property that reliance can justifiably be placed on the service it delivers) and reliability (the measure of the

ability of system to continue operating over time) in software-intensive systems can be achieved only through the application of solid design principles (Hinchey et al., 2008), which are more likely to happen when formal methods are applied. It must be considered the fact that it costs too much in downtime and maintenance not to formally prove the correctness of software-intensive systems (Davis, 2005). Formal methods contribute to demonstrably cost-effective development of software with very low defect rates (Broadfoot and Hopcroft, 2005). Some successful cases in industry were published in (Clarke and Wing, 1996; Abrial, 2006; Vrancken et al., 2008).

Despite the many advantages, formal methods also present drawbacks and by no means can be considered Software Engineering silver bullets' (Brooks, 1987; Davis, 2005). Practitioners often consider formal methods inadequate, restricted to critical systems, too expensive, insufficient, too difficult and "not at all practical" (Broy, 2007). Real-world examples are still lacking or the application scope is generally considered limited (Wassyng and Lawford, 2003). Although formal methods can strongly reduce the errors that can occur in the translation from requirements to executable source code, they cannot make sure that the initial document does not contain errors, inconsistencies, serious omissions or features that later on turn out to be undesirable (Vrancken et al., 2008).

The difficulty of learning formal methods is also highly debated. Hall (Hall, 1990) considers that the mathematical theory for formal methods is affordable for all engineers and the training is not difficult. For Abrial (Abrial, 2006), experience has shown that engineers can easily learn the mathematical concepts and notations used in formal methods. On the other hand, according to (Luqi and Goguen, 1997), formal methods are difficult to most practicing programmers, who have little training or skill in advanced mathematics.

2.2.2 Object-Oriented Methods

The object-oriented paradigm was first proposed for programming languages. Simula is considered the first object-oriented programming language. Other programming languages were created from scratch with object-oriented features, such as Smalltalk and Eiffel, and in others these features were included in an existing language to create another language, such as C++ and Objective C, that are derived from the C programming language.

The proliferation and success of object-oriented programming languages bring the idea to use these languages with specifications also created based on the object-oriented paradigm (Booch et al., 2005). Thus, a variety of methodologies and modeling languages were created. This led to many problems. Designers

had trouble finding a modeling language that could fulfill their needs completely (Booch et al., 2005). The effort to learn and introduce a methodology, buy and integrate tools and notations, could become useless if the methodology did not succeed. This period was informally known in Software Engineering as the "method wars" (Booch et al., 2005). Each methodology creator advocated that his own methodology was better in terms of expressivity, ability to capture all important aspects of the system design, or natural implementation in an object-oriented programming environment. Three of the most successful creators decided to unify their methodologies (Booch, OMT and OOSE), together with Statecharts (Harel, 1987), in order to create not another methodology, but a modeling language. The effort officially started in October 1994 and the first release of UML (Unified Modeling Language) was offered for standardization to the OMG in January 1997 (more about the history of UML in (Booch et al., 2005)).

2.3 UML and its Profiles

UML (OMG, 2007) is currently the *de facto* standard object-oriented modeling language in the software industry. Its relative success led to the creation of UML profiles for more specific domains.

2.3.1 UML

UML is a modeling language used to visualize, specify, construct and document the artifacts of software. UML is conformant to MOF (Meta Object Facility), a metamodeling architecture used for metadata-driven interchange and metadata manipulation (OMG, 2006).

Although derived from other methods and languages, UML is a graphical modeling language to be used in a software process. Despite many problems (Henderson-Sellers, 2005), the language has proven to be useful in the development of systems in many domains, such as automotive and telecommunication systems (Lavagno et al., 2003), real-time systems (Douglass, 2004) and safety-critical complex systems (Anda et al., 2006).

There are thirteen types of diagrams in UML 2.0 (simply UML in this text), which can be classified as those that model the static aspects of the system (structural diagrams, Fig. 2.3) and those that model the dynamic aspects of the system (behavioral diagrams, Fig. 2.4) .

UML was created mainly for general software development. Customizations for more specific purposes were created, such as MARTE (OMG, 2008b) for

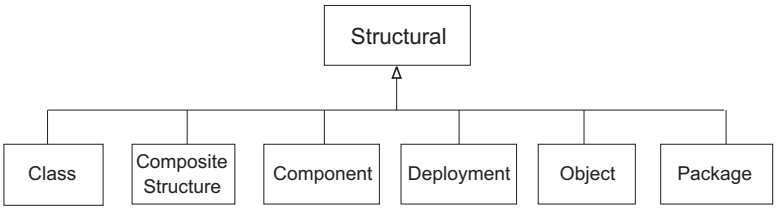

Figure 2.3: UML Structural Diagrams

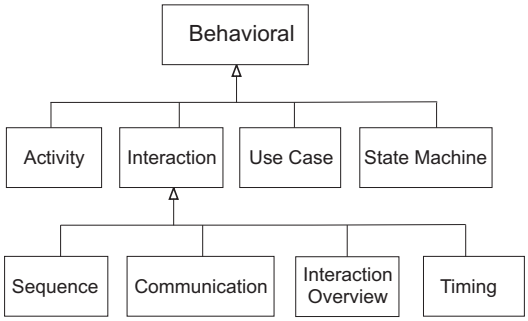

Figure 2.4: UML Behavioral Diagrams

distributed real-time systems, and SysML for Systems Engineering.

2.3.2 SysML

SysML is a systems modeling language that supports the specification, analysis, design, verification and validation of a broad range of complex systems (OMG, 2008a). The language is derived from UML, taking into account systems aspects such as hardware, software, information, processes and personnel. This may facilitate the communication between heterogeneous teams (for instance, mechanical, electrical and software engineers) that work together to develop a software-intensive system. The language is effective in specifying requirements, structure, behavior, allocations of elements to models, and constraints on system properties to support engineering analysis. SysML is supported by the OMG Systems Engineering Domain Special Interest Group and by INCOSE (International Council on Systems Engineering).

SysML is considered both a subset and an extension of UML (see Fig. 2.5). As a subset, UML diagrams considered too specific for software (Objects and Deployment diagrams) or redundant with other diagrams (Communication and Time Diagrams) were not included in SysML. Some diagrams are derived from UML without significant changes (Sequence, State-Machine, Use Case, and Package Diagrams), some are derived with changes (Activity, Block Definition, Internal

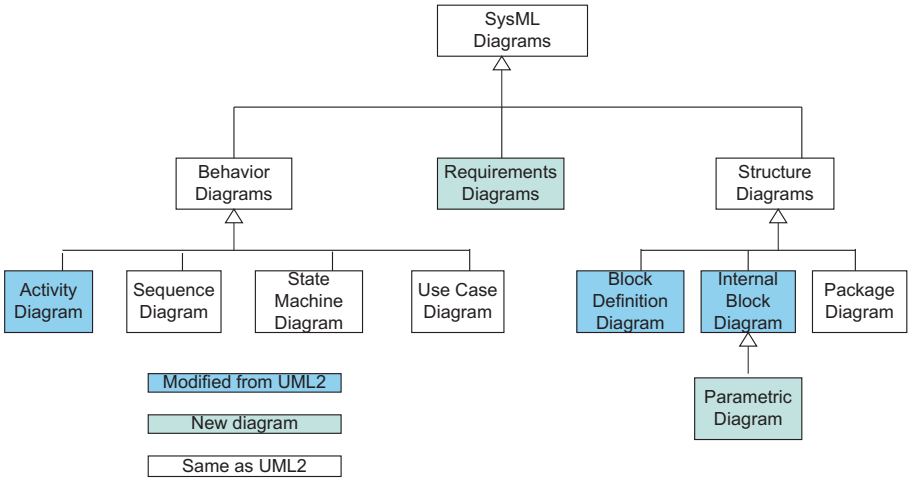

Figure 2.5: SysML Diagrams and the relation to UML

Block Diagrams) and there are two new diagrams (Requirements and Parametric Diagrams). As a matter of fact, SysML is compatible with UML, which can facilitate the integration of the disciplines of Software and System Engineering. Nevertheless, there is still a lack of research on using both languages together, and the boundaries and relationships are not yet clear. For instance, the syntax of UML and SysML Use Cases are the same, but the difference in semantics was not well-defined in the SysML specification document. Another example is the relationship between a SysML Block and a UML Class. According to the SysML specification, SysML Blocks are based on UML Classes as extended by UML composite structures. The main issues are what exactly this "based" means, and how they can be related in practice.

2.4 The Need for Integration of Modeling Languages

Many methods and languages have been proposed for software and systems development (Sommerville, 2007). These methods and languages do not cover or are not considered "good enough" by experts to cover all the steps of the software life cycle (Smith, 2008). Normally a set of views is necessary in order to capture the essential aspects that need to be modeled during the development of software-intensive systems. In most cases the modeling language can not model all essential aspects of a software-intensive system and needs to be complemented or adapted (Jackson, 1995). One clear example are the extensions to structured methods proposed in (Hatley and Pirbhai, 1987) in order to design real-time systems.

Paige (Paige, 1999) defines method integration as "the process of combining two or more methods to form a new method which is more useful than any of the separate methods". It concerns the combined use of different languages or methods which, when considered independently, are not sufficient to tackle the multiple views (data, component, behavior, concurrency, etc) necessary to design software-intensive systems. The basic purpose is to complement the modeling languages during system design in order to get the best of each modeling language, by combining strengths and reducing weaknesses.

2.4.1 Integrating Modeling Languages

The integration of a formal method in the system design process provides improved semantics, some even provide model execution by simulation and formal proofs, contributing to the early validation of requirements. Nevertheless, the integration of methods and languages may have drawbacks in particular situations (Paige, 1997). For instance, it may be difficult to manage multiple syntaxes in specifications. In addition, how to construct relationships between several methods is an important issue. A discussion on when methods are complementary is given in (Paige, 1999).

There are many examples of the integration of modeling languages. Method integration has been studied in a number of contexts: combining formal and heuristic methods (Jackson, 1983; Semmens et al., 1992; Kronlöf, 1993; Polack et al., 1993), and combining multiple formal methods (George et al., 1995; Bicarregui et al., 1996; Paige, 1998; Hoenicke and Olderog, 2002; Taguchi et al., 2004; Karkinsky et al., 2007). As UML is currently the "de facto" modeling language, a variety of researches have been proposed on integrating UML and other modeling languages.

2.4.2 Integration of UML with other Modeling Languages

In order to try to improve UML semantics and to apply UML to critical systems, some approaches were proposed that combine UML diagrams with formal methods. For instance, UML-B (Snook and Butler, 2006) is a precise and semantically well-defined profile created by the combination of UML and the B-method. The profile has been used in several real-time control systems in industry projects.

The integration of UML and Petri nets is often studied. UML provides diagrams that are strong in representing structure, which is not well-represented with Petri nets. On the other hand, dynamic behavior is better defined with Petri nets than UML. A set of rules to transform UML Sequence Diagrams to

Colored Petri for animation purposes is shown in (Ribeiro and Fernandes, 2006). In (Campos and Merseguer, 2006), UML dynamic diagrams are used for system modeling while Stochastic Petri nets are used for performance and quantitative evaluation. In (Eichner et al., 2005), UML Sequence Diagrams and some fragments are transformed to Colored Petri nets and the semantics is enriched with Algebra. In (Soares and Vrancken, 2008a), a metamodeling approach is proposed to transform UML Sequence Diagrams to Time Petri nets with inhibitor arcs, including not only synchronous and asynchronous messages, but also the operators ALT (alternatives), LOOP (iterations), and PAR (parallelism).

Chapter 3

Road ITS: Intelligent Systems Applied to Road Traffic

This chapter describes the importance of road transportation for society and gives examples of problems related to safety and congestion (Section 3.1). A proposed solution is given: deploying Intelligent Transportation Systems (ITS) to make road traffic control responsive to actual traffic conditions in order to improve traffic efficiency and safety (Section 3.2). ITS in this thesis is related to systems applied to the transportation infrastructure. A list of relevant control measures used in road ITS is presented in Section 3.3. Nevertheless, this is just the traffic engineering part of the solution. ITS are software-intensive systems. We still need the application of well-defined methods, processes and tools to build ITS, i.e., we need a systems engineering approach, as recommended in Section 3.4.

3.1 The Importance and Problems of Road Traffic

Transportation of goods and people is crucial for society. Goods need to be transported after being produced, bought or sold. People have to move from one place to another due to diverse purposes, such as going to work, to school, traveling on vacation, or even for health reasons. In modern countries, transportation itself is an important sector of the economy. The source for the statistics in this section are (Eurostat, 2006) and (Eurostat, 2009). For instance, the transportation industry accounts for about 7% of the European GDP (Gross Domestic Product). The number of direct employment in the EU-

27 [1] is currently about 1.7 million in passenger transportation (bus, coach, taxi operations) and 2.6 million in freight transportation.

Road traffic is currently the most important and flexible means of transportation. Considering the EU-27 countries, road freight transportation represented about 73% (in tonne-kilometers) of the inland freight transportation market. Rail contributed for only 17%. In the Netherlands, inland freight transportation was responsible for 61% of the total, and more than 90% in countries such as Portugal, Spain and Greece. The largest share of intra-EU passenger transportation, around 85% (in passenger-car equivalents), is made by road.

On the other hand, road traffic is also a very problematic mean of transportation. Road traffic is dangerous, expensive and presents a high pollution rate. Road congestion is costing the EU-27 about 1% of its GDP. Accidents injure or kill thousands of people every year (around 43 thousand fatalities in 2007 in EU-27 and 709 in the Netherlands). Traffic congestion in many big cities has almost gotten out of control.

Environmental damage is another issue. CO_2 emissions from transportation in general and road transportation in particular have been rising faster than emissions from all other major sectors of the economy. In 2006 the emissions from the transportation sector accounted for 23% of the total CO_2 emissions in the European Union, with road transportation generating 71% of total transportation emissions.

Basically two approaches can be applied in order to solve or at least minimize these transportation problems. The most straightforward solution is to build more infrastructure, such as bridges, roads and viaducts, in order to increase road capacity. This solution is no doubt useful, especially for decreasing congestion, but is not sufficient (McQueen and McQueen, 1999; PIARC, 1999). Constructing new road infrastructure is limited due to environmental, social and financial constraints (McDonald et al., 2006). The second approach, presented in the next section, is to control traffic, which contributes to efficiency as well as safety and environmental improvements.

3.2 Intelligent Transportation Systems

With difficulties of building more infrastructure and the aforementioned transportation problems, an approach in which already existing capacity is better

[1] The member states of the EU-27 are: Austria, Belgium, Bulgaria, Cyprus, the Czech Republic, Denmark, Estonia, Finland, France, Germany, Greece, Hungary, Ireland, Italy, Latvia, Lithuania, Luxembourg, Malta, the Netherlands, Poland, Portugal, Romania, Slovakia, Slovenia, Spain, Sweden, and the United Kingdom.

used is welcome. This is done by applying intelligence to the current infrastructure, switching from static to more dynamic road traffic control. This approach has been used with varying degrees of success, and it is the one followed in this thesis.

Intelligent Transportation Systems (ITS) (McQueen and McQueen, 1999; Stough, 2001; Maccubbin et al., 2005) refers to the application of telecommunications and information processing technology to the operation and control of transportation systems. Examples of ITS include electronic toll-collection systems, vehicle-tracking systems, in vehicle routing systems, and traffic control systems. The definition of road ITS, in short ITS, used in this thesis is as follows:

Definition 3.2.1 *Road ITS* is the use of Information, Communication and Control Technologies to transportation infrastructure and vehicles in activities such as monitoring, prediction, control, and visualization of traffic, with the purposes of better using existing capacity and improve road traffic safety.

The benefits of ITS have been recognized in many countries. In (USDOT, 1997), measured and predicted benefits in travel times, fatalities' rates, crashes, customer satisfaction, emissions and fuel consumption are described. The report was published in 1997, and by that time, the benefits were already remarkable. A more recently published report (Maccubbin et al., 2005) shows, in full detail, the benefits and costs of ITS implementations, together with a number of lessons learned by those planning, deploying, and evaluating ITS.

The research area of ITS is multidisciplinary. Examples of disciplines that are involved are Control Engineering, Traffic Engineering, Information and Communication Technologies, Software Engineering, Systems Engineering, and Psychology. Examples of ITS research areas include Emergency Management, Electronic Payment and Pricing, Traveler Information, and Freeway and Urban Management. The focus in this thesis is on dynamic road traffic management, i.e., road traffic management systems using real-time traffic data.

3.3 Road Traffic Management Systems

Road Traffic Management Systems (RTMS) are distributed real-time systems that influence traffic by using a variety of actuators, such as traffic signals and Variable Message Signs (VMS), based on acquired data using various types of sensors, such as video cameras and inductive loops. These systems' environment is inherently distributed, with vehicles and road-side equipment (sensors and actuators) geographically spread in the network. The purpose of RTMS is to

monitor, predict, visualize and control traffic. RTMS use a variety of Road Traffic Control measures to coordinate, guide and improve traffic flow. The list to be presented in this section (traffic signals, ramp metering, variable message signs, and variable speed limits) is not exhaustive, but a compilation of the most common ones and the ones most important for this thesis.

3.3.1 Traffic Signals

Traffic signals are applied to reduce congestion and delays, making crossing possible, regulate vehicle flow, warn and improve safety in a road junction for vehicles and pedestrians (Roess et al., 2003). Among the main advantages of traffic signals are the flexibility of the signaling scheme, the ability to provide priority treatment and the feasibility of coordinated control along streets.

The first traffic signal using colored lights was installed in London in 1868 at the intersection of George and Bridge Streets near the Houses of Parliament. The main purpose of the signal was to give protection to pedestrians. The first mechanical semaphore appeared in the United States about 1913 in Detroit. It displayed the words "stop" and "go" on alternate faces. The first interconnected traffic signal system was installed in Salt Lake City in 1917, with six connected intersections controlled simultaneously from a manual switch. The yellow phase in addition to the common red and green phases was first installed in a traffic signal in Detroit, in 1920. The purpose of the yellow (amber) light was to signal the drivers to "clear the intersection". A more detailed history of traffic signals can be found in (Mueller, 1970).

Modern traffic controllers implement signal timing and ensure that signal indications operate consistently and continuously in accordance with the preprogrammed phases and timing. The first computer-based traffic control system was installed in Toronto (Casicato and Cass, 1962; Roess et al., 2003).

Advanced traffic signal control systems have demonstrated great benefits for traffic, such as optimizing travel times and the number of vehicle stops, improving average speeds, and minimizing delays. The result is a moderate fuel consumption and lower environmental impact. A report provided by the U.S. Department of Transportation (FHWA, 1998) shows a decrease in travel times of 8% to 25%, and in delays of 17% to 44%.

3.3.2 Ramp Metering Systems

The purpose of ramp metering systems (Huddart, 1999) is to control the flow rate of traffic merging onto a motorway. Ramp meters are claimed to reduce congestion on motorways by reducing demand and by breaking up platoons

of cars (Huddart, 1999). When properly designed, ramp meterings can keep vehicle density below saturation, improving traffic flow on the motorway. In a study by the Minnesota Department of Transportation, ramp metering was found to increase motorway throughput in 9% on average, with a 14% increase during peak hours (Cambridge Systematics, 2001).

The control is performed by means of a traffic signal on the entrance ramp, and the flow rate is determined by selecting appropriate phase timings. Sensors on the motorway nearby the ramp provide traffic flow data to a ramp control that determines the optimum flow of traffic from the entrance ramp, and then adjusts the ramp traffic signal phases. Thus, vehicles enter the motorway from the ramp at the most appropriate rate. When the traffic signal phase is green, it allows a limited number of vehicles to proceed onto the motorway at a time. This flow rate may be increased by extending the green time phase of the traffic signal. When traffic on the motorway is already heavy, then the rate can be adjusted by decreasing green time.

A ramp metering schema is shown in Fig. 3.1 (FHWA, 2006, Chapter 3). The sensors on the main road serve a dual purpose: adjustment of the ramp metering rate in response to real-time demand and collection of historical volume and occupancy data. A demand sensor on the ramp indicates the arrival of a vehicle at the stop line and the commensurate start of the metering cycle. Demand at the stop line is typically required before the ramp signal is allowed to turn green. A passage sensor detects when the vehicle passes the stop line and returns the ramp signal to red for the next vehicle. The passage sensor can also be used to monitor meter violations, such as drivers who ignore the red stop signal, and provide historical data about the violation rate at each ramp.

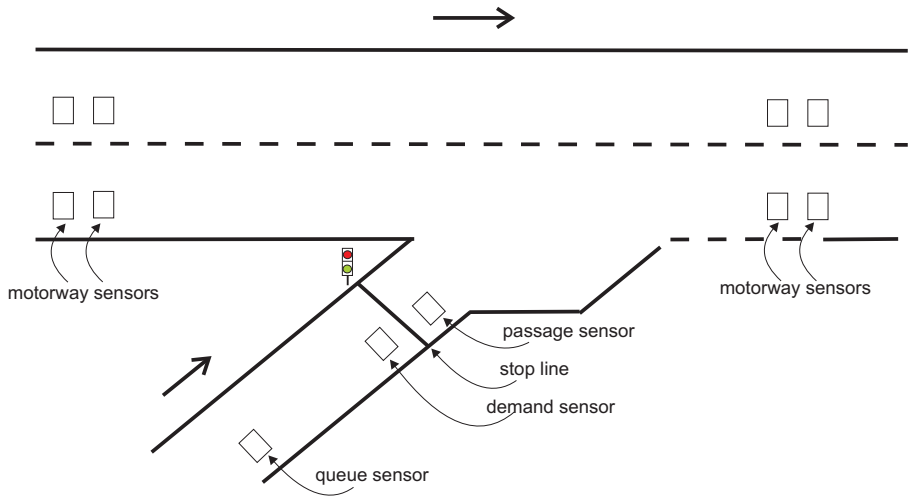

Figure 3.1: Ramp metering schema

3.3.3 Variable Message Signs

Variable Message Signs (VMS) are signs that display real-time generated messages. The basic purpose of VMS is to provide information to drivers (Fig. 3.2). Examples of information are instructions on what actions should be taken in the event of unplanned traffic incidents, such as accidents, and informing upcoming changes to traffic conditions as a result of planned traffic incidents, such as road works. Thus, in the event of a traffic congestion caused by an accident, VMS can display the expected delay. Knowing what to expect adds to the driver's comfort level. More importantly, VMS can advise drivers of alternate routes.

Figure 3.2: Variable Message Sign

3.3.4 Variable Speed Limits

Motorways normally have a maximum speed that drivers must respect. The main problem is that this speed is fixed no matter the conditions. With variable speed limit (VSL) systems (Fig. 3.3), the limit may change dynamically depending on traffic conditions such as congestion or adverse weather conditions. As the speed limit can be adapted to specific circumstances, the homogeneity of driving speed can be enhanced (van Nes et al., 2008).

In The Netherlands, the VSL signs are commonly posted on motorways. The system keeps track of all traffic movements and lowers the speed limit if it detects the start of traffic congestion. When activated, the speed limit can be set at 90, 70, or 50 km/h according to the level of expected traffic congestion.

Figure 3.3: Variable Speed Limit Signs

3.4 The need of a Systems Engineering Approach to ITS

Studies (Hecht, 1999; Galin and Avrahami, 2006; Boehm et al., 2008), have shown that using well-defined systems engineering approaches for software-intensive systems development results in better cost and schedule performance, and increases the likelihood that the implementation will meet the user's needs. Other benefits include the production of adaptable and resilient systems, improved reuse and better documentation. On the other hand, uncontrolled approaches for software-intensive systems development lead to a variety of problems observed in many reports (van Genuchten, 1991; Johnson, 2002; The Standish Group, 2003; Charette, 2005), such as cost overruns and project delays.

All the difficulties of development and maintenance of software-intensive systems presented in Chapter 1 hold for ITS in general and specifically for RTMS. For example, after deployment these systems are used for many years, which means that they must be maintained in order to cope with hardware and policy changes. This makes it very unlikely that a new ITS project will start from scratch. In most cases, legacy systems have been in operation for many years and must be taken into account. For instance, major cities have deployed urban traffic control systems with a varying degree of sophistication, control algorithms and hardware. These systems can not simply be turned-off, as they may have been offering sufficient results for many years.

The importance and potential benefits of a well-defined approach for developing ITS was recognized by the ITS community in the late 1990's (CALTRANS, 2007). The United States Department of Transportation included requirements

for a systems engineering approach in the FHWA Rule/FTA Policy (Final Policy) and the FHWA Final Rule on Architecture Standards and Conformity (Final Rule) [2]. The Final Rule requires the development of regional ITS architectures and that all ITS projects using Federal funds be developed using a systems engineering approach. The goal is to help ITS agencies, researchers and practitioners to use common, consistent and well-established systems engineering methodologies.

In the research that led to this thesis a multi-disciplinary approach, combining Traffic Engineering and Software Engineering, was used. Traffic engineers come up with new control strategies and algorithms for improving traffic. Once new solutions are defined from a Traffic Engineering point of view, there is the problem of obtaining operational systems that address all requirements. Knowing *what to build* is just the first step that must be followed by the *how to build*. Both are problematic and they depend on each other. This is an important gap noted in practice (Vrancken and Soares, 2010).

[2] see `http://ops.fhwa.dot.gov/its_arch_imp/index.htm`

Chapter 4

User Requirements Modeling of Software-Intensive Systems

Requirements Engineering is a broad area of Software Engineering research. The Requirements Engineering activities that are covered in this chapter are presented in Section 4.1. Then, an example of a subset of a list of user requirements for RTMS is presented to be further modeled and analyzed (Section 4.2). Current techniques for requirements modeling are presented in Section 4.3. A number of problems and limitations related to these techniques are discussed in the same section. These shortcomings led to a list of requirements for requirements modeling languages in Section 4.4 and the proposed approach in Section 4.5 to fulfill the missing characteristics of the list. From the conclusion of Section 4.4, the starting point for requirements modeling languages is to use SysML diagrams and tables, which are presented in detail in Sections 4.6, 4.7, and 4.8. A classification for user requirements based on the IEEE 830-1998 standard is given in Section 4.9. Then, SysML's constructions are extended in Section 4.10 and proposed to model the initial list of user requirements (Section 4.11). The chapter ends with conclusions (Section 4.12).

4.1 Activities of Requirements Engineering

Requirements for a system are a collection of needs expressed by stakeholders respecting some constraints under which the system must operate (Pressman, 2005; Robertson and Robertson, 2006). Requirements can be classified in many ways. The first classification used in this chapter is related to the level of detail (the second classification is presented in Section 4.9). In this case, the two classes of requirements are user requirements and system requirements

(Sommerville, 2007). User requirements are high-level abstract requirements based on end users' and other stakeholders' viewpoint. They are usually written using natural language, occasionally with the help of domain specific models such as mathematical equations, or even informal models not related to any method or language (Luisa et al., 2004). The fundamental purpose of user requirements specification is to document the needs and constraints gathered in order to later develop a system based on those requirements.

Systems requirements are derived from user requirements but with a detailed description of what the system should do, and are usually modeled using formal or semi-formal methods and languages. This proposed classification allows the representation of different views for different stakeholders. This is good Software Engineering practice, as requirements should be written from different viewpoints because different stakeholders use them for various purposes.

Requirements Engineering can be divided into two main groups of activities (Parviainen et al., 2004): i) requirements development, including activities such as eliciting, documenting, analyzing, and validating requirements, and ii) requirements management, including activities related to maintenance, such as tracing and change management of requirements. This chapter is about user requirements development, mainly the activities of documenting and analyzing user requirements for software systems. These are modeling activities that are useful for further Requirements Engineering activities. The assumption in this chapter is that improving requirements modeling may have a strong impact on the quality of later requirements activities, such as requirements tracing, and in the design phase. The modeling is proposed for a list of user requirements for RTMS.

4.2 List of Requirements for RTMS

The list of requirements given below is a subset from a document which contains 79 atomic requirements for RTMS (AVV, 2006). The document is a technical auditing work based on an extensive literature study and interviews, in which the stakeholders were identified. The complete list of requirements is presented in Appendix B. The requirements were gathered through interviews with multiple stakeholders. These requirements have also been gathered from the documents written by the involved stakeholders (AVV, 2006).

The stakeholders (and the related number of requirements) were classified as: the Road Users (1), the Ministry of Transport, Public Works and Water Management (2), the Traffic Managers (10), the Traffic Management Center (8), the Task, Scenario and Operator Manager (22), the Operators (4), the Designers of the Operator's Supporting Functions (15), and the Technical Quality

Managers (17). In this chapter the requirements of the Traffic Manager and the Traffic Management Center were selected as example to be modeled using SysML diagrams and constructions in Section 4.11. The requirements are given as follows.

Traffic Manager

- TM4 - It is expected that software systems will be increasingly more intelligent for managing the traffic-flow in a more effective and efficient manner.
- TM5 - To optimize traffic flow, it is expected that gradually, region-wide traffic management methods will be introduced.
- TM6 - The traffic management systems must have a convenient access to region-wide, nation-wide, or even European-wide parameters so that the traffic-flow can be managed optimally.
- TM7 - It must be possible for the traffic managers/experts to express (strategic) "task and scenario management frames", conveniently.
- TM8 - The system should effectively gather and interpret all kinds of information for the purpose of conveniently assessing the performance of the responsible companies/organizations that have carried out the construction of the related traffic systems and/or infrastructure.
- TM9 - The system must support the traffic managers/experts so that they can express various experimental simulations and analytical models.
- TM10 - The system must enable the traffic managers/experts to access various kinds of statistical data.
- TM11 - The system must enable the traffic managers/experts to access different kinds of data for transient cases such as incidents.
- TM12 - The system must provide means for expressing a wide range of tasks and scenarios.
- TM13 - The traffic management will gradually evolve from object management towards task and scenario management.

Traffic Manager Center

- TMC14 - The operational costs of the traffic management centers and shared resources must be minimized.

- TMC15 - The operators' reaction speed must be improved, especially in critical and unanticipated situations.
- TMC16 - The operators' decision accuracy must be improved, especially in critical and unanticipated situations.
- TMC17 - The system must provide means to manage various "traffic management configuration information" conveniently.
- TMC18 - The system must provide tools so that the operators can perform their work more efficiently.
- TMC19 - The system must provide tools so that the operators can perform their work more effectively.
- TMC20 - The system must make it intuitively obvious in which function/-context the operator is working in.
- TMC21 - The education material and process necessary to train the operators must be simplified, standardized and supported. This should improve the effectiveness of tutoring.

4.3 Requirements Modeling Approaches

There are several approaches to modeling requirements. Basically, these approaches can be classified as graphics-based, purely textual, or a combination of both. Some are generic while others are part of a specific methodology.

The most common approach is to write user requirements using *natural language*. The advantage is that natural language is the main mean of communication between stakeholders. However, problems such as imprecision, misunderstandings, ambiguity and inconsistency are common when natural language is used (Kamsties, 2005).

With the purpose of giving more structure to requirements documents, *structured natural language* is used (Cooper and Ito, 2002). Nevertheless, structured natural language is neither formal nor graphical, and can be too much oriented to algorithms and specific programming languages. Other collateral effects are that structured specifications may limit too early the programmers' freedom, and are mostly tailored towards procedural languages, being less suitable for some modern languages and paradigms.

User Stories have been used as part of the eXtreme Programming (XP) (Beck, 1999) agile methodology. They can be written by the customer using non-technical terminology, in the format of sentences using natural language. Although XP offers some advantages in the Requirements Engineering process in

general, such as user involvement and defined formats for user requirements and tasks, requirements are still loosely related, not graphically specified, and oriented to a specific methodology.

A well-known diagram used for requirements modeling are the *Use Cases*. Even before UML emerged as the main Software Engineering modeling language, Use Cases were already a common practice for graphically representing functional requirements in other methodologies, such as Object-Oriented Software Engineering (OOSE) (Jacobson, 1992). Their popularity can be explained due to their simplicity, making them act as a bridge between technical and business stakeholders, the compact graphical nature to represent requirements that may be expanded to several pages, and even as a basis for managers when doing project estimation (Diev, 2006). Use Cases also have some disadvantages and problems (Simons, 1999). They are applied mainly to model functional requirements and are not very helpful for other types of requirements, such as non-functional ones (Soares and Vrancken, 2007c). Use Case diagrams lack well-defined semantics, which may lead to differences in interpretation by stakeholders. For instance, the *include* and *extend* relationships are considered similar, or even the inverse of each other (Jacobson, 2004). In addition, Use Cases may be misused, when too much detail is added, which may incorrectly transform the diagrams into flowcharts or making them difficult to comprehend. Finally, although being an important part of an object-oriented language, the diagram itself is not object-oriented (Booch et al., 2005).

Two SysML diagrams are distinguish as useful mainly for Requirements Engineering activities: the *SysML Requirements* diagram and the *SysML Use Case* diagram (OMG, 2008a). One interesting feature of the SysML Requirements diagram is the possibility of modeling other type of requirements besides the functional ones, such as non-functional requirements. The SysML Use Case diagram is derived from the UML Use Case diagram without important modifications. In addition to these diagrams, *SysML Tables* can be used to represent requirements in a tabular format. Tabular representations are often used in SysML but are not considered part of the diagram taxonomy (OMG, 2008a). Detailed explanation about SysML diagrams and tables for Requirements Engineering are given in Sections 4.6, 4.7, and 4.8.

A comparison of the aforementioned requirements modeling approaches is given in the next section. The objective is to identify shortcomings of these approaches, which is used as the starting point of the proposed approach for a solution, in Section 4.5.

4.4 Desirable Requirements Specification Properties for Software-Intensive Systems

A list of desirable requirements for requirements modeling languages, together with a mapping of common languages and techniques, is given in Table 4.1. This non-exhaustive list of requirements for requirements modeling languages is based on literature review presented in Subsection 2.1.1, on the modeling languages briefly presented in Section 4.3, and on specific texts about requirements (IEEE, 1998; Beck, 1999; Luisa et al., 2004; Robertson and Robertson, 2006). This list uses "(M) Must have" and "(S) Should have" for each entry of the table, according to the MoSCoW labels (Page et al., 2003) (Must, Should, Could, Want/Won't have).

The characteristics proposed in (IEEE, 1998, Section 4.3) (correct, unambiguous, complete, consistent, ranked for importance, ranked for stability, verifiable, modifiable, and traceable) are related to a good Software Requirements Specification (SRS) document. In this chapter, these characteristics were used in the context of requirements modeling languages and techniques.

The reason for each entry of the list is given as follows.

4.4.1 Must Have Requirements

The modeling languages must provide graphical meanings to express requirements. Software-intensive systems are often designed from a multi-actor point of view, with multiple designers from diverse fields working together. Common graphical models may facilitate their communication. Models must be human readable, as the multiple involved stakeholders have to understand the models. In this case a balance is necessary, as the more machine readable requirements are, the less human readable they become. In addition, as multiple stakeholders and designers with different backgrounds are involved, the modeling languages should be as methodology independent as possible.

It is well-known by Software Engineering researchers and practitioners that requirements are related to each other (Robertson and Robertson, 2006). The survey (Robinson et al., 2003) introduces the discipline of Requirements Interaction Management (RIM), which is concerned with the analysis and management of dependencies among requirements. These interactions affect various software development activities, such as release planning, change management and reuse. A study has shown that the majority of requirements are related to or influence other requirements (Carlshamre et al., 2001). Due to this fact, it is almost impossible to plan systems releases only based on the highest priority requirements, without considering which requirements are related to each other.

From a project management point of view, one important characteristic of a requirement is its priority. Prioritizing requirements is an important activity in Requirements Engineering (Davis, 2003). The purpose is to give an indication of the order in which requirements should be addressed. Another important property of a requirement from the project management point of view is to identify its risk. For instance, a manager may be interested in identify what is the impact for a project if a specific requirement is not fulfilled. Risk management is basically the activity concerned with trying to detect previously risks in a project and preventing problems with specific plans.

Despite their importance, non-functional requirements are usually not properly addressed in requirements modeling languages. For instance, UML Use Case diagrams are strong in modeling functional requirements. The various types of requirements must be identified in order to provide better knowledge of requirements for the stakeholders.

From the software design point of view, grouping requirements in the early phases of software development helps in identifying subsystems, components, and relationships between them. As a matter of fact, grouping requirements has a positive effect in designing the software architecture.

According to (IEEE, 1998), a SRS should be consistent and modifiable. In this thesis, these two properties of a SRS are considered of great significance, and are grouped with "must have" requirements. The reason is that inconsistency between documents (Boehm, 1973; Pressman, 2005) and difficulty of changing requirements (Berry, 2004) are major causes for future problems during software development. A SRS is consistent if it agrees with other documents of the project, such as project management plans and system design models. Thus, the modeling language must be able to highlight conflicting requirements and non-conformances between requirements and design. A SRS is modifiable if its structure and style are such that any changes to the requirements can be made easily, completely, and consistently while retaining the structure and style. For instance, requirements must be expressed individually, rather than intermixed with other requirements. Thus, the modeling language must be able to describe requirements in a well-structured way.

Finally, as changing requirements is a source for problems, knowing how stable a requirement is, i.e., how ready it is for further design phases, is essential.

4.4.2 Should Have Requirements

Ambiguity should be solved, as ambiguity in requirements is a major cause for misunderstandings between stakeholders and designers. Thus, modeling languages should provide well-defined semantics, which increases machine read-

ability.

According to (IEEE, 1998), a SRS is correct if every requirement stated is one that the software shall meet. The user can determine if the SRS correctly reflects his/her actual needs. Thus, the modeling languages should facilitate the user in this activity.

According to (IEEE, 1998), a SRS is complete if all significant requirements of every type are included. Thus, the modeling languages should be able to specify all types of requirements.

According to (IEEE, 1998), a SRS is verifiable if there is a cost-effective process with which a person or machine can check that the software meets the requirement. In general, any ambiguous requirement is not verifiable. Thus, the modeling language should provide non-ambiguous constructions in order to facilitate that the designer can create non-ambiguous requirements models.

According to (IEEE, 1998), a SRS is traceable if the origin of each of its requirements is clear, and if it facilitates the referencing of each requirement in future development. Thus, the modeling language should provide means to trace the requirement through design phases. In addition, the type of these relationships should be explicit.

4.4.3 Resulting Table

Table 4.1 maps the list of requirements for modeling languages discussed in this section with the modeling languages discussed in Section 4.3. In the table, NL stands for Natural Language, SNL stands for Structured Natural Language, XP stands for the XP User Stories, UC stands for both SysML and UML Use Cases, RD stands for SysML Requirements diagram, and T stands for SysML Tables.

From the table, it is clear that “Must have” requirements, such as “Priority between requirements”, “Requirements risks”, “Identify types of requirements”, and “Ranking requirements by stability” are partially addressed or not addressed at all by most of the studied requirements modeling languages. The next section presents the approach followed in order to try to fulfill all the given requirements.

Another conclusion from the table is that some “Must have” requirements and the majority of “Should have” requirements are fulfilled or at least partially fulfilled by a combination of SysML Requirements diagram and SysML Tables. Thus, a possible starting pointing to address all requirements is to extend these SysML constructions.

List of requirements	NL	SNL	XP	UC	RD	T
(M) Graphical modeling	○	○	○	◐	●	○
(M) Human readable	●	◐	●	◐	◐	◐
(M) Independent towards methodology	●	◐	○	●	●	●
(M) Relationship between requirements	○	○	○	◐	●	●
(M) Relationship requirements/design	○	○	○	○	●	◐
(M) Requirements risks	○	○	◐	○	○	○
(M) Identify types of requirements	◐	◐	◐	○	○	○
(M) Priority between requirements	○	◐	●	○	○	○
(M) Non-functional requirements	●	●	●	○	◐	◐
(M) Grouping related requirements	●	○	○	●	○	○
(M) Consistency	○	○	○	◐	●	●
(M) Modifiable	◐	◐	◐	◐	●	●
(M) Ranking requirements by stability	○	○	◐	○	○	○
(S) Solve ambiguity	○	◐	○	○	◐	◐
(S) Well-defined semantics	○	◐	○	◐	◐	◐
(S) Machine readable	○	○	○	◐	◐	●
(S) Correctness	◐	◐	◐	◐	◐	◐
(S) Completeness	●	◐	◐	◐	●	●
(S) Verifiable	○	◐	○	◐	◐	◐
(S) Traceable	◐	◐	◐	◐	●	◐
(S) Type of relationship requirements	○	○	○	◐	●	●

Table 4.1: List of requirements properties and representation techniques

4.5 Proposed Approach

With the explicit choice to use SysML, the proposal starts with detailing SysML capacities for Requirements Engineering (Sections 4.6, 4.7, and 4.8). In Section 4.9, a classification for each atomic requirement is proposed, avoiding the confusion of which type of requirement is written in the user requirements document. The basic SysML Requirements diagram is extended with new requirements properties such as priority. Individual requirements modeled by the SysML Requirements diagram may be combined depending on their semantics. This can be useful for the early discovery of subsystems, in project management activities such as release planning, and to propose the system architecture (Section 4.10). User requirements are also represented in a tabular format, which may facilitate requirements tracing during the system life cycle. This is important to know what happens when related requirements change or are deleted, which improves traceability. Finally, Use Case diagrams are used to represent the actors involved and the scenarios to be implemented (Section 4.11). Then, Use Cases are related to SysML Requirements using one of the proposed relationships.

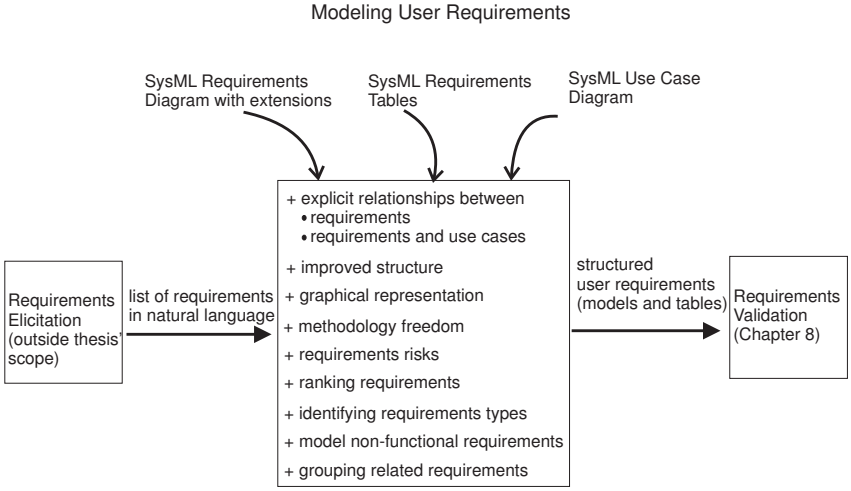

Figure 4.1: Approach for Modeling User Requirements with SysML

Although the idea in this chapter is to use graphical models already in the early phases of system development, natural language is still considered important. Despite its problems, there are also advantages, as natural languages are the primary communication medium between people.

After being structured and graphically represented (Fig. 4.1) using SysML Tables, SysML Requirements and SysML Use Case diagrams, user requirements are detailed into system requirements, being specified using other models, such

as other UML/SysML diagrams or using formal methods (Chapters 6 and 7).

The SysML constructions (diagrams and tables) for modeling user requirements are explained in detail in the following section.

4.6 SysML Requirements Diagram

The SysML Requirements diagram helps in better organizing requirements, and also shows explicitly the various kinds of relationships between different requirements. Another advantage of using this diagram is to standardize the way of specifying requirements through a defined semantics. As a direct consequence, SysML allows the representation of requirements as model elements, which means that requirements are part of the system architecture (Balmelli et al., 2006).

The SysML requirements constructs are intended to provide a bridge between traditional requirements management specifications and the other SysML models. When combined with UML for software design, the requirements constructs provided by SysML can also fill the gap between user requirements specification, normally written in natural language, and Use Case diagrams, used as initial specification of system requirements.

A SysML Requirement can also appear on other diagrams to show its relationship to design. With the SysML Requirements diagram, visualization techniques are applied from the early phases of system development. The SysML Requirements diagram is a stereotype of the UML Class diagram, as shown in Fig. 4.2.

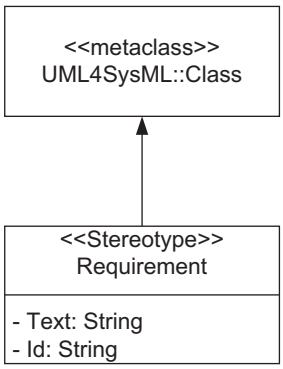

Figure 4.2: Basic SysML Requirements diagram

Implementing all requirements in a single system release may be unattractive because of the high cost involved, lack of sufficient staff and time, and even client

and market pressures. These difficulties make prioritization a fundamental activity during the Requirements Engineering process. Prioritizing requirements is giving an indication of the order in which requirements should be considered for implementation. However, it is not always possible to plan a system release based only on the set of more important requirements due to requirements relationships. A better knowledge of requirements relationships may be useful to make more feasible release plans, to reuse requirements and to drive system design and implementation.

The SysML Requirements diagram allows several ways to represent requirements relationships. These include relationships for defining requirements hierarchy, deriving requirements, satisfying requirements, verifying requirements and refining requirements. The relationships can improve the specification of systems, as they can be used to model requirements. The relationships: *hierarchy*, *derive*, *master/slave*, *satisfy*, *verify*, *refine* and *trace* are briefly explained as follows.

In large, complex systems, it is common to have a hierarchy of requirements, and their organization into various levels helps in dealing with system complexity. For instance, high-level business requirements may be gradually decomposed into more detailed software requirements, forming a hierarchy. Discovering the hierarchy of requirements is an important design step in Requirements Engineering. SysML allows splitting complex requirements into more simple ones, as a *hierarchy* of requirements related to each other (represented by the symbol $\oplus$). The advantage is that the complexity of systems is treated from the early beginning of development, by decomposing complex requirements.

The concept of hierarchy also permits the reuse of requirements. In this case, a common requirement can be shared by other requirements. The hierarchy is built based on master and slave requirements. The slave is a requirement whose text property is a read-only copy of the text property of a master requirement. The master/slave relationship is indicated by the use of the *copy* keyword.

The derive relationship relates a derived requirement to its source requirement. During Requirements Engineering activities, new requirements are created from previous ones. Normally, the derived requirement is under a source requirement in the hierarchy. In a Requirements diagram, the derive relationship is represented by the keyword *deriveReqt*.

The satisfy requirement describes how a model satisfies one or more requirements. It represents a dependency relationship between a requirement and a model element, such as other SysML diagrams, that represents that requirement. This relationship is represented by the keyword *satisfy*. One example is to associate a requirement to a SysML Block diagram.

The verify relationship defines how a test case can verify a requirement. This

includes standard verification methods for inspection, analysis, demonstration or test. For example, given a requirement, the steps necessary for its verification can be summarized by a state-machine diagram. The keyword *verify* represents this relationship.

The refine relationship provides a capability to reduce ambiguity in a requirement by relating a SysML Requirement to another model element. This relationship is typically used to refine a text-based requirement with a model. For example, how a Use Case can represent a requirement in a SysML Requirements diagram. The relationship is represented in the diagram by the keyword *refine*. The refinement is distinguished from a derive relationship in that a refine relationship can exist between a requirement and any other model element, whereas a derive relationship is only between requirements.

The *trace* relationship provides a general purpose relationship between a requirement and any other model element. Its semantics has no real constraints and is not as well-defined as the other relationships. For instance, a generic trace dependency can be used to emphasize that a pair of requirements are related in a different way not defined by other SysML relationships.

4.7 SysML Requirements Table

Requirements traceability is an important quality factor in a systems's design. A definition of requirements traceability is given in (Gotel and Finkelstein, 1994) as: "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 periods of ongoing refinement and iteration in any of these phases". Basically, requirements traceability helps in identifying the origin, destination, and links between requirements and models created during system development.

Identifying and maintaining traces between requirements are considered important activities during Requirements Engineering (Sahraoui, 2005). The activity of requirements tracing is very useful, for example, to identify how requirements are affected by changes. For instance, in later development phases a requirement may be removed, and the related requirements may also be deleted or reallocated. Another case is when a requirement has changed and the stakeholders need to know how this change will affect other requirements. Traceability also helps to ensure that all requirements are fulfilled by the system and subsystem components. When requirements are not completely traced to the specific design elements, there is a tendency to lose focus as to the specific responsibility of each design model. This can lead to costly changes late in the life cycle and can also lead to incorrect or missing functionality in the delivered system. As

a matter of fact, important decisions on requirements and the correspondent models are better justified when traceability is given proper attention (Ramesh and Jarke, 2001). One way to manage the requirements traceability in SysML is by using requirements tables.

Table 4.2: A SysML Hierarchy Requirements table

Id	Name	Type

SysML allows the representation of requirements, their properties and relationships in a tabular format. One proposed table shows the hierarchical tree of requirements from a master one. The fields proposed for Table 4.2 are the requirement's ID, name and type. There is a table for each requirement that has child requirements related by the relationship *hierarchy*.

4.8 SysML Use Case Diagram

The SysML Use Case diagram is derived without important extensions from the UML 2.0 Use Case diagram. The main difference is the wider focus, as the idea is to model complex systems that involve not only software, but also other systems, personnel, and hardware.

The Use Case diagram shows system functionalities that are performed through the interaction of the system with its actors. The idea is to represent what the system will perform, not how. The diagrams are composed of actors, use cases and their relationships. Actors may correspond to users, other systems or any external entity to the system.

There are four types of relationships in a Use Case diagram: *communication*, *generalization*, *include* and *extend*. The *communication* relationship is used to associate actors and use cases when they effectively participate in the use case behavior. The *generalization* relationship occurs from actor to actor or from use case to use case. The semantics is the same used in other diagrams, such as the UML Class diagram: the child element inherits all behavior of its parent, and can add some more specific behavior. The *include* relationship provides a mechanism useful when a sequence of events is common to more than one use case. These sequences of events can be encapsulated as one use case and reused by other use cases. The execution of the base use case implies also in the execution of the included use cases. The *extend* relationship provides optional functionality, which extends the base use case at defined extension points under

specified conditions. This relationship is useful when a use case is too complex, with many alternatives and optional sequences of interactions. The solution is to separate each alternative or option of the base use case into another use case, and relate them using the *extend* keyword. The base use case is independent of the extended ones, which may only be executed if the condition in the base use case that causes it to execute is set to true.

The detailed sequence of events in a use case can be represented in different manners. It is common to describe the sequence of events in structured language based on a pre-defined pattern. Considering a model-driven approach, it is also possible to specify use case behavior by Activity diagrams (Almendros-Jiménez and Iribarne, 2005), Sequence diagrams (Almendros-Jiménez and Iribarne, 2007), or Petri nets (Soares and Vrancken, 2008d). Within SysML, a use case may also be related to a SysML Requirements diagram. Which of these techniques to use depends on the intended reader and the development phase. A combination of techniques can also be used in order to present the best manner to each stakeholder.

One important limitation of Use Cases diagrams is that their focus is on specifying only functional requirements. Non-functional requirements, such as performance, and external requirements, such as interfaces, which are fundamental in software-intensive systems, are not well-represented by Use Case diagrams.

4.9 User Requirements Classification

A common classification proposed for requirements in the literature is based on the level of abstraction, in which requirements are classified as functional or non-functional (Robertson and Robertson, 2006). Functional requirements describe the services that the system should provide, including the behavior of the system in particular situations. Non-functional requirements are related to emergent system properties such as safety, reliability and response time. These properties cannot be attributed to a single system component. Rather, they emerge as a result of integrating system components. Non-functional requirements are also considered quality requirements, and are fundamental to determine the success of a system.

A table of contents of a Requirements Specification with the following requirements items: external interfaces, functions, performance, logical database, design constraints, and software system attributes, is suggested in (IEEE, 1998). For sake of simplicity, and as some of the items can be considered non-functional requirements (performance, design constraints and software system attributes), or functional requirements (logical database), the second classification used in this Chapter (after user vs. system requirements) is as follows:

- Functional: describes what the system should do, how the system should react to particular inputs, and how the system should behave in particular situations (the functionalities).
- Non-functional: are related to emergent system properties, such as reliability and performance. These requirements do not have simple yes/no satisfaction criteria. Instead, it must be determined whether a non-functional requirement has been satisfied.
- External: a detailed description of all inputs into and outputs from the software system, such as system, user, hardware, software and communication interfaces. It is an important classification to decompose the system into subsystems, helping in the identification of system architecture.

4.10 Extensions to SysML Requirements Diagram and Tables

SysML is a highly customizable and extensible modeling language (OMG, 2008a). Organizations that develop systems for several different domains may create a profile for each domain. Profiles may specialize language semantics, provide new graphical icons and domain-specific model libraries. When creating profiles, it is not allowed to change language semantics; normally profiles may only specialize and extend semantics and notations.

The basic SysML Requirements diagram is extended in this section. The purpose is to try to address the identified shortcomings presented in Table 4.1. The first extension is performed by creating stereotypes of stereotypes, in which case they are named sub-stereotypes (Subsection 4.10.1). Sub-stereotypes are similar to class inheritance in UML: they inherit any properties of their super-stereotypes, and add their own. These stereotypes are used to express the different types of user requirements proposed in Section 4.9. The second extension is to add properties besides the two default ones (Id and Text) (Subsection 4.10.2). The third extension is about grouping related requirements (Subsection 4.10.3). The last extension is to extend the SysML Table to provide requirements in a tabular format (section 4.10.4).

4.10.1 Types of Requirements

Stereotypes are the main mechanism used to create profiles and extensions to the SysML metamodel. A stereotype extends a metaclass or another stereotype. A well-known example of a stereotype for the UML metamodel are the classes

control, *entity* and *boundary*, each one with its own graphical icon. When used in a Class diagram, these stereotypes improve semantics for the diagram readers.

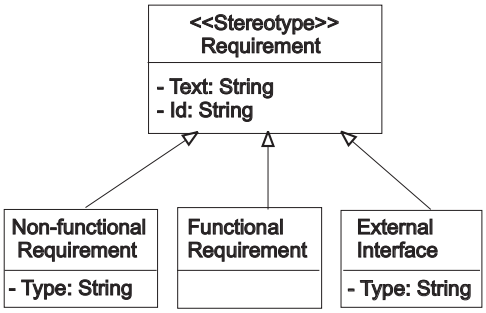

Figure 4.3: Extension to SysML Requirements diagram using the proposed user requirements classifications

According to the classification proposed in Section 4.9, three requirements stereotypes are proposed: Functional, Non-functional and External Interfaces (Fig. 4.3). The Non-functional and External Interface requirements have the property "type" that may have several tagged values. Examples of possible values are Performance, Security and Efficiency for Non-functional Requirements, and User, Hardware, Software and Communication for External Interface requirements.

4.10.2 Additional Properties

Properties add information to elements of the model, and are normally associated to tagged values encoded as strings. Tagged values add extra semantics to a model element. Constraints may also be used as semantic restrictions applied to elements. One example of a constraint is the association of the "xor constraint" specifying a restriction (exclusive or).

The Id and Text properties are default to the requirements diagram. As an addition, the following properties are proposed: Risk, Source, Priority, Responsible, Version/Date, and Relationship. These additional properties are not mandatory and may appear in any order. The requirements engineer may use all of them, some or just the original properties. The following paragraphs suggest a number of tagged values to be attached to each property, and also an explanation of each new property.

A risk is an uncertain event or condition that, if it occurs, has a positive or negative effect on a project's objectives (PMI, 2008). The Risk property is related to the requirement risks. There are at least two important values to

be added that concern risks: the probability of the risk becoming real and the effects of its occurrence. The extensions proposed attaches a tuple R = {P,I} to each requirement, in which P indicates the probability and I the impact of the effects of occurring that risk. The suggested values for P are: very low, low, moderate, high and very high. The suggested values for I are: insignificant, tolerable, serious, very serious or catastrophic. Numeric values can also be assigned, but may lead to confusion. The combination of both values can be used as input to strategies to manage project risks.

If the requirement is derived from another requirement, it is useful to know its source. The Source property describes where the derived requirement originated. This information is important to trace requirements during system life cycle development.

According to the PMBOK (PMI, 2008), knowing which requirements have high priority is useful for risk analysis and during system development. Prioritizing requirements is giving an indication of the order in which requirements should be addressed. A review of requirements prioritization techniques can be found in (Greer, 2005). Some recommendations on how to prioritize requirements (or triage) can be found in (Davis, 2003). A well-performed prioritization provides better system release planning, based on balancing importance versus effort. Ranking assignment is the simplest prioritization technique (Greer, 2005). Basically, it consists of dividing requirements into groups, giving to each requirement a label, such as (critical, standard, optional) or the MoSCoW labels. The number of groups may vary, but within a group, all requirements have the same priority.

At least the main stakeholder directly responsible for the requirement should be known. In case there is more than one responsible stakeholder, the choices are to write all of them, or just write the most important. This information is represented in the Responsible property.

The requirements version is useful to show if the requirement was changed. This property is fundamental, as uncontrolled changes are a source of problems in Requirements Engineering. In addition to the version, the date of creation/change is added.

In order to improve the activity of tracing requirements to design models, a property that relates the specific requirement to models of the design is added. Identifying and maintaining traces between requirements and design are considered important activities in Requirements Engineering (Sahraoui, 2005).

The resulting SysML Requirement with the proposed extensions is depicted in Fig. 4.4.

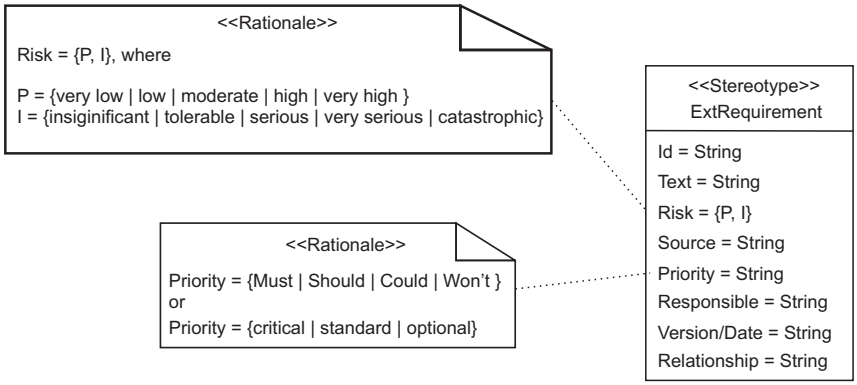

Figure 4.4: Extension to the SysML Requirements Diagram

4.10.3 Grouping Requirements

By modeling requirements with SysML, system complexity is addressed from the early system design activities. Managing decomposition is a crucial task in order to deal with complexity. Requirements may be decomposed into atomic requirements, and may later even be related in the sense that together they are capable of delivering a whole feature, i.e., they are responsible for a well-defined subsystem.

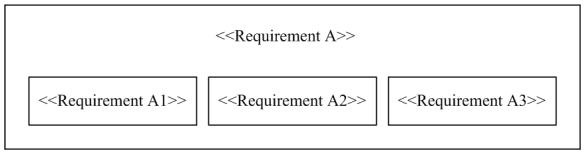

Figure 4.5: Grouping Requirements

SysML requirements may be part of other SysML requirements, as a hierarchy. Related SysML requirements can be grouped into a single SysML requirements sub-package (similar to the UML Package diagram, which combines several class diagrams), creating categories of requirements (Fig. 4.5).

4.10.4 Extension to the SysML Table

Table 4.3 shows an example of requirements data expressed in a tabular format. The proposed table shows the requirement Id, the name of the requirement, to which requirement it is related (if any), the type of relationship and the requirement type. This allows an agile way to identify, prioritize and trace requirements. As a matter of fact, whenever a requirement is changed or deleted,

the SysML Requirements Relationship Tables (SRRT) are useful to show that this can affect other requirements.

Table 4.3: A SysML Requirements Relationship Table

Id	Name	RelatesTo	RelatesHow	Type

4.11 Case Study: RTMS User Requirements Modeling with SysML

In this section, a modeling approach is applied to model the list of user requirements presented in Section 4.2.

4.11.1 SysML Requirements diagrams

The associated SysML Requirements diagrams for the list of user requirements are depicted in Figs. 4.6 and 4.7, respectively concerning the Traffic Manager and the Traffic Management Center requirements. For the sake of simplicity, not all properties are included.

4.11.2 SysML Requirements Tables

Tables 4.4, 4.5, 4.6, 4.7, and 4.8 show SysML Requirements tables expressing hierarchy for requirements TM4, TM7, TM9, TMC14, TMC15 and TMC16.

Table 4.4: Hierarchy Requirements table - TM4

Id	Name	Type
TM5	Region-wide traffic management	Functional
TM6	Traffic flow managed optimally	Functional

The other proposed type of table (SRRT), relating requirements and their relationships for each SysML Requirements diagram is presented in Tables 4.9 and 4.10.

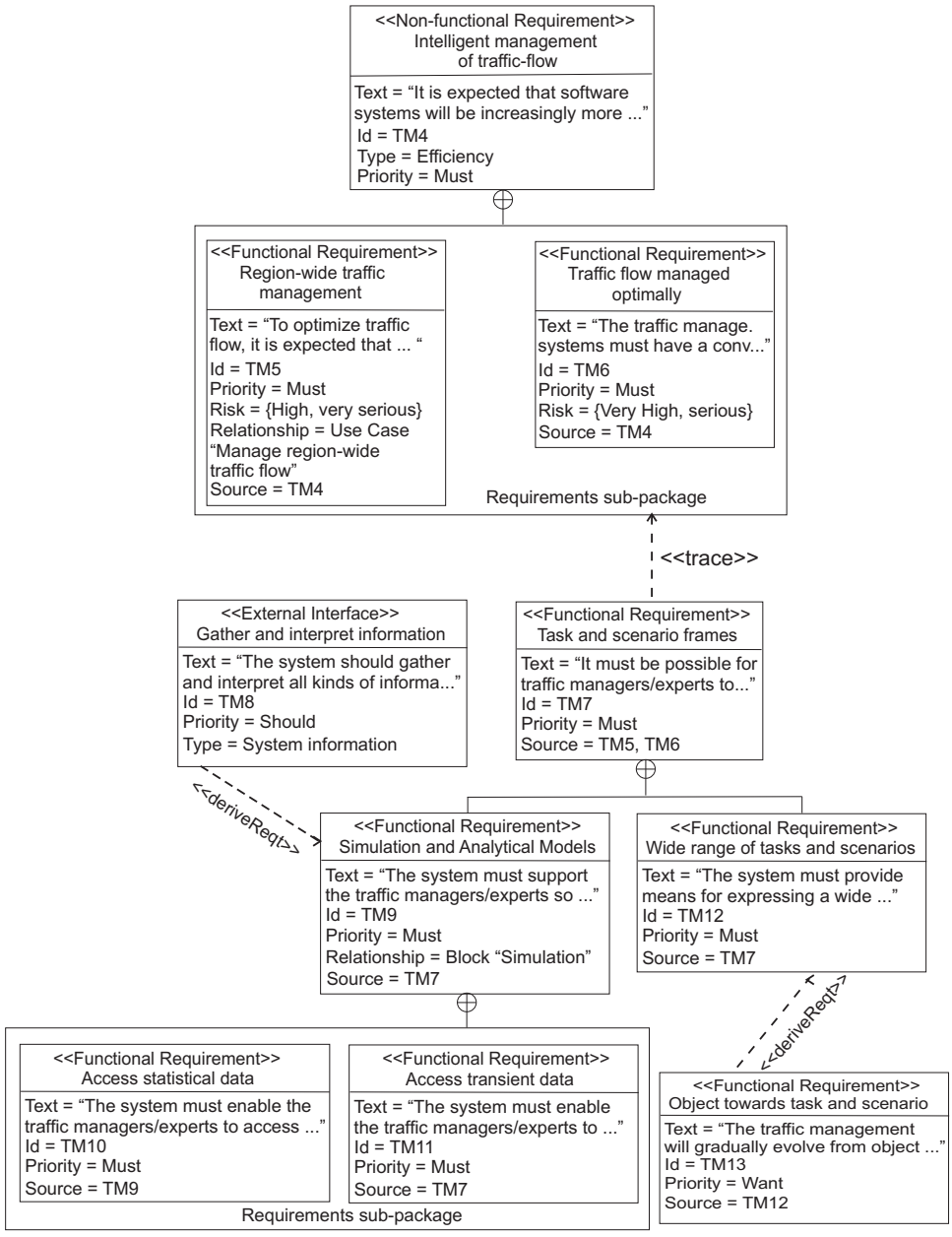

Figure 4.6: SysML Requirements diagram for Traffic Management Stakeholders

4.11.3 SysML Use Case diagrams

The associated Use Case diagrams are depicted in Figures 4.8 and 4.9, respectively concerning the Traffic Manager and the Traffic Management Center requirements.

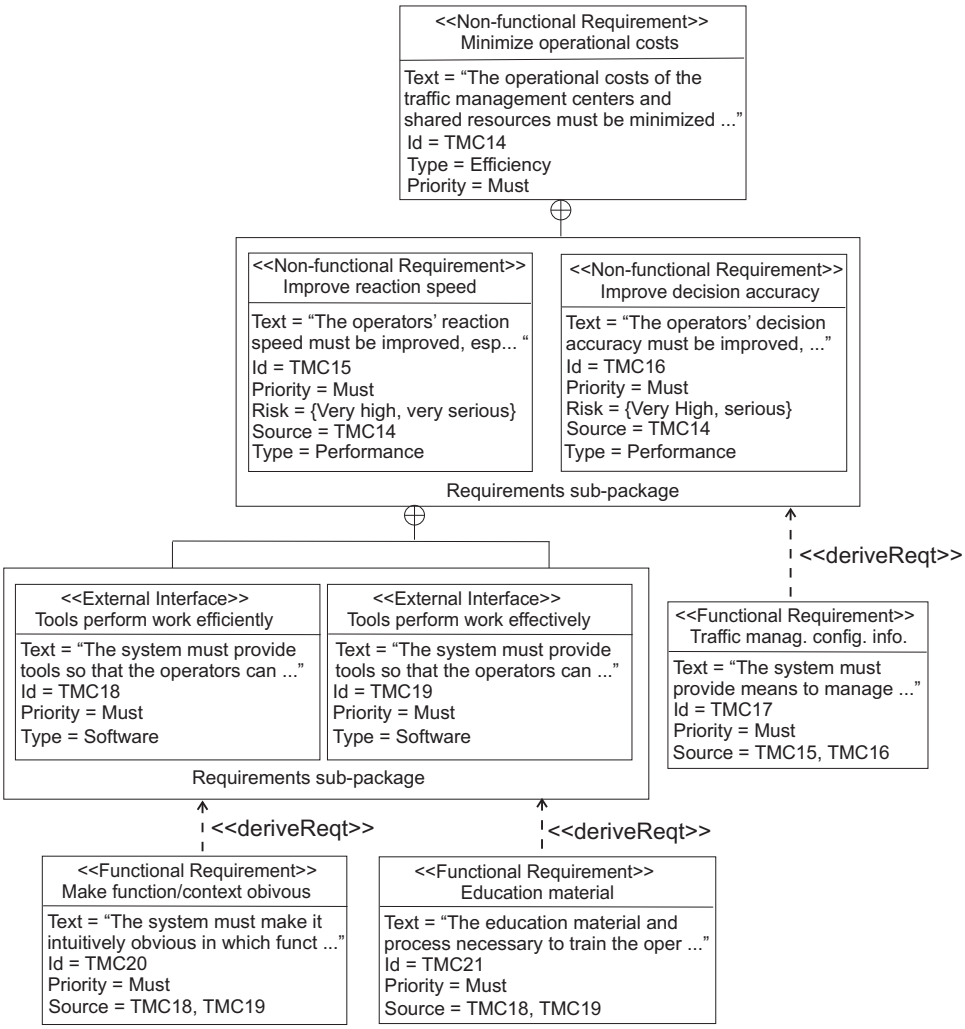

Figure 4.7: SysML Requirements diagram for Traffic Management Center Stakeholders

Table 4.5: Hierarchy Requirements table - TM7

Id	Name	Type
TM9	Simulation analytical models	Functional
TM12	Wide range tasks scenarios	Functional

4.11.4 Relationship between Use Cases and SysML Requirements Diagram

The SysML *refine* relationship can be used to relate requirements to other SysML models. For example, the Requirements sub-package representing re-

Table 4.6: Hierarchy Requirements table - TM9

Id	Name	Type
TM10	Access statistical data	Functional
TM11	Access transient data	Functional

Table 4.7: Hierarchy Requirements table - TMC14

Id	Name	Type
TM15	Improve reaction speed	Non-functional
TM16	Improve decision accuracy	Non-functional

Table 4.8: Hierarchy Requirements table - TMC15, TMC16

Id	Name	Type
TM18	Tools perform work efficiently	Functional
TM19	Tools perform work effectively	Functional

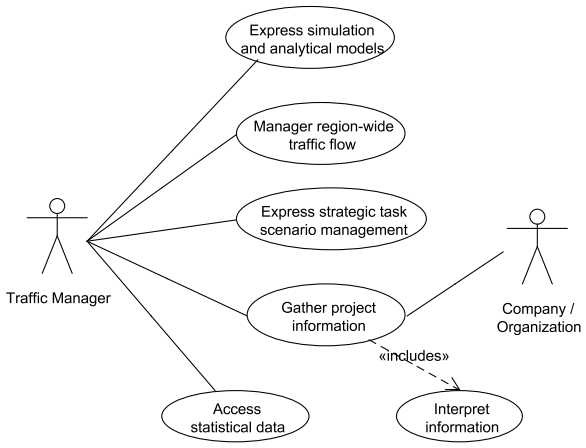

Figure 4.8: Use Case diagram for Traffic Manager

quirements TM5 and TM6 can be associated by the refine relationship to the Use Case *Manage region-wide traffic flow*, which means that the requirements are represented by the Use Case. Figure 4.10 shows this example. Later, this Use Case can be detailed by including other Use Cases and relationships, or even by using other SysML diagrams, such as the Sequence diagram. As a result, one knows which Sequence diagram models a specific SysML Requirement, which narrows the gap between requirements modeling and software design.

Table 4.9: SysML Requirements relationship table for TM

Id	Name	RelatesTo	RelatesHow	Type
TM7	Task/scenario frames	{TM5, TM6}	trace	Functional
TM8	Gather/interpret info.	TM9	deriveReqt	External
TM13	Object task/scenario	TM12	deriveReqt	Functional

Table 4.10: SysML Requirements relationship table for TMC

Id	Name	RelatesTo	RelatesHow	Type
TMC17	T.M. config. inf.	{TMC15, TMC16}	deriveReqt	Funct.
TMC20	Make context obv.	{TMC18, TMC19}	deriveReqt	Funct.
TMC21	Education material	{TMC18, TMC19}	deriveReqt	Funct.

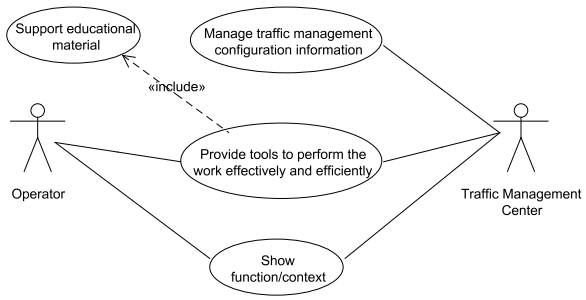

Figure 4.9: Use Case diagram for Traffic Management Center

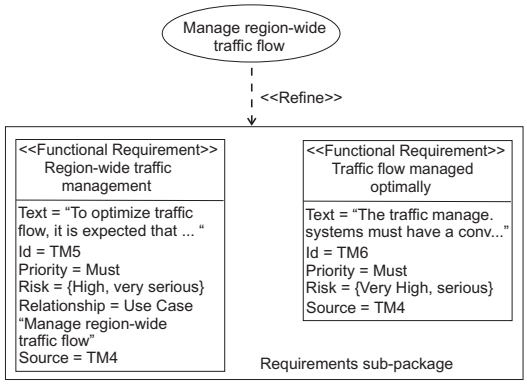

Figure 4.10: Example of the *Refine* relationship

4.12 Conclusions

It is essential to have properly structured and controlled requirements specifications that are consistent and understandable by stakeholders. This is addressed

in this chapter by presenting an approach to model and analyze a list of user requirements using the SysML Requirements diagram, the SysML Table, and the SysML Use Case diagram.

As usual in system development, changes in requirements are likely to happen, and using the SysML Requirements diagram is useful for developers to manage these changes. For instance, when a stakeholder asks for a change in one specific requirement, using the many relationship types that describe traceability between models helps to uncover possible impacts in other models. The relationships are also useful to aid in requirements prioritization in order to decide which requirements should be included in a certain system release. Another advantage of using the SysML Requirements diagram is to standardize the way of specifying requirements through a defined semantics. As a direct consequence, SysML allows the representation of requirements as model elements.

In this chapter, a classification of user requirements is proposed. Then, the SysML Requirements diagram is introduced and the requirements relationships are detailed. SysML tables are useful to represent decomposition in a tabular form and to improve traceability, which is an important quality factor when designing software-intensive systems. The SysML Requirements diagram is extended with new stereotypes including the proposed classification, which distinguish requirements as Functional, Non-functional or External. Some properties not presented in the original SysML Requirements diagram are added in order to represent important requirements characteristics. These properties were chosen based on an extensive literature review.

The list of desirable properties of requirements specification, shown in Table 4.1, is used again, this time to evaluate the approach proposed in this chapter (see Table 4.11). In the list, SRDE stands for the extended version of the SysML Requirement diagram, and SRRT for an extended version of SysML Tables used for Requirements Engineering activities.

From the list, it is clear that the proposed user classification in Section 4.9 and the extensions in Section 4.10 fulfill almost all the properties identified in Table 4.1. The partially fulfilled properties, “Well-defined semantics” and “Solve ambiguity”, are not fulfilled even when the extended SysML Requirements diagram and SysML Tables are used. These properties are solvable by increasing formality for the modeling language, i.e., by using formal methods, which is the topic of Chapter 7. However, when using formal methods, other properties, such as “human readable” may be lost.

In (IEEE, 1998), a list of characteristics that are expected for a software requirements document is given. To finalize the evaluation, how each of these characteristics are addressed by the approach presented in this chapter is briefly presented as follows:

List of requirements	SRDE	SRRT
(M) Graphical modeling	●	◐
(M) Human readable	◐	◐
(M) Independent towards methodology	●	●
(M) Relationship between requirements	●	●
(M) Relationship requirements/design	●	◐
(M) Requirements risks	●	●
(M) Identify types of requirements	●	●
(M) Priority between requirements	●	○
(M) Non-functional requirements	●	●
(M) Grouping related requirements	●	●
(M) Consistency	●	●
(M) Modifiable	●	●
(M) Ranking requirements by stability	●	○
(S) Solve ambiguity	◐	◐
(S) Well-defined semantics	◐	◐
(S) Machine readable	◐	●
(S) Correctness	◐	◐
(S) Completeness	●	●
(S) Verifiable	◐	◐
(S) Traceable	●	●
(S) Type of relationship requirements	●	●

Table 4.11: List of requirements properties and representation techniques

Correctness: according to (IEEE, 1998), no technique can ensure correctness. However, the SysML Requirements diagram provides the possibility of relating requirements to other design models, facilitating that the user can determine if the SRS correctly reflects the actual needs.

Unambiguity: ambiguity can be solved with the use of formal methods, which is the topic of Chapter 7. The issue is that natural language is ambiguous, but unavoidable in the early phases of Requirements Engineering. For reasons already discussed in Section 2.2.1, human readability of requirements is diminished when formal methods are used.

Completeness: the proposed types for requirements are well-described with the extensions proposed for the SysML Requirements diagram and Tables. Thus, all types of requirements can be modeled.

Consistency: conflicts between requirements can be discovered by explicitly describing their relationships, and the type of each relationship. In addition, by grouping related user requirements, conflicts within a group of requirements and between groups can be discovered.

Ranked by importance: Typically, not all requirements are equally important. The approach presented in this chapter fulfill this characteristic by adding two properties to the basic SysML Requirements diagram: Risk and Priority.

Ranked by stability: Stability can be expressed in terms of the number of expected/performed changes to any requirement. This is addressed in this chapter by controlling version and date of a requirement, through the additional property Version/Date.

Verifiable: as ambiguity is not solved with the application of SysML, this characteristic is not fully present. However, the advantage of using SysML is the possibility of relating SysML Requirements to formal design models that can be formally verified.

Modifiable: The requirements document is modifiable if its structure and style are such that any changes to requirements can be made completely, and consistently, while retaining the structure and style. Expressing each requirement separately is highly desirable. This characteristic is addressed in this chapter by modeling requirements using a well-defined SysML Requirements diagram, and by organizing the relationship between requirements.

Traceable: A requirement is traceable if its origin is clear and if it is possible to refer to it in future development. The solution proposed in this chapter is

to create SysML tables expressing the relationships between requirements and other design models.

The proposed approach presented in this chapter uses two SysML diagrams and SysML tables. This is necessary because multiple aspects of user requirements modeling are covered, which is useful as multiple stakeholders are involved. Thus, the SysML Use Case provides systems' view of functional requirements and actors, delimiting the system scope. Requirements relationships and properties are graphically represented using the SysML Requirements diagram, and SysML tables gives a tabular format for requirements.

Finally, requirements are important to determine the architecture. For instance, external requirements help in delimiting the system context in relation with its environment. When designing the architecture, at least part of the functional requirements should be known. In addition, the non-functional requirements that the architecture has to conform with, such as portability, performance, and other quality attributes (security, modifiability), should be made explicit. Domain architecture and software architecture are, respectively, the topics of chapters 5 and 6.

Chapter 5

Architecture for Road Traffic Management Systems

The focus of this chapter is on architectures for Road Taffic Management Systems (RTMS). Section 5.1 introduces the importance of defining a domain architecture and position the relationship between the proposed architectures in the thesis. The Architecture for Traffic Control (ATC), used in The Netherlands, is described in Section 5.2. This architecture is based on a top-down, centralized control, which has some drawbacks and limitations. These shortcomings are used as input for a set of requirements (Section 5.3) for an extension to the ATC architecture, the Distributed Traffic Control Architecture (DTCA), which is explained in Section 5.4. The DTCA architecture extends ATC by adding multi-agent control. Finally, a case study on a real-life implementation of a RTMS is presented in Section 5.5. The chapter ends with conclusion and a link to the next chapter (Section 5.6).

5.1 Positioning Multiple Architectures

ITS Architecture is defined in (McQueen and McQueen, 1999) as a framework specifying the technical, organizational, and commercial features of the future system in an outline and graphical format, showing how essential subsystems and components will work together. Developing an ITS architecture provides many benefits. The architecture shows, at an abstract level, the whole picture, and help stakeholders to understand the consequences of the decisions that are taken prior to investment in the design and development of the elements of the system. The architecture shows how the many subsystems and components are interconnected, how they cooperate, and the boundaries and

interfaces between each other. This is useful, for instance, for vendors creating their products according to proposed standards. In addition, the architecture enables the identification of major risks and how to mitigate them. Finally, better understanding of the whole is useful to create project plans with realistic budget and duration. Due to its importance, creating an ITS Architecture to develop ITS is common in many countries. Some examples follow below.

The first version of The National ITS Architecture (Stough, 2001) was proposed by the U.S. Department of Transportation in 1996. The architecture is an interconnected presentation of user services, logical architecture, physical architecture, implementation, and standards-oriented components. Currently there are 33 User Services which describe the basic purpose of a system from the user's perspective and are starting points for developing a system. Examples of User Services are "Traffic Control" and "Incident Management". Each User Service is defined by a set of requirements, that are further specified by a series of data flows, describing what is called the Logical Architecture. The detailed specifications of the subsystems, major components and the data flow interface are described in the Physical Architecture.

The Japan ITS Architecture (VERTIS, 1999) was proposed in order to deal with the original User Services which respond to Japan's own natural and social environment. It is comparable to the National Architecture in the sense that it defines User Services that are later specified in a Logical and Physical Architecture. The major difference is that within the Japan ITS Architecture it was decided to use the object-oriented paradigm to describe the Logical and the Physical Architectures. The chosen modeling language was UML.

According to the definition of architecture in Chapter 2, both the National Architecture and the Japan Architecture are more related to software detailed design and less to the domain architecture.

In Europe, the European ITS Framework Architecture (Jesty et al., 2000) was created in order to provide guidelines and a common approach to the planning, development and implementation of ITS. It was created by the KAREN Project (Keystone Architecture Required for European Networks) mainly with focus on road-based applications. The European ITS Framework Architecture is designed to provide a flexible high level framework that individual countries can tailor to their own requirements. As a matter of fact, national ITS Architectures were created by European countries, such as ACTIF (France), ARTIST (Italy), TTS-A (Austria) and TEAM (Czech Republic), based on a common approach and methodology, but each with focus on the aspects of local importance. There have been attempts to develop an ITS architecture for the Netherlands, but only the traffic control part has reached maturity. The architecture is described in the next section.

Fig. 5.1 positions this chapter and the next one, and the relationship between the multiple architectures cited in this thesis. The proposal in this thesis is that each ITS domain (e.g., Traffic Control (TC), Electronic Toll Collection (ETC)) should have an architecture. This is important from the practical point of view, as more specific domain architectures provide better separation of concerns and more abstract levels, which are considered good principles of Software Engineering (Bourque et al., 2002) (see Section 1.1) and scientific knowledge in general (Dijkstra, 1982).

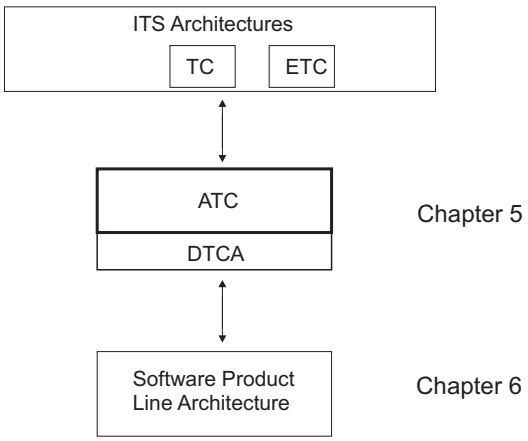

Figure 5.1: Relationship between ITS, Traffic Control, and Software Product Line Architectures.

Architecture for Road Traffic Control (RTC), i.e., the domain architecture, is the main topic of this chapter. First the ATC is presented, and due to its drawbacks and limitations, the DTCA is proposed as an extension. The implementation of a family of systems that conform to this architecture and its requirements is the topic of Chapter 6, in which a software product line architecture is proposed. This approach to architecture, in which two architectures are proposed, is used to cover the gap found in practice between traffic engineers and software engineers (see Section 3.4).

5.2 Architecture for Traffic Control

The ATC (Architecture for Traffic Control) (Vrancken et al., 1998; Taale et al., 2004; Stoelhorst and Middelham, 2006) can be used to develop and implement a set of coherent traffic management measures and the necessary technical and information infrastructure. The goal of ATC is to be a domain architecture to build RTC systems. Therefore, the ATC should be combined with a software architecture as basis to develop the software systems.

The ATC framework, showing the main parts of the architecture, is depicted in Fig. 5.2. The three horizontal layers are most relevant here. Bottom layers provide services for immediately top layers.

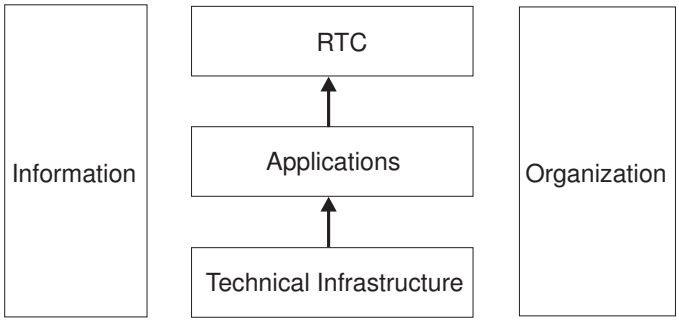

Figure 5.2: Architecture Framework for Traffic Control.

Within a country, the network is divided into a number of regions, under the control of a TMC (Traffic Management Center). Within the region of a single TMC, attention is not given to all roads and junctions homogeneously, but attention is focussed on a number of areas where traffic is more problematic than at others, such as the belt roads around cities (Vrancken and Soares, 2009a).

The Applications and Technical Infrastructure layers form the technical parts of the architecture. The Applications layer describes the applications to implement the RTC features mentioned in the RTC layer, such as the control systems for each of the local measures, and the operator's user interface and support tools in the TMC. This layer includes roadside sensors and actuators used in control systems. Applications in this layer can be developed based on a common software architecture, which is discussed in Chapter 6.

The Technical Infrastructure is about the data communications network and the processing platforms along the roads and in the TMCs. This layer also comprises a vitally important middleware layer that facilitates high-level communication between application components.

The RTC layer describes the ATC approach to RTC, in typical traffic engineering terms. This approach is typically single-agent, top-down control. RTC is one of the main activities within road traffic management, with the purpose of influencing traffic streams in order to improve traffic flow. All coordination of local measures is done via the TMCs and ultimately via the desk and the tools of human traffic operators.

The RTC layer can be expanded into the layered model depicted in Fig. 5.3. The purpose of this model is to show how *society goals* in traffic management, such as improved travel time reliability and less delay, can be translated, in five

steps, into the signals shown to drivers.

Control strategies are general principles and strategies for RTC. Examples are the prioritization of certain types of roads, such as belt roads around cities, and the restriction of slow traffic to the right-most lane (in countries with driving on the right). Control strategies hold in principle for the whole network under consideration and do not refer to any specific part of the network.

Control tactics are a set of general rules and principles to improve traffic conditions. Examples are buffering of less important traffic streams and rerouting of traffic streams to relieve more important roads. Control tactics focus on a specific region of the network.

The *Control Scenarios* layer is the layer at which operators in TMCs work. In this layer the control tactics for a specific part of the network is implemented. Control Scenarios are integrated programs of control measures that cover the area under control of the TMC. The focus of control scenarios are a specific region on specific moments in time. Scenarios are developed off-line and their execution is triggered by and responds to certain recurring patterns in traffic, such as the morning rush hour. The Control Scenario that matches best with the current traffic state is determined automatically on-line. In principle, each recurring pattern found in traffic has its own Control Scenario.

Control Measures are the usual local measures, such as traffic signals, ramp metering, speed instructions and warning signals. All coordination of local

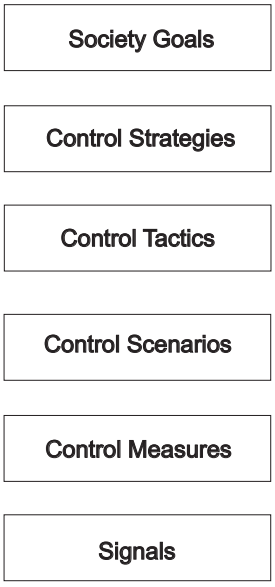

Figure 5.3: The RTC layer expanded.

measures takes place at the Control Scenarios layer. In ATC, local measures are not supposed to communicate with each other. Instead, they receive instructions top-down from the TMCs.

Finally, the *Signals* layer deals with the various visual signals present on the road to be shown to drivers (e.g., VMSs, traffic signals).

The ATC is extended in the next Sections.

5.3 Requirements for the Distributed Traffic Control Architecture

ATC has a number of limitations, as discussed in (Soares and Vrancken, 2007a), primarily caused by its top-down, single-agent nature. First of all, single-agent (also called centralized or top-down) control is known to have serious limitations in handling complex situations, in addition to limited scalability (the controlled area cannot easily be extended because that would soon overload the operator), communication overhead and limited computational capacity. Second, the scenarios are optimized to a specific traffic pattern, which may differ from real cases. In practice, the real traffic situation will always differ somewhat from the situation the scenario was developed for, making the scenario less optimal. Third, scenarios are activated by triggers which are difficult to implement in an optimal, proactive way. For instance, a scenario to prevent rush hour congestion may be less optimal if the congestion has already started caused by an accident. A fourth problem with scenarios is that they are difficult to combine. Often one is tempted to combine scenarios because of the many different situations one has to cope with in the road network.

To overcome these problems a new architecture is proposed in Section 5.4 based on additional functional and non-functional requirements presented as follows.

5.3.1 DTCA Functional Requirements

F1_DTCA - ATC should be extended to overcome its limitations.

F2_DTCA - It should be adaptive to the real traffic state thus making RTC less dependent on scenarios.

F3_DTCA - It should be able to predict the traffic state within a certain time frame enabling the proactive countering of congestion.

F4_DTCA - It should function automatically and at a lower level, thus being able to more finely apply measures (like buffering) over multiple locations in the network.

F5_DTCA - It should decentralize the control.

F6_DTCA - It should work together with ATC and existing applications.

5.3.2 DTCA Non-Functional Requirements

A number of non-functional requirements can be derived that give guidance for the additions and choices in the technical layers.

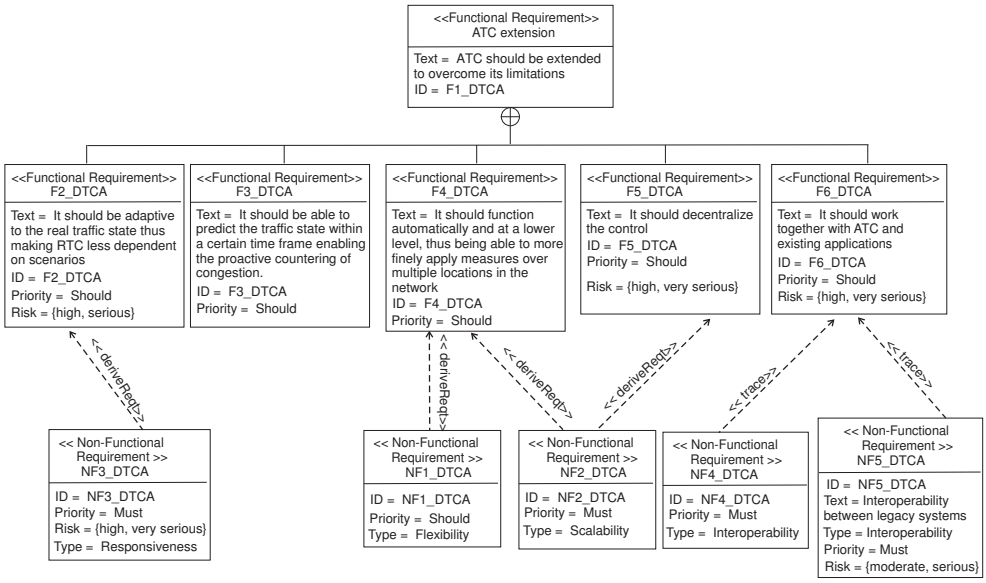

Figure 5.4: SysML Requirements diagram for DTCA

NF1_DTCA - Flexibility The way to do network-level control is still largely to be discovered, by simulation but also by real-life experiments. This means that control systems should allow frequent adaptations, which makes flexibility (more precisely adaptability) a prime requirement.

NF2_DTCA - Scalability The scalability of multi-agent control should not be hampered by any technical aspects, thus the technical part of control systems must also be scalable.

NF3_DTCA - Timely responsiveness One of the drawbacks of distributed systems is related to a possible poor performance. The performance problem is compounded by the fact that communication, which is essential in a distributed system, is typically quite slow. RTC is a form of process control. Therefore, as mentioned above, real-time properties of applications are a major concern.

NF4_DTCA - Interoperability When experimenting with different forms of multi-agent control, applications should be easy to connect to and operate with, which means that interoperability between RTC applications is important.

NF5_DTCA - Interoperability with legacy systems The presence of legacy systems along the roads, and the circumstance that it would be too costly to ignore the existing systems, implies that interoperability with legacy systems is an important requirement.

Table 5.1: Hierarchy Requirements table - F1_DTCA

Id	Type
F2_DTCA	Functional
F3_DTCA	Functional
F4_DTCA	Functional
F5_DTCA	Functional
F6_DTCA	Functional

A SysML Requirements diagram relating the functional and non-functional requirements is given in Figure 5.4.

A SysML Table for requirements hierarchy related to requirement "F1_DTCA" and a SysML Requirements Relationships Table are shown, respectively, in tables 5.1 and 5.2.

In order to fulfill these requirements, the ATC architecture is extended towards a distributed architecture (DTCA) in the next section.

Table 5.2: SysML Tables for DTCA Requirements Relationships

Id	RelatesTo	RelatesHow	Type
NF3_DTCA	F2_DTCA	deriveReqt	Responsiveness
NF1_DTCA	F4_DTCA	deriveReqt	Flexibility
NF2_DTCA	{F4_DTCA, F5_DTCA}	deriveReqt	Scalability
NF4_DTCA	F6_DTCA	trace	Interoperability
NF5_DTCA	F6_DTCA	trace	Flexibility

5.4 Distributed Traffic Control Architecture

DTCA consists of a number of additions to ATC. In the RTC layer of ATC, bottom-up control is added as an extra measure in the Control Measures sub-layer (Figure 5.3). This measure entails that network software elements such as junctions and road segments between junctions (links), but also routes and origin-destination pairs, become active agents. Agents are autonomous software entities capable of communication and cooperation (Wooldridge, 2009)

Network elements measure their own traffic state and communicate with other network elements, mostly the adjacent ones (in the case of junctions and road segments). The communication consists of requests for information about traffic states in nearby elements and of requests to take certain measures, such as reducing the inflow to the requesting element. The bottom-up control is intended as an addition to and fine-tuning of the top-down control, certainly not as its replacement.

On itself, it is not obvious why the addition of multi-agent control could be useful, as multi-agent systems are usually not very predictable in their behavior. However, there are reasons why, in this case, this addition looks promising. Due to the many local measures spread in the road network, control of traffic is already strongly distributed. The essential addition in DTCA is that it makes local measures communicate with each other. More precisely, measures are assigned to network elements and the network elements communicate with each other about the measures. This means that no options are lost and a wide realm of new control options is added. Moreover, such local, direct influencing of network elements among themselves gives a much quicker, real-time response to local changes in traffic and a much shorter control loop than via the TMC. In addition, multi-agent control, by its local nature, can easily scale to any size of network.

The inevitability of multi-agent control can be argued as follows (Vrancken and Soares, 2010). No decision making entity has infinite capacity. Actually, the capacity of both automated and of human decision making is rather limited. This means that each control entity has a limited geographical extent; in practice this may be as small as a single junction. These control areas touch upon each other at their borders, which means that all the ingredients for a typical multi-agent control setting are present: many entities, each with its own control area, and with mutual influence because the areas have common borders.

Given the addition of multi-agent control in the RTC layer of ATC, and the ensuing non-functional requirements, the following additions to ATC in the technical layers are proposed.

In the Applications layer, components are added that represent the communi-

cating network elements, such as road segments, junctions and routes. This is an immediate consequence of the choice to add multi-agent control to the RTC layer. The multiple views used to implement the software applications are discussed in Chapter 6.

In the Technical Infrastructure layer of ATC, an explicit choice is made for asynchronous Publish-Subscribe middleware (see the implementation architecture in Section 6.3). This is the most appropriate kind of middleware, given the requirements of real-time properties and scalability (Eugster et al., 2003; Fiege et al., 2006a; Lea et al., 2006). In addition, it is known for efficient use of network bandwidth. This kind of middleware can accommodate both the top-down and the bottom-up control mechanisms, in addition to the communication needed for data collection.

Summarizing, the DTCA consists of the ATC architecture with a number of additions to the horizontal layers of Figure 5.2 for multi-agent control. The network elements, such as junctions and road segments, become control agents that communicate with each other, distributing the control. The result is that DTCA is responsive to the actual traffic state as opposed to the use of fixed scenarios, which only cover a small number of possible traffic states.

5.5 Case Study: HARS

The DTCA has been applied in an experimental implementation: the HARS system (HARS: *Het Alkmaar Regelsysteem*, the Alkmaar control system in Dutch) (Eurlings, 2008). Alkmaar is a medium sized Dutch city with serious traffic problems on its belt road, which is part of a provincial north-south corridor (Figure 5.5). It consists of motorways, as well as provincial and urban roads.

The goal of HARS was therefore first of all to tackle the traffic problems in Alkmaar. As an implementation of DTCA, the goal of the system was to prove the concepts behind DTCA, especially the assumption that in RTC, a dual control strategy is stronger than just top-down control. HARS can be considered successful if it has a noticeable, positive effect on traffic in Alkmaar, if the multi-agent control plays a substantial role in this and if the system turns out to meet the non-functional requirements mentioned in Subsection 5.3.2.

Three types of sensors were used: velocity-intensity measure points, the induction loops at traffic signals sites, and a traffic simulation model called MaDAM (OmniTRANS, 2006). The induction loops and measure points are sensors that collect real-time traffic information from the environment, such as velocity and density. MaDAM acquires the information from the inductive loops and

Figure 5.5: The Belt road around Alkmaar.

velocity-intensity measure points and determines what the traffic state is on links that have no sensors of their own. Traffic signals react based on information sent by the links. They are used to reduce or increase the intensities of traffic flows by adjusting the length of green time for each road section (see Section 7.4).

HARS takes the existing local RTC measures and their systems, such as traffic signals at junctions and VMS as starting point, and adds a software overlay layer to these local measures. VMS are used for rerouting and informing drivers on the current traffic state of routes downstream of the VMS. In case of congestion, drivers will thus know the extent of the congestion that they can expect. More importantly, route information enables drivers to make a different route choice. Drivers are thus diverted away from the congested road with positive effects on its congestion.

The top-down control part is in accordance with ATC. It consists of control scenarios, defined on the basis of regularly occurring traffic patterns, such as the morning rush hour or the weekend exodus, made proactive by MaDAM. This simulation model generates traffic states half an hour ahead, in steps of 5 minutes. MaDAM also has a role in supplying data at locations where the system assumes a sensor for traffic data collection, but no physical sensor happens to be present.

In HARS, bottom-up control is performed by so-called capacity services which actually are actions configured in agents. Each network element E (links, junctions) offers, to each adjacent network element F downstream of E, the service of reducing inflow into F. More precisely, links compare the information about their traffic state with a so-called "reference framework" (Rijkswaterstaat, 2003). The reference framework describes the criteria that the traffic state on the links should ideally meet. If the traffic state differs from the criteria, links will take appropriate action. They will communicate with other links and ask them to perform appropriate services to induce a traffic state that meets the criteria. The first capacity service to be implemented is called "reduce outflow", which is offered by all links. When called upon, a link will reduce its outflow by setting different green times for the traffic signals. If the link does not have a traffic signal (or any other way to implement the service) it will forward the service-call to his upstream neighboring link(s). Also, when the link calculates that the reduction in green time (and thus in outflow) will make it miss its own reference framework, the link will send a corresponding "reduce outflow" request to its predecessors. This reactivity is one characteristic of an agent and makes the system intelligent. If there is more than one upstream link, the link will divide the request among the links. In this division, the traffic states on the upstream links as well as their priorities within the network are taken into account. First, links with remaining buffer-capacity will be allocated as much of the service request as their buffer capacity allows. Second, links with higher priority only have to realize a smaller portion of the outflow reduction.

5.6 Conclusion

RTMS are distributed real-time systems, and due to this, are systems that can benefit if developed with a well-defined architecture. The focus in this chapter was on the domain architecture.

The addition of multi-agent control to ATC leads to DTCA, the Distributed Traffic Control Architecture. A real-life implementation of a DTCA-based control system has been realized on the belt road of the Dutch city of Alkmaar, The Netherlands. This implementation can serve as a real-life test bed for multi-agent RTC.

In this chapter, an hybrid approach combining both top-down (central) and bottom-up (distributed) control strategies for RTC is proposed. The bottom-up approach will not replace the top-down approach. In fact, the bottom-up approach is intended as a supplement to the top-down approach. By applying this approach to traffic control systems, it can have advantages from both the

top-down and bottom-up approaches. As a matter of fact, the use of real-time data combined with historical statistics enables the system described in the case study to be more predictive than using off-line control scenarios only, which enhances the efficiency of traffic.

The layeredness of DTCA contributes to the flexibility non-functional requirement. The choice for publish-subscribe middleware contributes to flexibility, scalability, real-time properties and the interoperability with legacy systems. HARS illustrated that DTCA-based control systems can be deployed on the existing technical infrastructure and existing sensors and actuators, thereby contributing to the future-proofness of these parts. Only the interoperability with other DTCA-based control systems has not been demonstrated, but this is most probably easier than interoperability with legacy systems and is also taken care of by the middleware.

The next chapter focus on a multi-view software architecture used as the basis for a family of RTMS, in which HARS is included.

Chapter 6

Software Product Line Architecture for Distributed Real-Time Systems

This chapter starts by showing the importance of having a well-defined software architecture for designing distributed real-time systems (Section 6.1). Due to the system characteristics and considering the domain architecture presented in the previous chapter, the 4+1 View Model of Architecture is chosen as the architectural framework in this chapter. Section 6.2 presents the reasons why, how and where SysML is included as modeling language in the 4+1 View Model of Architecture, to be used together with UML. A Software Product Line Architecture for RTMS is presented in Section 6.3. The models for each view of the architecture are designed using a combination of UML and SysML diagrams. This architecture is the basis for the design of a family of RTMS presented as case studies in Section 6.4.

6.1 The Importance of Software Architecture

The domain architecture proposed in Chapter 5 is useful to define road traffic control systems from the traffic engineering point of view. It can be used as an input for the software architecture, and for communicating and making explicit decisions related to business. Typically the domain architecture proposes a family of systems. This is necessary in order to avoid stovepipe systems, as the target family of systems should share many assets. However, the domain architecture is too high-level to be used as a basis for detailed design. The basis for software design is better represented by a software architecture. The two

architectures should be compatible. For instance, as the domain architecture uses a layered approach for representing diverse abstraction levels, the software architecture should provide a layered approach as well.

Software architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution (ANSI/IEEE, 2000). In simple terms, the description of the architecture of a software system involves the elements that compose the system and how they interact, including the interaction points.

Software architecture is part of design, but with focus on major elements, such as major classes and their communications, patterns, frameworks, layers, subsystems, components and interfaces (Kruchten, 2003). The way these elements are distributed, communicate with each other and with the system environment is also described in the software architecture. A common misconception is that software architecture is only about structure of elements and their relationships (Kruchten, 2003). Software architecture is also about behavior and interfaces between elements.

Decisions made when defining the software architecture have long-lasting impact on the design and major quality attributes of the software, such as performance, evolution and safety. These decisions will reflect on design models and on software implementation. As a matter of fact, the architecture description is an effective communication mean between different stakeholders.

A common accepted notion for developing software is that having a well-defined software architecture description is essential (Kruchten et al., 2006). Knowing the structure of the system and the relationship between components helps in software development and to communicate decisions to stakeholders.

Software-intensive systems are rarely built from scratch (Boehm and Turner, 2003). The software architecture is useful for creating the necessary conditions to improve reusability. Reusing parts of already existing, successful systems, has high potential to deliver reliable systems.

Future systems' maintenance and evolution are facilitated when the software architecture is clear for all stakeholders (Lindgren et al., 2008). It is frequently hard to fully replace legacy systems. Therefore, software-intensive systems must be flexible to evolve, in order to improve longevity (protecting high investments) and facilitate maintainability. Empirical results and practical experience show that systems with defined software architecture are resilient to change (Booch, 2007).

Research on software architecture is rapidly growing as many researchers and practitioners recognize the importance of having a well-defined software archi-

tecture to support activities such as project management, design and implementation. On the other hand, poor software architecture definition is recognized as a major technical risk when designing a software-intensive system (Clements et al., 2002). In summary, software architecture is a critical element for developing software-intensive systems.

6.1.1 The 4+1 View Model of Software Architecture

Many architecture models were proposed in past years, such as (Soni et al., 1995; ANSI/IEEE, 2000; Leist and Zellner, 2006). Among the software architectures proposed in the literature, the 4+1 View Model of Architecture (Kruchten, 1995) is investigated in this chapter. The choice is based on practical and empirical evidence of its success, and on its own capabilities. This architectural framework is considered sufficient for most software-intensive systems (Kruchten, 2003). The HARS system (see Chapter 5) was developed based on the 4+1 Architecture.

The architecture consists of four views (Logical, Process, Deployment, and Implementation) and the plus one (Use Case) crosscutting view that integrates the other four. A view is an abstraction of a system from a particular perspective, covering particular concerns and omitting elements that are not relevant to that specific perspective. The 4+1 View Model (Fig. 6.1) proposes the organization of the description of a software architecture using five different views, each one addressing one specific set of concerns. The Use Case view can be well-described using the approach proposed in Chapter 4. The layered architectural style described in Chapter 5 for the ATC Architecture can be used again, this time in the context of the Implementation view. The diverse physical elements of the system can be described in the Deployment view. Finally, RTMS need a well-defined representation of real-time process control, which is the subject of the Process view.

Separating concerns in different views helps in diminishing the misunderstandings and incorrect interpretations when a single view of the architecture is described (Kruchten, 1995). For instance, common elements used to describe the architecture, such as blocks and arrows, may represent different elements depending on which view is described. If the semantics of the view is not clear, its elements might be incorrectly interpreted. A brief description of each view of the 4+1 View Model is given as follows.

The *Logical view* concerns the functional requirements of the software. For instance, when using an object-oriented approach, the main elements are classes and relationships, such as association, composition, aggregation and inheritance. Only relevant elements, in terms of the architecture, are represented. In

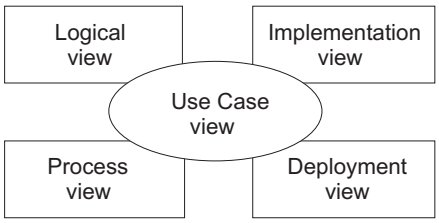

Figure 6.1: The 4+1 View Model of Software Architecture

this view, how classes cooperate with each other and the internal behavior of classes are also described.

The *Implementation view* suggests how the system is decomposed into subsystems, that are organized in layers and packages, which contain components, executables, files and source code. A layered implementation approach offers separation of concerns, from domain-specific and hardware independent (top layers) towards domain independent, hardware specific (bottom layers). This view can also help in defining parts to be built by subcontractors or internal development teams.

The *Process view* address the concurrent aspects of the software at runtime - tasks, processes, threads, as well as their interactions. Examples of interprocess communication mechanisms include synchronous communication, remote procedure calls, publish-subscribe, and event broadcast. Many aspects are addressed in this view, such as parallelism, fault tolerance and object distribution in real-time, dealing with issues such as deadlocks, response time and scalability.

The *Deployment view* shows how the various executables and other runtime components are mapped to the physical processing nodes and platforms.

The *Use Case view* contains the main scenarios that are represented as use cases. These scenarios act to illustrate in the software architecture document how the other four views (and elements) collaborate in order to realize the use case.

6.2 Adding SysML in the 4+1 View Model

The 4+1 View Model of Architecture is generic, thus it is possible to use various diagrams/languages for each view. In fact, the original paper published in 1995 used the Booch Method (Booch, 1994) to model the examples. Although not advocating a specific method or language to design the models, normally the most common modeling language associated with the 4+1 View Model is UML.

Nevertheless, important characteristics for the purpose of designing software-intensive systems are lacking in UML (OMG, 2008a). As a software-oriented language, it lacks modeling power, for non-software parts or aspects to be filled in by other languages, leading to additional integration efforts. The emphasis of UML is on the graphical modeling of functional requirements, and little attention (if any) is paid to other requirements types. For instance, there is no specific diagram to model non-functional requirements, which are proved to be fundamental for successful systems (Hofmann and Lehner, 2001; Damian et al., 2005; Minor and Armarego, 2005). The UML Activity diagram models discrete behavior, but does not have easy means to model continuous flow. In general, UML is weak on modeling other elements that are not related to software, such as hardware, machines and their parts, relationships, properties and constraints (Glinz, 2000; Henderson-Sellers, 2005).

The proposal in this section is to describe where and how SysML can be included in the 4+1 View Model of Architecture, and the possible advantages of combining UML and SysML models. Theoretically the proposal seems to be feasible due to two important facts: 1) UML and SysML are conformant to the same metametamodel, the MOF (OMG, 2006); and 2) SysML reuses and adapts diagrams and constructs from UML (see Section 2.3). In order to validate this proposal, it is applied in practice in this section and in the following. Finally, a case study is given in Section 6.4.2. This section explains which SysML diagrams are added for each of the five views, and why this is done.

Logical view: Currently, common diagrams used for creating models for this view are the UML Class and Object diagrams, for the structural aspects, and the Sequence and Activity diagrams for dynamic aspects. Three SysML diagrams are proposed for this view.

With the SysML Block diagram, elements of the system such as hardware, data, procedures, and persons can be modeled. The representation of system architecture can be made by means of blocks, without focusing only on the software structure of each system element, but also on the general structure, including parts of each block, constraints and properties not necessarily related to software. SysML Blocks are candidates to be refined into one or more UML Classes during the software design and implementation phases.

The SysML Requirements diagram (see Chapter 4) allows crosscutting with other diagrams, mapping a requirement to other design models. Relationships between requirements, and their classifications, are also represented, being useful for requirements management and to plan system releases. For instance, when a requirement is modified, it is possible to trace which requirements and models are affected, and how they are going to be affected. In addition, knowing all requirements dependencies show the stakeholders which requirements may

have priority over others and need to be implemented first. Another difficulty when using UML only is to map directly requirements into UML Use Cases (Anda et al., 2006).

The SysML Sequence diagram offers the same syntax and elements of the UML counterpart, but with different semantics. Within UML, Sequence diagrams are used to model communication between software objects, which are instances of software classes. In SysML there is no notion of instance or objects. Thus, the communication is between the main structural elements, normally blocks and subsystems. Another difference is that the messages exchanged are not methods of an object. Thus, SysML Sequence diagrams can be used when modeling general communicating elements of a system.

Process view: The Class diagram can also be used in this view, but the focus is different from the logical view. The classes in this view are active classes that represent threads and processes, such as the <<control>> stereotype for classes used within UML.

The UML Activity diagram is used with emphasis in tasks, main processes and flow of control. The SysML Activity diagram offers additional modeling possibilities when compared with the UML version. The notion of tokens flowing along edges through the activities still exists, but there is support for modeling continuous flow as well as discrete flow. A rate describing the frequency in which elements traverse an activity edge can be added to inflow and outflow edges. To each edge, <<continuous>> or <<discrete>> stereotypes can be added depending on the flow type. The flow can also contain a specific probability of occurrence of an activity/task, which describes at each outgoing edge how probable it is that the token will flow over it.

Deployment view: SysML does not provide a Deployment diagram. This decision, as well as not including other UML diagrams such as Object and Time, was taken in order to simplify the language and to avoid the notion that designers should first learn the object-oriented notions in order to understand SysML. As a SysML Block can describe general structural elements, varying from very small to very large, they can be used to represent the physical deployment nodes. As a matter of fact, SysML Blocks can be used to represent the physical architecture of the system.

Use Case view: The Use Case view (formerly known as the scenario view) is most frequently modeled with UML Use Case diagrams and natural language for descriptions. The SysML Use Case diagram is derived without important extensions from the UML Use Case diagram. The main difference is the wider

focus, as the purpose is to model complex systems that involve not only software, but also other systems, personnel, and hardware.

The SysML Use Case diagram shows system functionalities that are performed through the interaction of the system with its actors. The idea is to represent what the system will perform, not how. The diagrams are composed of actors, use cases and their relationships. Actors may correspond to users, other systems or any external entity to the system. As this view is directly related to requirements, the SysML Requirements diagram and the SysML Tables can be used to create models for this view (see Chapter 4).

Implementation view: The Package diagram is used to organize semantically coherent models by partitioning elements into packages and establishing dependencies between packages and/or model elements within the package. The same syntax and semantics are presented for both UML and SysML Package diagrams. Specific subsystems can be represented using the SysML Block diagram.

6.3 Software Product Line Architecture for RTMS

Having a software architecture is fundamental for creating the necessary conditions to improve reusability. Reusing parts of already existing, successful systems, has high potential to deliver reliable systems. Reusing source code is a well-known approach used for many years in order to facilitate the development of systems. Even more can be done regarding reuse. Better approaches should be based on a shift from opportunistic to strategic, planned reuse.

When developing a collection of similar systems that share common characteristics, principles and structure, a feasible solution is to reuse also other core assets besides source code, such as requirements, components, architecture, design and documentation. The investigated asset in this section is the software architecture for a domain specific series of applications, towards a *Software Product Line Architecture*, which is a common architecture for a family of related applications. These assets can be configured and composed in different ways to create all of the products in a line of related software. This is the core idea of Software Product Line research (Clements and Northrop, 2001; Pohl et al., 2005; van der Linden et al., 2007).

Although the idea of product lines was proposed long ago (Mcilroy, 1968; Dijkstra, 1970), it is an area of research that only recently is gaining more attention in Software Engineering (van Ommering and Bosch, 2002). The initial focus was on families of programs (Parnas, 1976), as for instance, software with dif-

ferent algorithm variants. Currently, researches are done in order to propose families of architectures, facilitating architecture reuse, which is an important asset for related systems. Well-architected systems make possible the creation of families of related systems, optimizing development. Having a common architecture serving as the basis for the design and implementation of a family of related systems is a way to address common development problems.

Few researches were done in Software Product Line Architectures in practice, and much has still to be discovered in this area. In this section, an architecture for the design and implementation of a family of RTMS is presented. The most important principles of the architecture are the layered implementation view, publish-subscribe middleware for process communication, a domain-specific language, component based development, and the object-oriented analysis and design paradigm. The architecture has been used in many projects in practice. Two projects related to intelligent road traffic control, and the visualization of road traffic data at junctions are presented as case studies in Section 6.4.

6.3.1 Domain Characteristics and Requirements

The characteristics and requirements of RTMS are important for the decisions on the fundamentals of the software architecture used for development. In addition, the decisions already described in the domain architecture in the previous chapter have to be taken into account when the software architecture is to be described.

Many non-functional requirements are important for RTMS. Scalability is of great significance because there may occur changes in the road network, which may include new sensors and actuators to be accommodated into the system. Thus, new software objects may be inserted into the software architecture each time the network grows. This is important to allow the system to evolve over time.

Performance is also an issue, as there are many components to be controlled. Real-time constraints must be observed as data should be updated regularly. Example of available data are road traffic measurements information that are geographically distributed within the network. Availability of data is of considerable influence to system reliability and overall performance.

Finally, flexibility is essential due to the possible change of component types, interfaces, and functionality, as the system evolves over time, and to cooperate with legacy systems.

Layer	Geographic element	Monitoring	Control
Network	Network Route	OD-matrix Route travel time	OD-Mgr. Route Mgr.
Link	Link merge/choice point	Capacity Avg. speed, turn fractions	Link Mgr. Junction Mgr.
Point	sensor/actuator position	Speed, flow, occupancy	Actuator Mgr.

Table 6.1: Elements of the Software Architecture

6.3.2 Logical View Architecture for RTMS

Distributed architectures improve scalability. This is an important requirement for RTMS. The logical view architecture shows which elements (see Table 6.1) cooperate with each other in a high level manner, without concerns about how this interaction is done. These network elements are:

- **Origin-Destination Managers** (ODMGR) represent the relation between an origin and a destination and comprise one or more routes.
- **Route managers** control the set of routes from one origin to one destination.
- **Links** come in two types, Main links and Accessor links. The Main link is the link from the merge point to the choice point and the Accessor link is the link from the choice point to the merge point (see Fig. 6.2).
- **Junctions** comprise the outgoing Main link and the incoming Accessor links of a crossing or motorway junction (see Fig. 6.2). A junction is a location where traffic can change its routes, directions and sometimes even the mode of travel.
- **Control Schemes** are coherent set of measures triggered by recurring patterns in the traffic state, such as the morning rush hours or the weekend exodus.

The representation of the logical architecture view is shown in Fig. 6.3 using a SysML Block diagram. Each road network element represented by a SysML Block has a direct software object representation. The distributed components have to communicate with each other, as they work in cooperation. For instance, links have to communicate with other links in order to achieve a traffic state. They continuously measure the traffic state and communicate about it to other links in real-time.

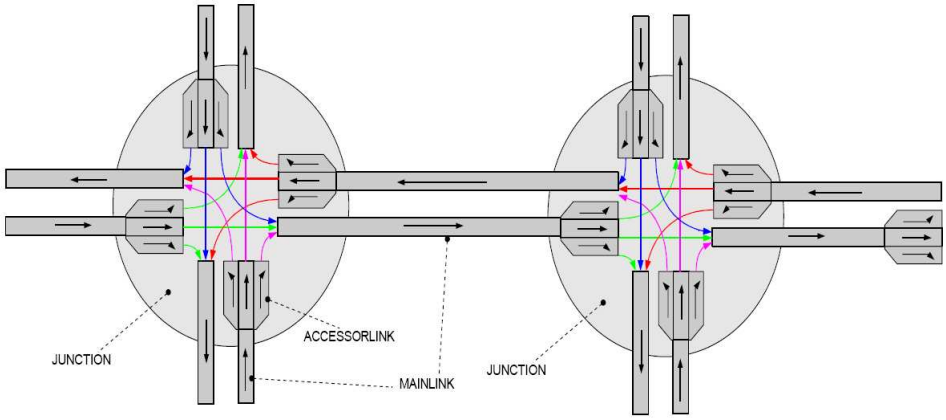

Figure 6.2: Junction style

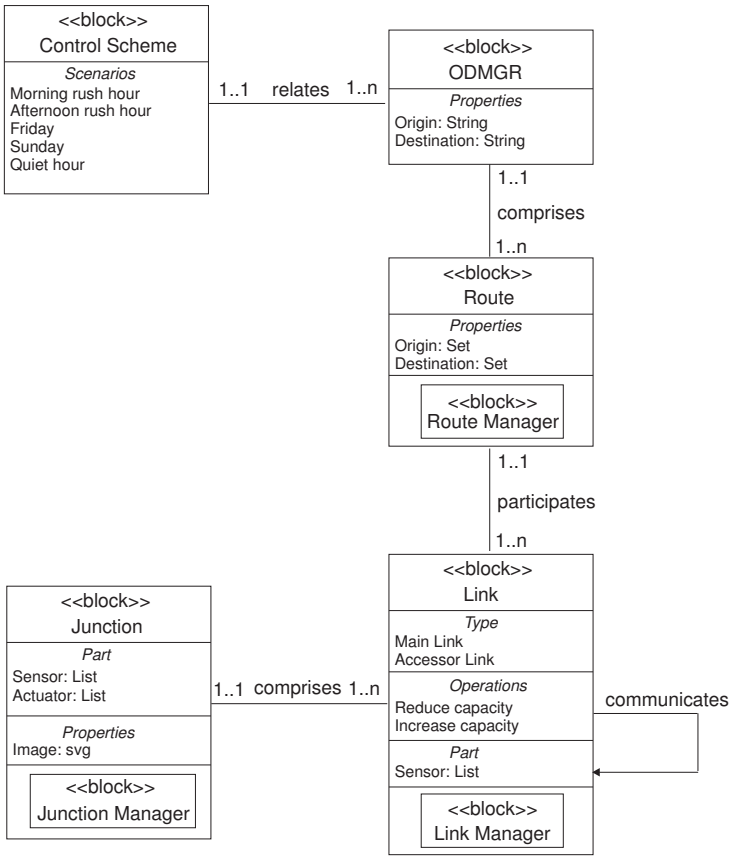

Figure 6.3: Logical view and the relationship between control elements

6.3.3 Implementation View Architecture for RTMS

The layered architecture model of implementation is depicted in Fig. 6.4 using a UML/SysML Package diagram. Layered models are ideally suited to express

important parts of an architecture, describing a way to organize systems by separating similar and related components in layers. Each layer accommodates related semantic models, providing strong separation of concerns, which is a well-known concept in Software Engineering and considered essential for software development.

There are many advantages in using a layered architecture for developing software (Garlan and Shaw, 1993; Bass et al., 2003; Vrancken, 2006). Layers show diverse levels of abstraction, from specific, top levels, towards general, lower levels, hiding details whenever convenient. Therefore, separation of concerns is addressed. Each layer can be detailed into sub-layers, and sub-layers can be combined into layers, providing flexibility for development. Finally, regular changes may be easily implemented when they are related to a specific layer instead of to a whole system.

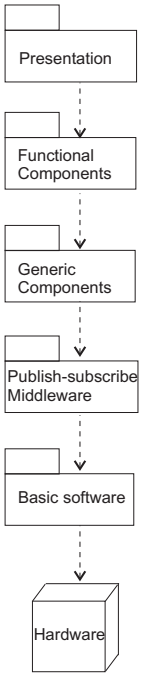

Figure 6.4: Implementation View of the Software Product Line Architecture

The Presentation layer is used mainly to present the applications results, and can be made for diverse devices/platforms. Developers may use a list of existing Generic Components and combine them as much as necessary to help in building Functional Components, which implement user requirements. The models of the Functional Components layer are designed following the object-oriented paradigm and UML as modeling language.

A domain-specific (Mernik et al., 2005) object-oriented programming language

is used to implement the object-oriented design. Domain-specific languages are restricted to a particular domain or problem. The advantages of using a domain-specific language in this case are the higher levels of abstraction that developers can work with and the possibility of adapting the language for specific purposes related to traffic domain. As a matter of fact, higher productivity and reusability are achieved.

Publish-subscribe middleware layer

The environment characteristics (distributed elements in a network) imply a need for a distributed software architecture, and the real-time constraints, which calls to high performance, are important non-functional system requirements. Not only having correct data is important, but also the availability of data. Respecting real-time constraints is difficult as sensors and actuators are physically distributed, which increase the possibilities of missing data or receiving data outside of the bounded time. This indicates the importance of using an asynchronous middleware to coordinate communication and data exchange between software objects.

Distributed software objects in a network must communicate with each other by asking and providing services. There are many different means of implementing process communications (Tanenbaum and van Steen, 2006). The simplest is by using synchronous communications, based on point-to-point and blocking communication between processes. This type of communication presents many problems (Lea et al., 2006). As processes have to wait for a response in order to continue further operation, performance is reduced. In addition, there is lack of transparency and flexibility, as it is necessary to know which process to call for a given service. These requirements can be refined into the following atomic ones:

- The middleware shall offer a communication framework to make communication less primitive.
- The middleware shall hide implementation details of communicating processes.
- The middleware shall decouple communicating processes in time and in space.
- The middleware shall prevent blocking of processes during communication.
- The middleware shall offer mediation between communicating processes, such that they do not need to know each other.

- The middleware shall make it possible to be redundant in producers of information.
- The middleware shall make it possible the interoperation between legacy systems.

The selected communication mechanism should provide asynchronous communication with decoupling of processes in time and space (Eugster et al., 2003), hiding implementation details of communicating processes, preventing blocking of processes during communication, and offering mediation between communicating processes. Thus, the required middleware is of the Publish-Subscribe type (Fiege et al., 2006b; Eisenhauer et al., 2006).

Publish-subscribe middleware provides a level of abstraction, by hiding the complexity of a variety of platforms, networks and low-level process communication. Application developers may concentrate on the current requirements of the software to be developed, and use lower-level services provided by the middleware when necessary.

The basic functioning of a generic publish-subscribe middleware is given in Fig. 6.5. Components may subscribe for information, unsubscribe, publish and consume information. Components may also notify that they are interested in some kind of information. The Event Handler receives all these events through an interface, and can also notify components when the information is of relevance to them.

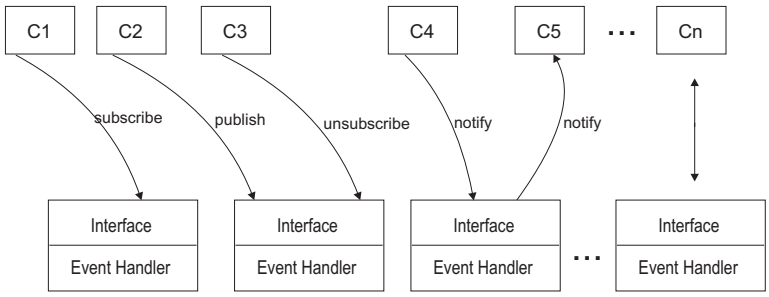

Figure 6.5: Publish-subscribe middleware scheme

The publish-subscribe communication mechanism naturally supports an asynchronous (non-blocking), many-to-many communication between components in a network. The notion of clients and servers exchanging messages is substituted by an event based communication between components that may act as publishers of information and/or subscribers for information. Publishers (acting as servers), publish information through an event, which will be delivered to all (and only) interested subscribers, which expressed their interest in

a certain type of information by subscribing to it. This allows improved system performance. Publishers are not aware of which particular subscriber will receive the published information. In a similar manner, subscribers are indifferent to which specific publisher produces the information. They even do not need to run on the same machine. This means that publishers and subscribers are space-decoupled (Eugster et al., 2003). Another important characteristic is that publishers and subscribers are fully decoupled in time (Eugster et al., 2003): publishers and subscribers do not need to be connected at the same time. All this manners of decoupling (synchronization, space and time decoupling) increases scalability and reduces the necessity of coordination, which makes publish-subscribe middlewares most suited to distributed environments, fulfilling the specified requirements of RTMS.

6.3.4 Process View Architecture for RTMS

The Process view is described with SysML Activity diagrams in the case study in Section 6.4.2. In Chapter 7, Petri nets are applied in the design of models for the process view.

6.3.5 Deployment View Architecture for RTMS

The Deployment view is described with SysML Block diagrams in Fig. 6.6. Trinivision is the graphical user interface developed by Trinité [1]. Traffic operators working in Traffic Management Centers use Trinivision in their daily activities, such as for traffic monitoring.

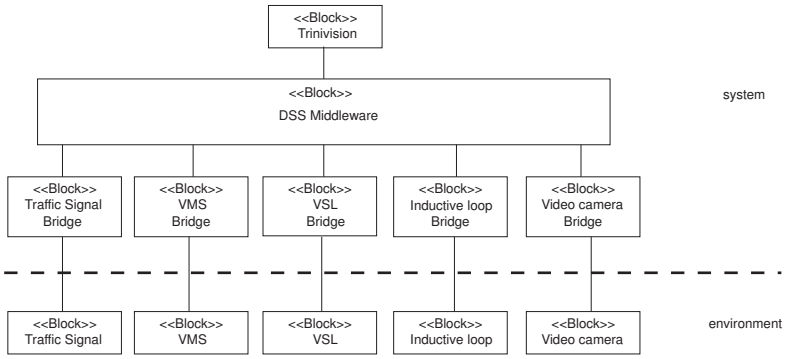

Figure 6.6: Deployment view

DSS Datapool is the publish-subscribe middleware, also developed by Trinité.

[1] www.trinite.nl

Each real world, physical entity has a correspondent software component named "bridge". Bridges are used as communication links between the environment and the system. The purpose of a bridge is to transfer updated data (e.g., status change, error report) from the physical object to the software object, which will deliver data to the DSS Datapool. In a similar manner, commands that are generated from the DSS Datapool to the software object will be transmitted via bridges to the physical objects.

6.4 Case Studies

Two systems of a RTMS product line are described as case studies as follows.

6.4.1 HARS revisited

In the HARS project, the functions of the ODMGR are enhanced and extended to enable more effective rerouting using the Route Manager. Top-down control is performed by Control Schemes, which are developed off-line and correspond to recurring patterns in the traffic state, such as the Morning Rush Hours or the weekend exodus (Friday). This is the top-down approach, in which the traffic management center (the top) is the only entity allowed to take decisions.

The bottom-up control is implemented by means of communicating agents (links and junctions). The control is performed by so-called *capacity services*, (Reduce capacity, Increase capacity) which actually are operations configured in links. By using capacity service calls, links (Main links or Accessor links) are able to offer and ask for capacity from downstream, upstream links respectively. Services are implemented in various ways, according to the available local measures in the network element: a link may implement it by reducing the maximum speed or closing one or more of its lanes; a junction may implement it by reducing/extending green times of traffic signals.

The asynchronous publish-subscribe communication is used as the middleware layer. The publish-subscribe style of communication reduces the dependency among the components and make the information available where and when needed. This should also reduce communication overhead between agents, since it is not necessary for the components to continuously request data and wait for response, which blocks the components. As a matter of fact, important quality factors for the system such as performance, scalability and flexibility are improved using this communication style.

6.4.2 Visualization of Junction Measurements

This case study is about a system to visualize measurements at junctions (Soares et al., 2009b,c). Measurements obtained by sensors and the effects of the application of actuators are important quantities to be visualized for decision makers. Thus, having application systems that can visually show numerical values resulting from measurements is fundamental. The system provides visualization of road traffic measurements at junctions controlled by traffic signals. It can be seen as a decision support tool for traffic operators, providing reliable, real-time information of road traffic measurements, which can be useful to improve traffic signals planning. This application will be used as a subsystem for a RTMS.

	←	↑	→
waiting time			
velocity			
intensity			
density			

Figure 6.7: Table of measurements

For each direction of the Accessor link (turn left, go ahead, or turn right), information about velocity, waiting time, intensity and density is shown in the system (Fig. 6.7). This information is a result of data gathered by sensors distributed in the network. Availability and reliability of data is achieved due to the publish-subscribe communication layer.

An automated image of the junction is created based on links information. This image and a table with measured data are shown in a graphical user interface (the presentation layer). Clicking in the junction image will result in an overview from this image containing the information presented in the tables. The full list of requirements is shown in Appendix A.

Figure 6.8: Use Case diagram

The design and implementation were performed following the object-oriented

paradigm. TriBASIC, a domain-specific object-oriented programming language is used to implement the design.

The models for each view of the 4+1 View Model of Architecture with UML and SysML are described as follows.

Use Case view

For this view, the used diagrams are the UML/SysML Use Case (Fig. 6.8), the SysML Requirements diagram (Fig. 6.9), and the SysML Table (Fig. 6.10).

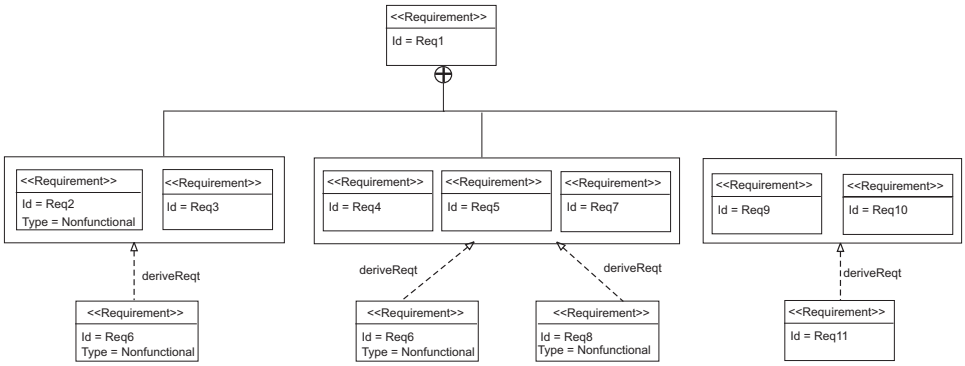

Figure 6.9: SysML Requirements diagram

ID	Relates to	Relates how	Type
Req 12	Req 2, Req 3	deriveReqt	Functional
Req 6	Req 4, Req 5, Req 7	deriveReqt	Non Functional / Interface
Req 8	Req 4, Req 5, Req 7	deriveReqt	Non Functional / Interface, Quality
Req 11	Req 9, Req 10	deriveReqt	Functional

Figure 6.10: SysML Table

Process view

A SysML Activity diagram (Fig. 6.11) is used to describe the Process view.

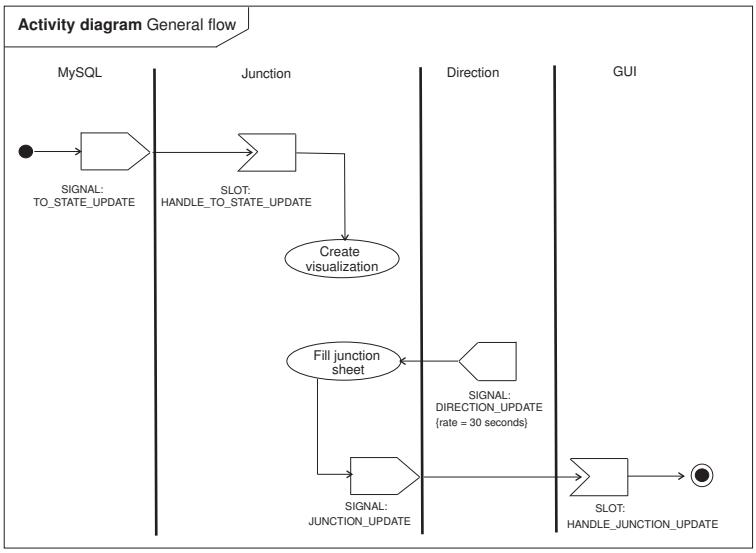

Figure 6.11: Activity diagram: General Flow

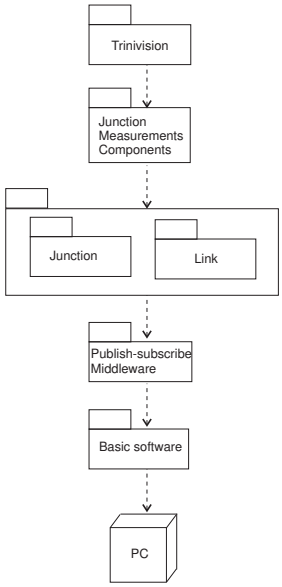

Figure 6.12: Implementation view for the Case Study

Implementation view

For this view, UML/SysML Package diagram is used (Fig. 6.12).

Logical view

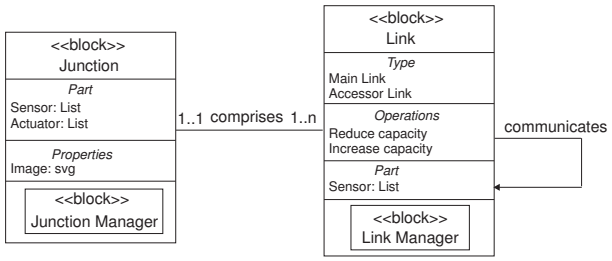

Figure 6.13: SysML Block diagram for the Case Study

Figure 6.13 depicts the specific part of the SysML Block diagram necessary for the case study.

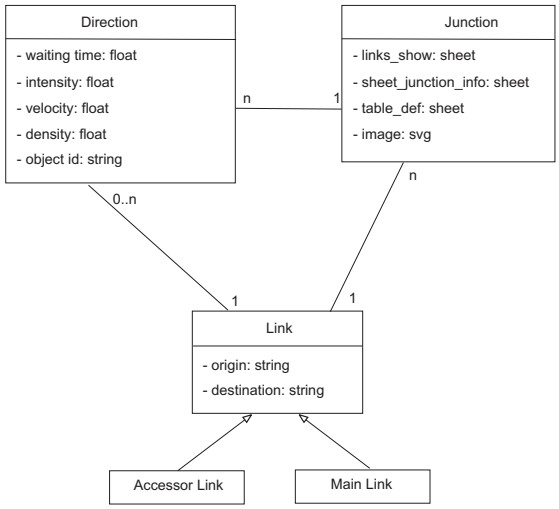

Figure 6.14: Class diagram for the Case Study

Part of the UML Class diagram is shown in Fig. 6.14. Junction, Link and Direction are components of the Generic Components layer (see Fig. 6.4) that are extended with new functionality to build the application. The SysML Sequence diagram is depicted in Fig. 6.15.

Deployment view

A SysML Block diagram (Fig. 6.16) is used to describe the Deployment view specific for the case study.

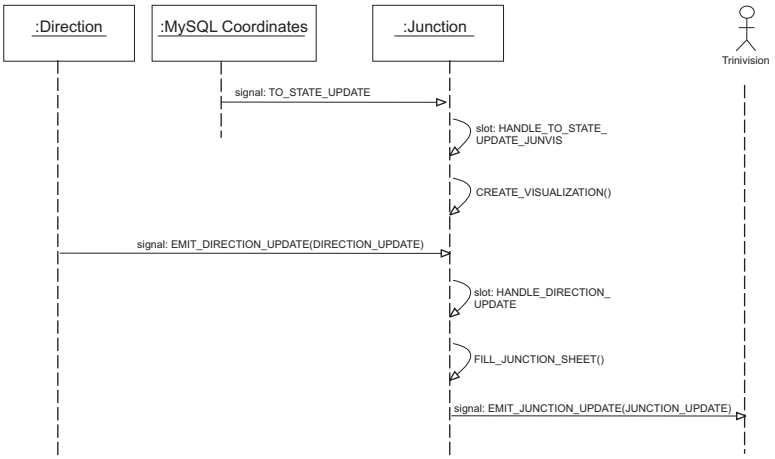

Figure 6.15: Sequence diagram for the Case Study

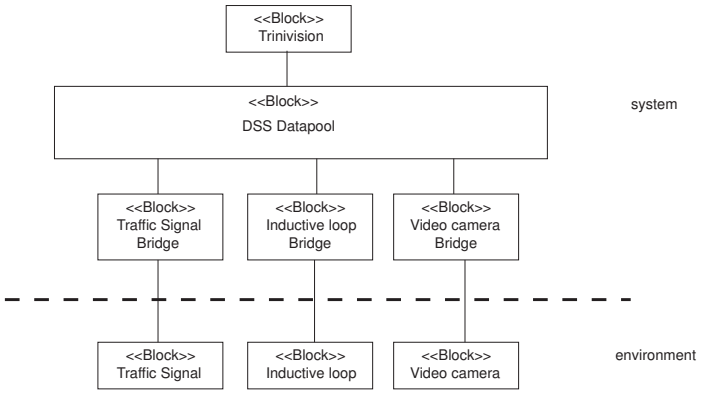

Figure 6.16: Deployment diagram of the case study

6.5 Conclusion

This chapter is about software architecture for distributed real-time systems. It presents the result of how and where to introduce a new modeling language, the SysML, into an existing model of architecture, the 4+1 View Model. The proposal is to combine UML and SysML to create models of each one of the five views of the architectural framework. A Software Product Line Architecture for RTMS is presented. This architecture uses SysML and UML, which is useful to create the design models of the case studies.

The SysML Block diagram is useful to model components of a system, such as hardware and its parts. Each structure, component or part of the system is represented by a SysML Block. Relationships between blocks are represented

as well using the SysML Block diagram. When combined with UML, a SysML Block can be refined into one or more UML Classes.

The behavior of the elements of the software architecture are modeled using SysML Activity and Sequence diagrams. The SysML Activity diagram offers the possibility of adding probabilities and rates to flows. The SysML Sequence diagram is more general than the UML Sequence diagram. It models communication between elements of a system (subsystems, components, parts) represented by SysML Block diagrams.

The combination of publish-subscribe middleware, layered implementation architecture, object-oriented analysis and design, components framework, and domain-specific language for applications development has been applied in many RTMS, and has provided high abstraction levels, tackle their complexity, and fulfill the domain requirements. Two systems were developed using the Software Product Line Architecture presented in this chapter and were presented as case studies.

The first case study was already presented in Chapter 5, but in this chapter the focus is on the software architecture, not the domain architecture. Together with the second case study, they are part of a product line of RTMS. The product line has one single architecture as the basis for design and implementation. As a result, it was recognized that having a unique architecture for a Software Product Line presented advantages. Reusing artifacts is facilitated in general, with source code and design the main cases. As a matter of fact, planned reuse leads to higher productivity and faster development.

All models created during the design of the case studies presented in this thesis so far were created based on semi-formal languages. The next chapter proposes the application of a formal language for the design of software-intensive systems. The main reasons are to increase formality, for instance, in the representation of time constraints, and to allow better verification and validation activities. However, the choice for a formal or semi-formal language does not exclude the other. In fact, a combination of languages is possible.

Chapter 7

Formal Design and Verification of Traffic Signals Control

In this chapter, the formal modeling and the verification of urban traffic signals control are described. The focus in this chapter is to model the dynamics of traffic signals as a Discrete Event System (Section 7.1). The choice is to use Petri nets with extensions to increase modularity and describe real-time constraints (Section 7.2). Two design strategies using Petri nets are briefly explained in Section 7.3: top-down and bottom-up. These strategies and the component model are useful to improve modularity when creating models for three examples of urban traffic signals control in Section 7.4. Model verification is given, first using Invariant Analysis in Section 7.5, and then using a theorem-prooving approach in Section 7.6, in which specific scenarios are studied. The chapter ends with conclusion in Section 7.7.

7.1 Modeling of Urban Traffic Signals Control as Discrete Event Systems

Traffic control in urban roads is a major area of ITS research. Traffic signals at road intersections are the major control measures applied in urban networks. The behavior of a traffic signal can be modeled as a Discrete Event System (DES). These systems are often large, distributed systems in which events occur at specific instants of time (Cassandras and Lafortune, 1999). These events are triggered by specific conditions, such as the starting of a process. Typically, DES involves relationship with the environment, conflict for resources

and simultaneous occurrence of activities.

From a DES point of view, a road junction (intersection) can be seen as a resource that is shared by vehicles at certain points of time. The design of the control logic of a traffic signal must take care of efficiency and speed, but also of safety and security. The main purpose of traffic signals is to provide safe, efficient and fair crossing of the junction. When properly installed and operated, traffic signals provide a number of benefits (Roess et al., 2003):

- the capacity of critical junction movements is increased;
- the frequency and severity of accidents is reduced;
- nearly continuous movement along a road section at a designated speed and under favorable traffic conditions;
- safe crossing of pedestrians and vehicles by providing interruptions in heavy streams.

However, when poorly designed, traffic signals can cause excessive delays when cycle lengths are too long, increase the number of accidents (especially rear-end collisions), violations of red light, and lead to sub-optimal rerouting by drivers who want to avoid traffic signals. In addition, a set of eight minimal warranties need to be fulfilled in order to determine whether or not the installation of a traffic signal is justified (Roess et al., 2003). For instance, traffic engineers have to study the physical and geometric features of the junction, which includes channelization, grades, parking locations, driveways and bus stops.

One classification for traffic signals control is the distinction between traffic controllers that operate in pre-timed (fixed-time) (e.g. TRANSYT (Robertson, 1969)) or in responsive (actuated) mode (e.g. SCOOT (Hunt et al., 1981)). Within fixed-timed controllers, the phase duration for each road section is determined off-line based on historical data. This simplifies the implementation of the control strategy. The drawback of fixed-time traffic signals is the fact that their settings are based on historical rather than real-time data. This may be a crude simplification, because the demands are not constant, varying during the day, and influenced by special events or weather conditions. Thus, pre-defined scenarios often do not reflect the actual traffic state. In addition, the demand changes in the long term, leading to "aging" of the optimized settings. This is natural, as the number of vehicles and the need for mobility within a region change continuously. In big cities, with thousands of traffic signals installed, it is difficult to constantly update the scenarios in order to make it more realistic according to current traffic flow.

Traffic-responsive strategies are more efficient, as they use real-time measurements gathered by detectors that provide real demand information to the controller. They can change the length of the green phase (green time) for a road section depending on demand. One example of a traffic-responsive application is to verify if no vehicle passes the detectors during the minimum green time specified. In this case, the traffic signal controller switches to the next phase. Another example is the possibility of extending the green time by a minimum value while vehicles are being detected. This is done until the maximum green time is reached. Traffic-responsive approaches are more costly, as they require the installation, operation and maintenance of a complex distributed real-time control system consisting of software, communication lines and sensors. However, they are cost-effective over long time intervals (five to ten years), due to their ability to adapt to changes in traffic-flow patterns (Klein, 2001).

7.1.1 Related Work

When designing distributed real-time systems for critical infrastructures, such as transportation and energy networks, system complexity is increased due to the large number of elements, strict real-time constraints, and reliability factors, as these systems involve human life. Therefore, it is necessary to use methods that are capable of addressing complexity and providing highly reliable solutions.

A number of approaches have been used to model distributed real-time systems in general, and traffic control systems in particular. Finite State Machines (FSMs) have been used for modeling distributed real-time systems (Cassandras and Lafortune, 1999), but have the shortcoming of state explosion (Brave, 1993). Statecharts (Harel, 1987) extends state-machines by endowing them with orthogonality, depth, and synchronization. An approach to model urban traffic signal control using Statecharts was proposed in (Huang, 2006). Artificial Intelligence techniques such as Agents (Deng et al., 2005; Srinivasan and Choy, 2006) and Fuzzy Logic (Zeng et al., 2007) have also been applied to model traffic signals control. Discrete Event System Specification (DEVS) (Zeigler et al., 2000) was used to generate optimal signal times based on the temporal sequence of traffic signal control variables and traffic information collected for each intersection (Lee and Chi, 2005).

Petri nets are used in this chapter. The choice is based on their modeling expressivity, their facilities for validation and verification activities, the availability of computer tools, and the range of available extensions (Girault and Valk, 2002; Hruz and Zhou, 2007). Petri nets are suitable to model distributed real-time systems because they offer representation of conflict situations, shared resources, synchronous and asynchronous communications, and precedence constraints.

As a formal language, Petri nets allow formal checks of desirable properties. With computer tools, the verification and simulation can be easily performed. The formal analysis of a Petri net model can reveal design flaws. For instance, with reachability analysis it is possible to find out whether an unsafe state that could cause an accident can be reached. The extensions proposed in this chapter to the basic Petri net model are useful to address complexity and model real-time constraints.

Petri nets are a well-known formalism applied in traffic control research. With focus on only one intersection, (Lin et al., 2003) described the traffic signal control using Petri nets, and then used an approach via programmable logic controller (PLC) to implement the control logic. The evaluation and performance analysis of a traffic signal control by means of Petri nets was described in (Tolba et al., 2003). An approach based on the application of Petri nets and theorem proving to formally prove the good properties of traffic signals controlling one road intersection is proposed in (Soares and Vrancken, 2007d).

Considering networks of roads, one of the first applications of Petri nets for traffic signals control in urban networks was shown in (DiCesare et al., 1994), in which models of Petri nets were used to do performance analysis and source code generation. (List and Cetin, 2004) used Petri nets to model the control of signalized intersections, and evaluated the good properties of the system by means of invariant analysis and simulation. (Febbraro and Giglio, 2006) proposed a modular, deterministically-timed Petri nets in order to microscopically model a signalized urban area. A modular framework based on colored timed Petri nets to model the dynamics of signalized traffic network systems was presented in (Dotolia and Fanti, 2006).

This chapter proposes to represent phases such as green time using an interval of minima and maxima values. This is done by using the p-time Petri nets extension, in which an interval is associated to each place. A comparable approach has been applied in a rail and road control system for safety crossing a shared intersection (Dutilleul et al., 2006), but the paper does not use the places to represent phases. Some works propose the control logic with performance purpose (Tolba et al., 2003; Febbraro et al., 2004) which is not the focus of this chapter. However, in many of the previously cited works, verification of the control logic is poor or nonexistent.

7.2 Petri Nets and Extensions

Petri nets (Petri, 1962; Reisig, 1985; Murata, 1989) are a graphical, formal method applicable to a large variety of systems in which concurrency, dynamic behavior, synchronous and asynchronous communication, and resource sharing

have to be modeled. The basic Petri net is composed of four elements: places, transitions, arcs and tokens. Due to their representational power, the elements can express various roles. For instance, places normally represent conditions, status, states or operations. Transitions are generally used to represent starting or stopping of events, which occurs to change the status of places. Arcs are the connections between places and transitions, and tokens can represent number of elements or the current availability of resources.

The formal definition of a *Petri net structure* and of a *Petri net* (Murata, 1989) are given as follows:

Definition 7.1 A *Petri net structure* is a 4-tuple $N = (P, T, Pre, Post)$, where*:*

$P = \{p_1, p_2, ..., p_n\}$ is a finite set of places*;*

$T = \{t_1, t_2, ..., t_m\}$ is a finite set of transitions*;*

$Pre : (P \times T) \rightarrow \mathbb{N}$ is an input function that defines directed arcs from places to transitions, and $\mathbb{N}$ is the set of nonnegative integers*;*

$Post : (P \times T) \rightarrow \mathbb{N}$ is an output function that defines directed arcs from transitions to places;

Definition 7.2 A *Petri net* is a 2-tuple $PN = (N, M_0)$, where*:*

N is a Petri net structure, and

$M_0 : P \rightarrow \mathbb{N}$ is the initial marking.

If $Pre(p, t) = 0$, then there are no directed arcs connecting p to t. In a similar manner, if $Post(p, t) = 0$, there are no directed arcs connecting t to p. If $Pre(p, t) = k$ (the same holds for $Post(p, t) = k$), then there exist k parallel arcs connecting place p to transition t (transition t to place p). Graphically, parallel arcs are represented by a simple arc with a number k as label and representing the presence of k multiple arcs (this is also called the *weight* k of an arc).

The dynamic behavior (execution) is described by changing markings (from M to M'), which is represented by the firing of an enabled transition. A transition t is enabled if each place P that has an input arc to t contains at least the number of tokens equal to the weight of the arcs connecting P to t ($M(p) \geq Pre(p, t)$ for any $p \in P$).

The flow of tokens represent state changing. The firing of an enabled transition t removes from each input place p_i the number of tokens equal to the weight of the directed arc connecting p_i to t, and deposits in each output place p_o

the number of tokens equal to the weight of the directed arc connecting t to p_o. Therefore, the firing of a transition yields a new marking $M'(p) = M(p) - Pre(p,t) + Post(p,t)$ for any $p \in P$. The matrix $C = Post - Pre$ is called the incident matrix of the corresponding Petri net.

A graphical representation of a Petri net and its execution is shown in Fig. 7.1. Initially, places PA with two tokens and PB with one token are enabling transition t1. After the firing of transition t1, the tokens in PA and PB are removed, and one token is deposited in place PC, as the weight of the arc from t1 to PC is one.

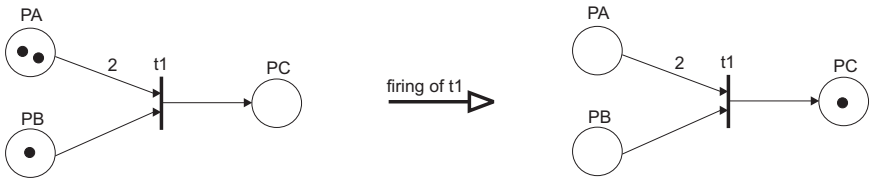

Figure 7.1: Execution of a Petri net

7.2.1 Properties of Petri Nets

The failure of software may have catastrophic consequences. For the correctness of a system, it is important that its design models are correct. Different concepts of correctness exist. For instance, a system is said to be correct when the specification and the actual implementation are equivalent (Pomello et al., 1992). Another definition for model correctness, the one considered in this thesis, is that a model is correct when it exhibits a set of desirable properties (Girault and Valk, 2002). These desirable properties allow the designers to identify the presence or absence of functional properties of the system under design.

A complete list of Petri net properties can be found in (Hruz and Zhou, 2007, chapter 8) and (Reisig, 1985; Murata, 1989). This subsection only briefly describes some of the most common ones.

- Reachability: In a dynamic system, it is fundamental to verify whether the system can reach a specific state as a result of a specific processing. In a Petri net, transitions firing will result in a sequence of markings that are reached with the change of token distribution. A marking M_f is said to be reachable from a marking M_i if there exists a sequence of firings that transforms M_i to M_f. The set of all possible markings reachable from M_i in a Petri net N is denoted by $R(N, M_i)$. The reachability problem for

Petri nets is the problem of finding if $M_f \in R(N, M_i)$ for a given initial marking M_i.

- Reversibility: A Petri net is reversible if for every reachable marking $M \in R(N, M)$, the initial marking M_0 is reachable from M. Thus, in a reversible Petri net, it is always possible to return to the initial marking or state.
- Boundedness: Given k a nonnegative integer, a Petri net is k-bounded if the number of tokens in each place does not exceed k for any marking reachable from M_0, i.e., $M(p) \leqslant k$ for every place p and every marking $M \in R(N, M_0)$.
- Safeness: A Petri net is safe if it is 1-bounded.
- Liveness: A Petri net is live if for every marking M that can be reached from M_0, it is possible to fire any transition of the net by progressing through some further firing sequence. This means that a live Petri net guarantees deadlock-free operation, no matter what firing sequence is chosen. In a Petri net model of a system, a deadlock is a marking M for which no transition is enabled. The fact that a Petri net is live ensures that for any reachable marking, at least one transition can be fired.

An important feature of Petri nets is the possibility of verifying the correctness of the designed model, i.e., to prove the presence or absence of properties. There are many methods to verify Petri nets, and these can be classified in different manners. A proposed classification of analysis methods given in (Girault and Valk, 2002, Chapter 13) is as follows: i) Graph-theory, ii) Linear Algebra, iii) State-based methods, iv) On-the-fly verification, v) Partial order methods and vi) Logic. The ones used in this chapter are Linear Algebra, in which the structure of the Petri net is analyzed (Section 7.5), and Logic, using theorem proving based on Linear Logic (Section 7.6).

7.2.2 P-time Petri Nets

In order to improve Petri nets' applicability to model systems, extensions to the basic theory were proposed (Wang, 1998; Bause and Kritzinger, 2002; Jensen and Kristensen, 2009). The extensions used in this chapter are explained in this and the following subsection.

Events that occur in actual systems modeled by a Petri net will possess an event time, which is a value from some discrete or continuous time domain (TD). It may be assumed that TD is totally ordered and has a minimum element 0 and addition operator +. A possible choice for TD is $\mathbb{Q}^+$.

The p-time Petri net (Khansa et al., 1996) model used in this chapter adds a time interval to each place, as expressed by the following definition.

Definition 7.3 A *p-time Petri net* is a pair (PN, Ip) where PN is a Petri net and Ip is the application defined as $Ip : P \rightarrow (\mathbb{Q}^+ \cup 0) \times (\mathbb{Q}^+ \cup \infty)$.

With each place $p_i \in P$ an interval $Ip_i = [\alpha_i, \beta_i]$ is associated, with $0 \leq \alpha_i \leq \beta_i$. The static interval Ip_i express the time interval during which a token in p_i is available. Before α_i, the token in p_i is in the non-available state. After α_i and before β_i, the token in p_i is in the available state for transition firing. After β_i, the token in p_i is again in the non-available state and cannot enable any transition anymore. This token is named "dead token", and can be seen as a time constraint that was not respected. The dynamic evolution of a p-time Petri net model depends on the marking M and on the availability of tokens (non-available, available, or dead). The definitions of a visibility interval and enabling interval of a transition for p-time Petri nets are given as follows (Khansa et al., 1996).

Definition 7.4 A *visibility interval* $[(\delta_p)min, (\delta_p)max]$ associated with a token of a place p of a p-time Petri net defines*:*

- *the earliest date* $(\delta_p)min$ when the token in p becomes available for the firing of an output transition of *p;*
- *the latest date* $(\delta_p)max$ after which the token becomes non-available (dead) and cannot be used for the firing of any transition.

Definition 7.5 If a transition has n input places and if each of these places have several tokens in it, then the *enabling interval* $[(\delta_p)min, (\delta_p)max]$ of this transition is obtained by choosing for each one of these n input places a token, the visibility interval associated to this token, and making the intersection of all the obtained visibility intervals.

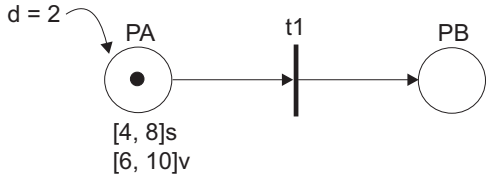

Figure 7.2: P-Time Petri net example

The Petri net of Fig. 7.2 illustrates these concepts. The static (initial) interval associated to the input place PA is [4,8]. It is supposed that the token in PA is

produced at date d = 2. Consequently, the corresponding visibility (dynamic) interval is [4+2,8+2]=[6,10] for the token in PA, which is the enabling interval for transition t1.

7.2.3 Petri Net Components

Large systems are composed of smaller parts, named components, that perform more specific functionalities. With the composition of several components to form the complete system, the complexity is increased and new properties emerge that do not belong to any specific component. For instance, performance is often related to a group of components rather than to just one component.

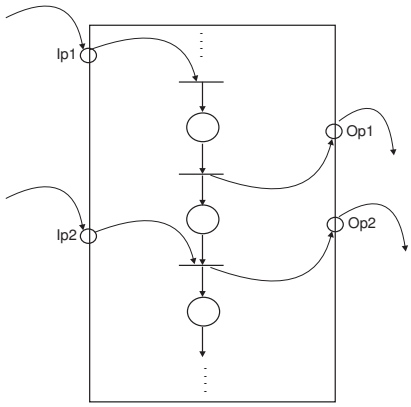

Figure 7.3: Petri net Component

The component proposed in this chapter is composed of behavior and interface. The behavior is specified by Petri nets. The component receives and sends messages through the interface. The interface is composed of specific input and output places of the Petri net. Components share only their interfaces. The behavior is not known for other components (similar to encapsulation and information hiding in object-oriented design). This characteristic of a component improves flexibility that allows changes without the need to alter the entire system. The definitions of components and interfaces used in this chapter are given as follows.

Definition 7.6 A *Petri Net Component (PNC)* is a tuple $PNC = (N, IP, OP)$, where PN is a Petri net, $IP = \{Ip_1, Ip_2, ..., Ip_n\} \subset P$ is the set of input places, and $OP = \{Op_1, Op_2, ..., Op_n\} \subset P$ is the set of output places.

Definition 7.7 An *Interface of a Petri net Component (IPNC)* is the union of the sets IP and OP: $IPNC = OP \cup IP$.

Graphically, a component is a box with internal behavior specified by a Petri net and with communication places positioned on the border (Fig. 7.3). The PNC receive data from input places (IP's), perform the specified processing, and then send processed data to output places (OP's). Interface arcs are a type of arc. They connect the outside world with input places of the PNC, and the PNC with the outside world.

The interfaces between components should always be consistent in order to allow modularity by substitution of Petri nets. The other advantage of modularity is that components can be simulated and verified separately. Although this does not guarantee that the global Petri net is going to work properly as the single components at first, an approach in which components are assembled in a controlled manner may help in discovering defects that may be caused by the last component included.

7.2.4 Petri Net Metamodel

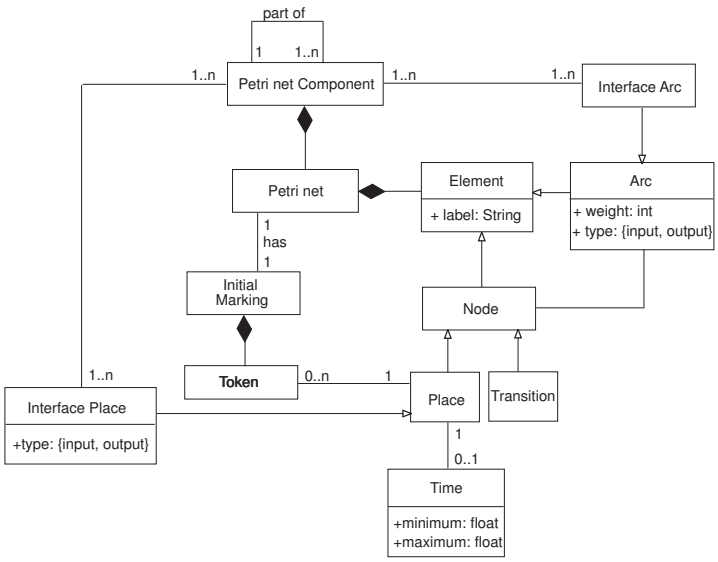

Figure 7.4: Petri net metamodel

As a result of the extensions, the Petri net metamodel used in this chapter is depicted in Fig. 7.4. The metamodel does not represent the execution of a Petri net. It represents the basic elements of the Petri net model used in this chapter at a specific situation, which involves tokens and markings.

7.3 Design Strategies using Petri Nets

In order to address software-intensive systems complexity, their design must be performed in a modular way, in which parts of the system, the subsystems, and even parts of the subsystems, can be specified and taken into account separately. In Software Engineering in general, modularity is a well-known principle, being successfully applied in different domains for many years (Parnas and Clements, 1986; Jackson, 2003; Sommerville, 2007). Modularity provides a mechanism for realizing the principle of separation of concerns (Pressman, 2005).

Two strategies are most common in the design of Petri net models in a modular way: composition, which is named bottom-up, and refinement, named top-down (Vogler, 1992). Transitions as well as places can represent sub-nets, allowing top-down modeling at several levels of detail (Suzuki and Murata, 1983; Christensen and Petrucci, 2000). In addition, it is feasible to construct large models by composing Petri nets, using a bottom-up approach (Girault and Valk, 2002). The modularity in this chapter is relative only to places, being it the refinement or the composition.

7.3.1 Top-down

The top-down modeling strategy decompose large systems into subsystems. It begins with an aggregate model of the system, which is then refined progressively to introduce more details (Suzuki and Murata, 1983; Zhou et al., 1992).

There are several advantages to use refinement and hence to start with a more abstract model (Choppy et al., 2008). It gives a better and structured view of the system to be designed by identifying the components within the system. The validation process also becomes easier: the system properties are checked at each step. Thus, abstract models are validated before new details are added. Refinement helps in coping with system complexity since it may preserve some properties or analysis results obtained at an earlier step for a more abstract model.

In a Petri net, one place may be refined into a subnet with many places and transitions (Choppy et al., 2008). By refining one place, a set of places can be created, resulting in an expanded Petri net. Fig. 7.5 depicts the refinement of place PA into places PA1, PA2, and PA3. This refinement is useful for detailing initial design, from a high-level Petri net to a complete model (Vogler, 1992, Chapter 4).

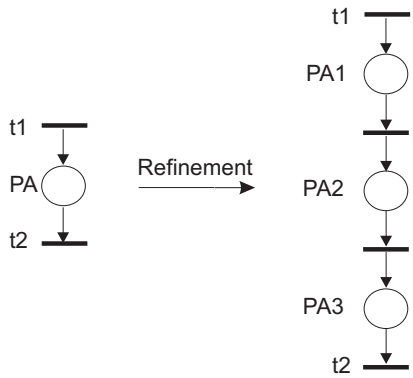

Figure 7.5: Example of top-down design with Petri nets

7.3.2 Bottom-up

A bottom-up modeling strategy usually starts by designing simple models representing parts of the system, and later combining them into more complex ones until the whole model is built. For instance, places of two diverse Petri nets can be merged (place composition), which creates a single Petri net (Jeng and DiCesare, 1995; Huang and Kirchner, 2009).

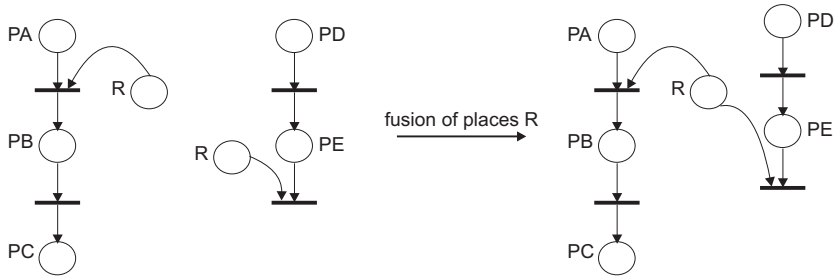

Figure 7.6: Example of bottom-up design with Petri nets

A classical example is the fusion of common places representing the same resource in two different Petri nets (Jeng and DiCesare, 1995; Cheung et al., 2006), which is a simple and effective way to model communication between subnets (Girault and Valk, 2002). In Fig. 7.6, initially there are two places R, each one representing the same resource that is used by two different processes modeled by two Petri nets. By merging the places R, the two Petri nets are merged into one, in which R is a shared resource.

7.4 Modeling Urban Traffic Signals with Petri nets

The approach in this chapter is to use Petri net Components with time interval associated to places for both top-down (refinement of places) and bottom-up (combining nets) modeling and perform verification using theorem proving and invariant analysis. The approach is applied to three studies:

- In subsection 7.4.1, to model one traffic signal controlling a single junction.
- In subsection 7.4.2, to model a responsive traffic signal that adapts the amount of green time.
- In subsection 7.4.3, to model a subnetwork with two junctions, each one controlled by one traffic signal, provided with an offset mechanism.

After the modeling, these models are verified in Section 7.5, using invariant analysis, and in Section 7.6, by applying a theorem proving approach to verify specific scenarios.

7.4.1 Modeling a Traffic Signal for One Junction

Fig. 7.7 shows a simple junction, with vehicles coming from three road sections: A, B or C. The junction S is controlled by traffic signals. Some important requirements (non-exhaustive) for the proper functioning of traffic signals are (Gallego et al., 1996):

- The traffic signal must not allow the green state to two conflicting road sections simultaneously.
- Each traffic signal must follow a defined color sequence, normally green, yellow, red, and then back to green.
- The right to use an intersection has to be given to all sections.

The requirements above are used in all of the following examples of this chapter.

The situation related to the road junction of Fig. 7.7 is modeled as the Petri net of Fig. 7.8. It is assumed that the three road sections have traffic flow almost equally distributed, i.e., priority should not be given to a specific road section. The junction can be seen as a resource that is shared among the three road sections, and is represented in the Petri net model as the S place. The states of the traffic signal are represented as places and the transitions separate

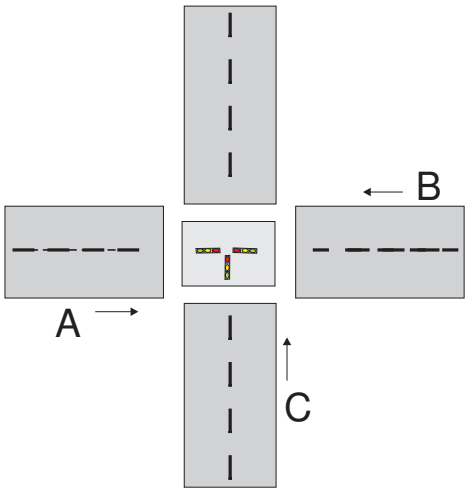

Figure 7.7: Junction with three road sections

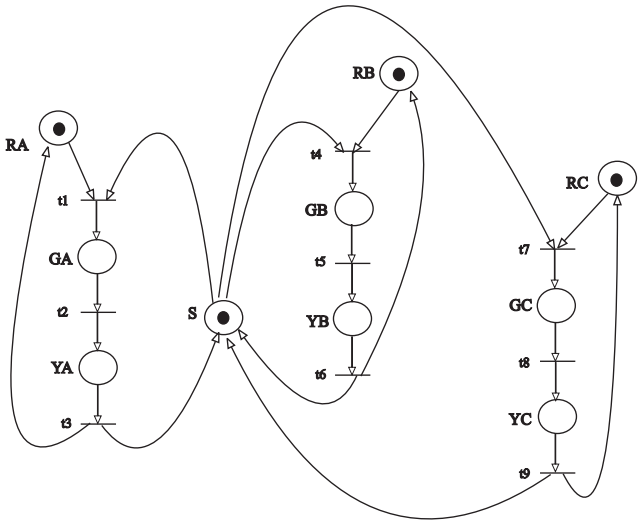

Figure 7.8: Petri net model for the traffic signal

one state from another. The firing of a transition in the model allows state changing.

A feasible solution for the conflict for the traffic signal of junction S is the firing of transition $t1$ first, leading to the green state GA, then the firing of transition $t2$ leading the yellow state YA, and finally the firing of transition $t3$, which releases the shared resource to road section B and allows again the red state to road section A (RA). Fig. 7.9 shows this situation. The place S corresponding to the shared intersection in the Petri net model is decomposed into places $S1$,

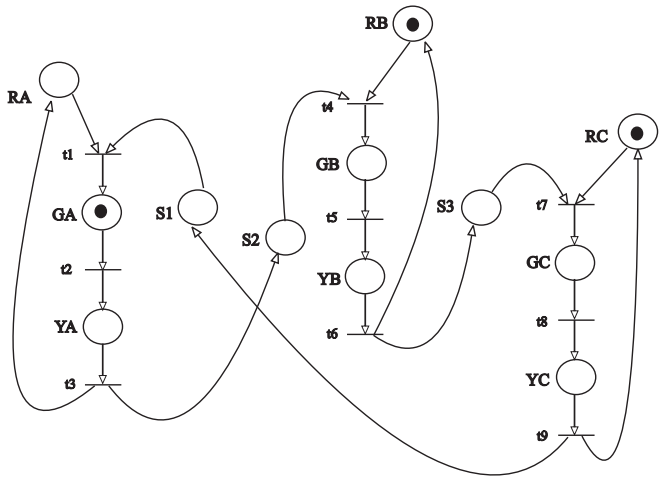

Figure 7.9: Petri net with the order of transition firing

$S2$ and $S3$, which means that green phases are allowed after red ones, passing to the yellow phases and returning to the red phase for sections A, B and C in sequence.

7.4.2 Responsive Traffic Signals

In order to be effective, advanced traffic signal control systems require an accurate current picture of the traffic flow and the status of the roadway network. Actuated control uses information about current demand obtained from sensors within the intersection to alter one or more aspects of the signal timing (Roess et al., 2003). An example of a sensor used in road traffic systems is an inductive loop. Actuated controllers may be programmed to accommodate variable green times or phase sequences. The approach of this example is to create different green periods. The green period is terminated in one of two ways: the maximum green time has been achieved, or it is not possible to extend the green due to high demand on the other road sections.

Fig. 7.10 shows a simple junction with two road sections. Road section B has a great flow of vehicles and road section A is a non-priority road. Considering the figure, there are sensors in the arterial road A to detect and count vehicles waiting for a green phase. The idea is to control the number of vehicles that are waiting for the green phase in such a way that big queues are not formed. The scenario proposed in this example is the one in which it is necessary to give priority to a main road (B in this case), which has greater traffic flow than the arterial road A. This priority is given by extending the period of the green phase as much as possible. Nevertheless, users in these arterial roads

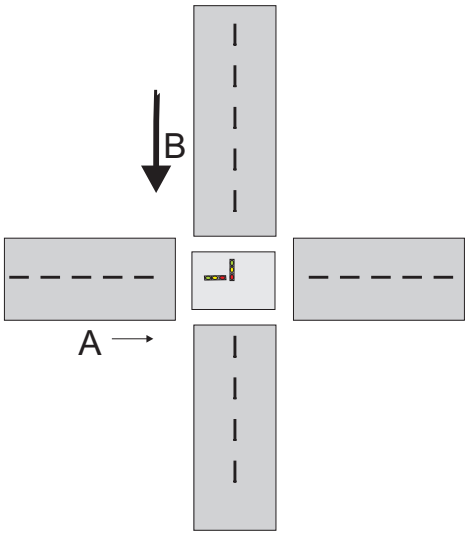

Figure 7.10: Junction with two road sections

should not think that the traffic signal is not working, which may increase the possibility of accidents. Therefore, every time there is a queue with a predetermined maximum number of vehicles, the controller will not allow any green time extensions anymore. This is important to make crossing possible for non-priority roads.

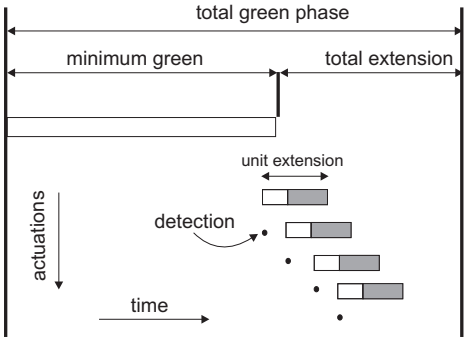

Figure 7.11: Actuated controller green phase intervals

An actuated phase normally has three timing parameters in addition to the yellow and red intervals. These are the minimum green, the unit extension, and the total green phase (maximum interval). Relationships among the intervals are shown in Fig. 7.11. Before the end of the unit extension period, in case a vehicle is detected, then the green phase is extended by one unit. The Traffic Detector Handbook (FHWA, 2006) recommends a unit extension of 3.5 seconds for roads where speeds are higher than 30 km/h. For the sake of simplicity, it is

assumed 4 seconds for each extension in this chapter, and a fixed interval (the same minimal and maximal duration) for each yellow state ($[5, 5]$ for instance).

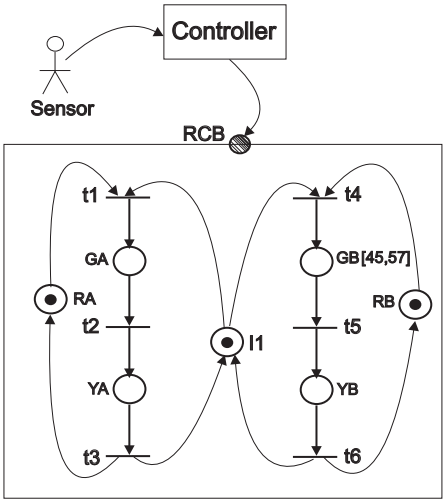

Figure 7.12: First level design for green time extension

The Petri net model of Fig. 7.12 is the first level of detail for the green time extension. It has an interval $[45, 57]$ associated to place GB, which corresponds to the minimum and maximum green time allowed.

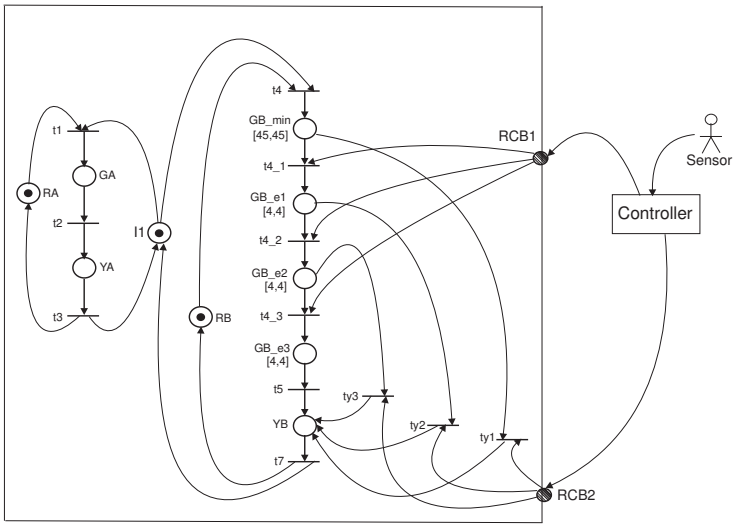

Figure 7.13: Second level design for green time extension

Sensors detect vehicles and send data to the controller, which will send the decision about the green time extension through the communication place RCB. The internal behavior of sensors and the controller are specified by Petri net

Components. In this example both are abstracted as black boxes receiving data and sending appropriate signals.

Before the end of the minimum green, considering a small amount of time that can be properly defined, the control will evaluate the presence of vehicles on the arterial road. If the queue is smaller than a pre-determined maximum number of vehicles, then the green time can be extended. Otherwise, the green phase ends and the state changes to yellow to allow safety time for vehicle crossing. Green time can be extended by a number of unit extension periods until there is a number of vehicles waiting in the non-priority section, or until the maximum green time is achieved.

The model of Fig. 7.12 is refined, through top-down modeling (see Section 7.3), into the model of Fig. 7.13 to allow a maximum of 3 extensions, each one of 4 seconds, for the green time of the main road. The communication places RCB1 and RCB2 (refined from place RCB), represented as shaded places in the model of Fig. 7.13, send real-time signals to the traffic signal. Before an extension is allowed, the sensor will send data to the controller, which will process and return one of two responses, as communication places: RCB1, allowing the extension, or RCB2, forcing the traffic signal to switch to the yellow phase. If the queue of vehicles reaches its maximum during the extended interval, a response is sent to the communication place RCB2, and the control switches immediately to the yellow state after the end of the unit extension period. The advantage of this approach is that instead of just identifying one vehicle and allowing the green for the non-priority road, the idea is to wait for a real demand of vehicles.

The 3 possible extensions are shown in Fig. 7.13 (places GB_e1, GB_e2 and GB_e3). Other scenarios, in which more than 3 extensions are allowed, can be modeled and used in a similar manner. This mechanism will allow a minimum of 45 seconds and a maximum of 57 seconds for the green time on the main road.

7.4.3 Modeling a Traffic Signal Control for a Subnetwork with Two Junctions

In order to achieve better results in terms of traffic flow, the network level must be evaluated, and problems such as traffic jams must be solved not only locally, but in a wide area (Soares and Vrancken, 2007a). Otherwise, the problems may just move to another place in the network. Fig. 7.14 shows a network of traffic signals controlling a network of road segments. The network is composed of main roads and arterial roads. Reasoning on a network level increases system complexity. The approach used in this chapter is to model and analyze complex systems by hierarchical decomposition into smaller components (divide and

conquer) (Rouse, 2003). Thus, the network is divided into subnetworks. Fig. 7.15 shows an example of a subnetwork of roads with two junctions: I1 and I2. The main road has a great flow from B to D, and A and C are non-priority roads.

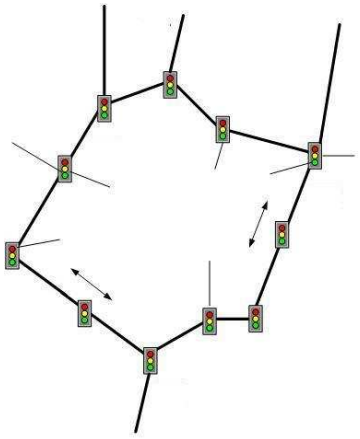

Figure 7.14: Network of traffic signals

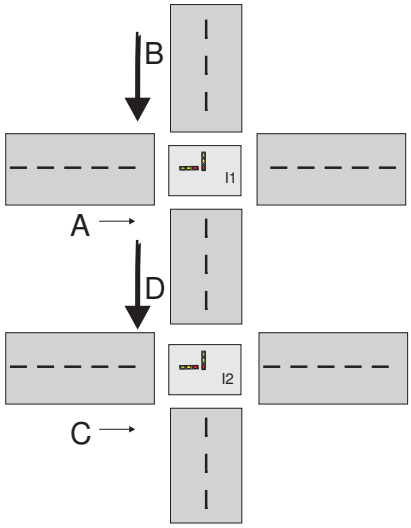

Figure 7.15: Subnetwork example

When a network of intersections is considered, it is possible to improve traffic flow by providing green-waves for main roads. The idea is to provide green time to the maximum number of vehicles in a sequence of junctions, such that they can cross without stopping. This is possible using adjusted offsets (Roess et al., 2003) (difference between green starting times in a network, considering intersections in sequence).

When reasoning on networks of road junctions, three additional requirements for traffic signals modeling are important (Soares and Vrancken, 2007e):

- Any user in the intersection should not wait for more than a maximum service delay, otherwise the user may presume that the traffic signal is not functioning, which can lead to non-secure decisions by the users (red light violations), or the formation of big queues.
- The offset is ideally designed in such a way that as the first vehicle just arrives at the next intersection in the network, the signal controlling this intersection turns green.
- The length of green time for each road section shall be different, depending on section priority, for instance. This length shall be extended to improve traffic flow.

The Use Case of Fig. 7.16 shows a context diagram for traffic signals controlling a network of junctions and considering the requirements just mentioned. The Use Case "Control Phases" depends on actors Vehicle and Sensor. Data detected by sensors (presence and number of vehicles) are sent to the controller. The actor Controller is responsible to selecting a feasible signaling schema to be sent to actor Traffic Signal. In order to correctly control the network, safety rules must be applied (Use Case *include* relationship). Whenever possible, which means that safety rules will not be violated, performance rules are considered (Use Case *extend* relationship).

Each Use Case is a set of actions performed by the system, which yields an observable result for the actors involved. The dynamic behavior of the Use Cases is specified with Petri nets in the approach presented in this chapter.

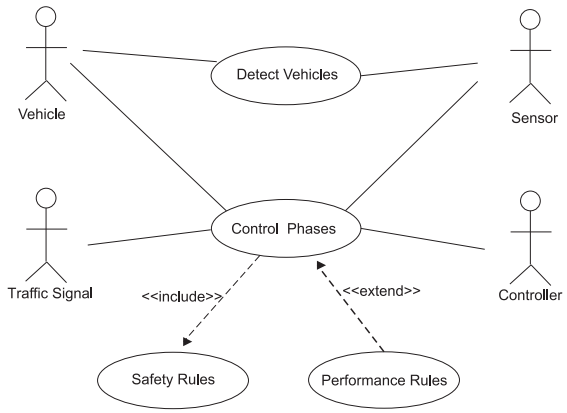

Figure 7.16: Context diagram for traffic signals controlling a network of junctions

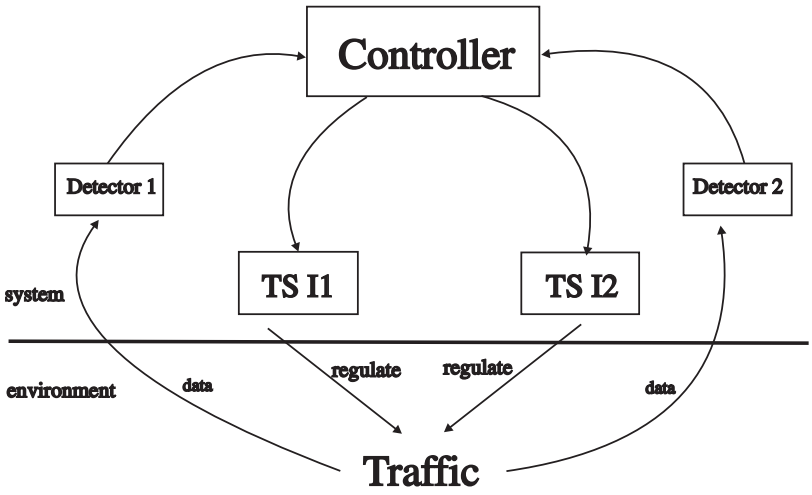

Figure 7.17: System architecture

Fig. 7.17 shows the elements of the system architecture and its relationship with the environment. All elements (controller, traffic signals, detectors) are specified using Petri net Components. Detectors send information about the traffic state to the controller. The Controller sends signals to the traffic signals respecting the requirements previously established. And the traffic signals regulate traffic.

Fig. 7.18 shows the controller and the traffic signal component. They communicate with each other through the component interface. The Controller sends control signals to the traffic signal, indicated by places such as "A_startGreen", which request the traffic signal to start a green phase for the corresponding road section. After the operation is finished by the traffic signal, a response is sent back by the traffic signal to the Controller ("A_RespG", for instance). After that, the controller can proceed with its processing.

Fig. 7.19 shows the controller model for controlling the traffic signal of intersection I1. In the initial state, both transitions $tsc1$ and $tsc4$ are enabled. The firing of these transitions allow respectively the green phase for road section A and then for road section B.

Supposing that $tsc1$ fires first, it will deposit a token in place "Go_GA", indicating that the controller commands the traffic signal of road section A to switch to green. The firing of $tsc1$ also deposits a token in the interface place "A_startGreen", which indicates to the traffic signal of road section A to start the green phase.

A composite Petri net is made by place fusion of smaller nets (see bottom-up

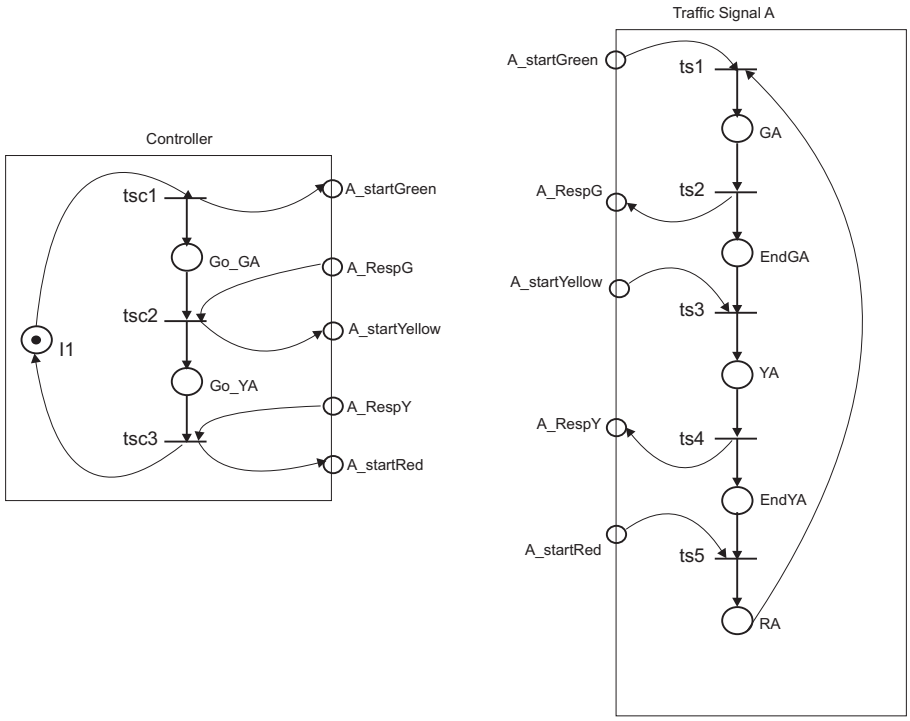

Figure 7.18: Controller and traffic signal components

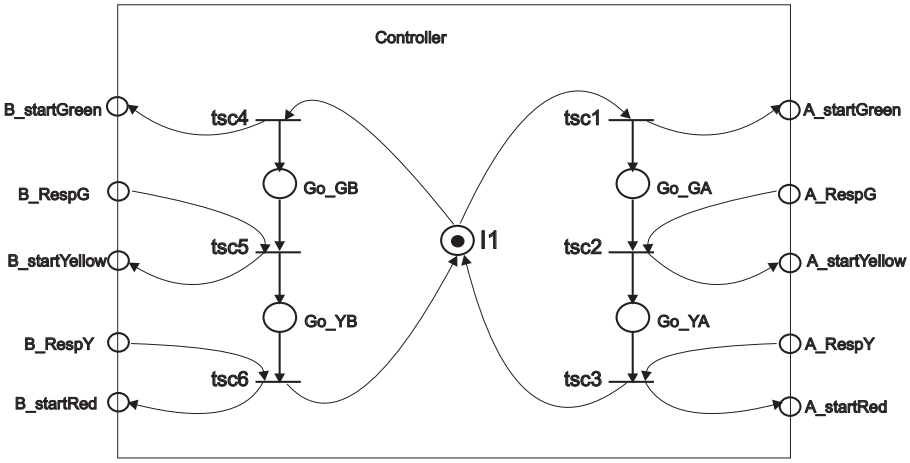

Figure 7.19: Controller model for road section I1

modeling in section 7.3). Places with the same label, such as "A_startGreen" in both controller and traffic signal components are fused in order to make one

single component.

Figure 7.20 represents the global Petri net specification concerning intersection I2, after place fusion of the controller and the traffic signals. A similar design is used for the controller and the traffic signal concerning intersection I1. The detector component is represented as a black box (its internal behavior is abstracted as it is not relevant to the design). Its interface is composed of place VD (vehicle detected), indicating that a vehicle was detected and this information is sent to the controller, which may respond using the place RVD.

To each place of the traffic signals a minimum and a maximum value ($[(\delta_p)min, (\delta_p)max]$) is associated, which indicates the duration a token must remain before being used to fire the transition. In practice, this is the duration of each phase of the traffic signal.

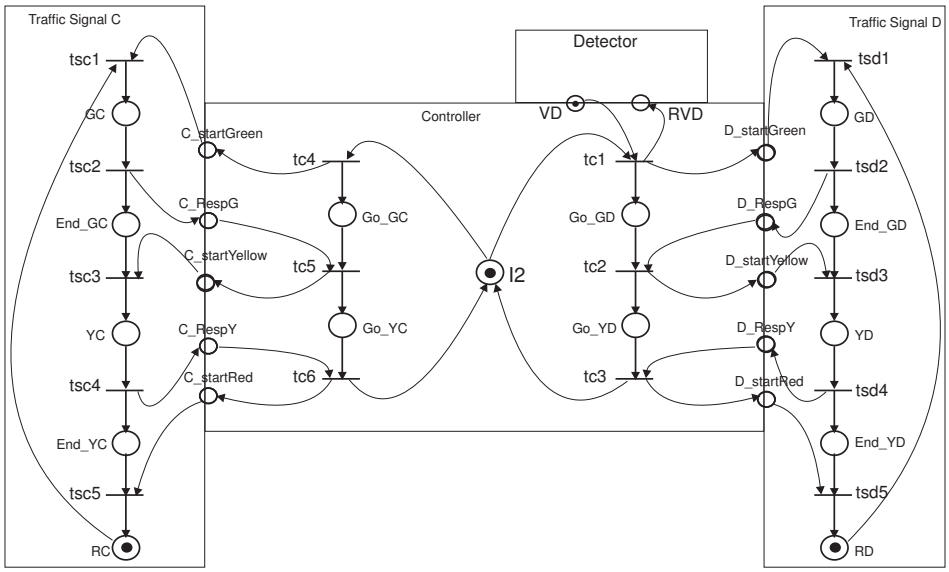

Figure 7.20: Combined Petri nets

Offset Mechanism

Coordinated traffic signals (Cheng et al., 2006) are very useful in road networks where traffic signals are located in close proximity. The progressing is given in such a way that the number of stops at intersections are minimized. However, coordination has some limitations. There are no guarantees that all vehicles in a platoon will be given green in the network at the required time. For long sections, the effect may not provide the expected result due to several factors, such as different vehicles speeds, traffic being already congested and difference in phases of the subsequent traffic signals. These factors may limit

the improvement in traffic flow.

Another problem that is often neglected is that the success in coordinating traffic signals in main roads may lead to traffic jams in arterial roads (where vehicles have to wait too long for green), which may move to the main road in the near future. This is avoided by limiting the green phase using p-time Petri net intervals.

The design proposed in this subsection makes possible the occurrence of green waves through the offset mechanism. Ideally, an offset is designed in such a way that when the first vehicle arrives at the next intersection in the network, the signal controlling this intersection turns green. This ideal offset is difficult to achieve with pre-determined offsets. The aim of the offset mechanism considered in this chapter is to facilitate that a platoon of vehicles coming from section B receives green time when reaching section D, improving traffic flow from intersection I1 to I2.

7.5 Invariant Analysis

From an initial marking, the marking of a Petri net can evolve by the firing of enabled transitions and, if there is no deadlock, the number of firings is unlimited. It may happen that all the reachable markings have some properties in common. A property which does not vary when the transitions are fired is said to be invariant. For instance, for certain subsets of places of a Petri net, the number of tokens in this subset remains constant.

Petri nets' structural properties are based on matrix equations. By examining the linear equations based on the execution rule of Petri nets and the matrices Pre and $Post$ it is possible to find subsets of places over which the sum of tokens remains unchanged.

For a Petri net PN with n transitions and m places, the Incident Matrix $C = [C_{ij}]$ is an $n \times m$ matrix of integers and its typical entry is given by $C = Post - Pre$. Considering $\bar{C}$ the transposed homogeneous incident matrix, an integer solution y of the equation $\bar{C}y = 0$ is called a P-invariant. The non-zero entries in a P-invariant represent weights associated with the corresponding places so that the weighted sum of tokens of these places is constant for all markings reachable from the initial marking.

7.5.1 Analysis of a Single Junction

P-invariant analysis is useful to prove mutual exclusion mechanisms. For example, considering the model of Fig. 7.8, there are three sets of p-invariants

that are similar to each other:

$$M(GA) + M(RA) + M(YA) = 1$$

$$M(GB) + M(RB) + M(YB) = 1$$

$$M(GC) + M(RC) + M(YC) = 1$$

The result of the first set is that there will be a token in place GA or RA or YA, which means that a token will be present in only one of these places at a determined period of time. This proves the mutual exclusion property and that the traffic signal switches from red phase, to green phase and yellow phase for the road section A. The same result is achieved for road sections B and C.

Another p-invariant is given by

$$M(GA)+M(GB)+M(GC)+M(S)+M(S2)+M(S3)+M(YA)+M(YB)+M(YC)=1$$

This invariant states that no two green or two yellow phases can start simultaneously for each of the conflicting sections, proving again the mutual exclusion property. As a matter of fact, the invariant analysis shows that no matter how the transition firing is performed, there is no possibility for the traffic signal to show unsafe states such as two greens for conflicting road sections.

7.5.2 Analysis of a Network of Junctions

The Petri net of Fig. 7.20 is p-invariant. There are 36 subsets of places over which the sum of tokens remain unchanged, as for instance, the set { End_GD, End_YD, GD, RD, YD }, which represents that there will be a token in End_GD or End_YD or GD or RD or YD. The sum of tokens in this set is always 1. As a matter of fact, it is not possible to achieve simultaneous green phase and yellow phase for road section D. There is an equivalent set for road section C.

Another interesting set is {Go_GC, Go_GD, Go_YC, Go_YD, I2}. The interpretation is that the controller will not allow simultaneous green phase or yellow phase for conflicting road sections, which fulfills an important safety requirement.

7.6 Scenario Analysis with Linear Logic

Linear Logic was proposed by Girard (Girard, 1987, 1995) as a refinement of traditional logic in order to deal with resources. In traditional logic, a proposition can be used as many times as one wants. For instance, if a fact A is used to conclude a fact B, the fact A is still available even after being used. Within Linear Logic, propositions are resources that can be produced and consumed. It is natural that resources can be counted, but traditional logic has no easy means to do that, due to two structural rules: weakening and contraction.

The weakening rule states that if $A \rightarrow B$ is valid, so is $A \wedge C \rightarrow B$. In a system that manipulates resources, this rule means that resources can appear and disappear at any time, and still the production is the same. In this case, the fact that a new resource is added has no effect in the system. The contraction rule states that if $A \wedge A \rightarrow B$, then $A \rightarrow B$. This means that even if a resource is eliminated, the final production is the same. These rules do not apply for systems that manipulate resources. In practice, resources can be counted and cannot simply appear or disappear without having an influence in a system. As Petri nets deal correctly with the notion of resource, some results appeared on combining Petri nets and Linear Logic (Brown, 1989; Engberg and Winskel, 1990).

7.6.1 Linear Logic and Petri nets

Linear Logic introduces three sets of connectives:

1. Multiplicatives: $\otimes$ ("times"), $\parr$ ("par"), and $\multimap$ (linear implication);
2. Additives: & ("with"), $\oplus$ ("plus");
3. Exponentials: ! ("of course") and ? ("why not").

The negation in Linear Logic, denoted $\perp$, does not express truth/falsehood properties, but rather concepts such as consumption and production of resources.

Only the multiplicative connectives $\otimes$ (times) and $\multimap$ (linear implication) are used in this chapter. As shown previously (Gehlot and Gunter, 1989), they are sufficient to represent, respectively, Petri nets markings and transition firings. The connective $\otimes$ represents accumulation of resources. Considering that A and B are resources, the presence of both is represented by the equation $A \otimes B$. The connective $\multimap$ represents causal dependency between resources. Considering

that A is a resource that is consumed to produce B, this is represented by the equation $A \multimap B$.

In this chapter, the translation from Petri nets to equations of Linear Logic is based on (Girault et al., 1997) to explicitly deal with markings. A marking M is represented by $M = P_1 \otimes P_2 \otimes \cdots \otimes P_k$ where P_i are place names with the presence of tokens. A transition is an expression of the form $M_1 \multimap M_2$ where M_1 and M_2 are markings representing respectively the consumed and produced resources. In fact, for the Petri net these are the equivalent Pre and $Post$ functions of the transition.

For example, considering the Petri net of Fig. 7.1, the initial marking is denoted $M_i = PA \otimes PA \otimes PB$ in Linear Logic, and transition t1 is represented as $t1 = PA \otimes PA \otimes PB \multimap PC$.

7.6.2 Linear Logic Proof Tree

In (Girard, 1987), all the rules for the sequent calculus proof are explained. In this chapter, the fragment Multiplicative Linear Logic is used. This fragment contains the multiplicative connective $\otimes$, representing accumulative resources, and the linear implication $\multimap$, representing causal dependency. There is no negation and the meta connective "," is commutative.

The deduction system used in Linear Logic is similar to the one used in classical logic, proposed by Gentzen in 1934 (Girard, 1995).

A linear sequent is denoted as $\Gamma \vdash \Delta$, where Γ and Δ are sets of formulas, i.e., $\Gamma = \Gamma_1, \Gamma_2, \ldots \Gamma_n$ and $\Delta = \Delta_1, \Delta_2, \ldots, \Delta_n$. The symbol Γ is the antecedent and the symbol Δ the consequent.

Linear Logic is commutative, which means that the order of the formulas in the consequent and the antecedent is indifferent. For instance, an antecedent can be represented as $\Gamma = F, G$ or as $\Gamma = G, F$; in a similar manner, a consequent can be represented as $\Delta = H, I$ or as $\Delta = I, H$.

A linear sequent proof consists of a set of hypothesis over an horizontal line and a conclusion above this line. Proving a sequent is to prove that it is syntactically correct.

$$\frac{hypothesis1 \quad hypothesis2}{conclusion}\ \text{rule}$$

The $\vdash$ (turnstyle) and the comma are Linear Logic primitive symbols. The $\vdash$ divides the sequent into the left part (premisses), and the right part (conclusions). The sequent is proved by applying the rules of the sequent calculus. To

prove a sequent is to show that it is syntactically correct. The proof tree is constructed bottom up. If the proof tree stops when all the leaves of the tree are identity sequents, such as $P \vdash P$, then in this case the sequent is proved. Otherwise, if there are no further Linear Logic rules that can be applied to transform the sequent into an identity one, then the sequent is not proved. The Linear Logic rules are:

Identity

$$\frac{}{F \vdash F}\text{id} \qquad \frac{\Gamma \vdash F, \Delta \qquad \Gamma', F \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\text{cut}$$

Structural

$$\frac{F, G \vdash \Delta}{G, F \vdash \Delta} X_L \qquad \frac{\Gamma \vdash F, G}{\Gamma \vdash G, F} X_R$$

Linear negation

$$\frac{\Gamma \vdash F, \Delta}{\Gamma, F \vdash \Delta}\text{L} \qquad \frac{\Gamma, F \vdash \Delta}{\Gamma \vdash F, \Delta}\text{R}$$

Multiplicatives

$$\frac{\Gamma, F, G \vdash \Delta}{\Gamma, F \otimes G \vdash \Delta}\otimes_{\text{L}} \qquad \frac{\Gamma \vdash F, \Delta \qquad \Gamma' \vdash G, \Delta'}{\Gamma, \Gamma' \vdash F \otimes G, \Delta, \Delta'}\otimes_{\text{R}}$$

$$\frac{\Gamma, F \vdash \Delta}{\Gamma, \Gamma', F \parr G \vdash \Delta, \Delta'}\parr_{\text{L}} \qquad \frac{\Gamma \vdash F, G, \Delta}{\Gamma \vdash F \parr G, \Delta}\parr_{\text{R}}$$

$$\frac{\Gamma \vdash F, \Delta \qquad \Delta', G \vdash \Delta'}{\Gamma, \Gamma', F \multimap G \vdash \Delta, \Delta'}\multimap_{\text{L}} \qquad \frac{\Gamma, F \vdash G, \Delta}{\Gamma \vdash F \multimap G, \Delta}\multimap_{\text{R}}$$

Additives

$$\frac{\Gamma, F \vdash \Delta}{\Gamma, F \& G \vdash \Delta}\&_{\text{L 1}} \qquad \frac{\Gamma, G \vdash \Delta}{\Delta, F \& G \vdash \Delta}\&_{\text{L 1}}$$

$$\frac{\Gamma \vdash F, \Delta \qquad \Gamma \vdash G, \Delta}{\Gamma \vdash F \& G, \Delta} \&_{\mathrm{R}} \qquad \frac{\Gamma, F \vdash \Delta \qquad \Gamma, G \vdash \Delta}{\Gamma, F \oplus G \vdash \Delta} \oplus_{\mathrm{L}}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash F \oplus G} \oplus_{\mathrm{R}\ 1} \qquad \frac{\Gamma \vdash G, \Delta}{\Gamma \vdash F \oplus G, \Delta} \oplus_{\mathrm{R}\ 2}$$

In this chapter, the logical rules id, $\otimes_L$, $\otimes_R$ and $\multimap_L$ are applied to build the proof tree as follows (Girault et al., 1997):

- the rule $\otimes_L$ transforms a marking such as $P1 \otimes P2$ into a list of atoms (P1, P2 for instance). This rule is applied repetitively to separate a marking into atoms, which allows the application of the $\multimap_L$ rule.
- the $\multimap_L$ rule expresses a transition firing. It generates two sequents. The left sequent represents the tokens that were consumed by the transition firing. The right sequent represents the new tokens produced by the transition firing. In case the new sequent is an identity one (e.g., $P1 \vdash P1$), a leaf was reached and this branch of the tree was proved. Otherwise, the rule $\otimes_R$ is applied.
- the rule $\otimes_R$ transforms sequents of the form $P1, P2 \vdash P1 \otimes P2$ into two identity ones $P1 \vdash P1$ and $P2 \vdash P2$, which are proved leaves.

A scenario is a set of events that transform an initial marking into a final one. A linear sequent $M_i, t_s \vdash M_f$ represents a scenario where M_i and M_f are respectively the initial and final markings, and t_s is a set of non-ordered transitions.

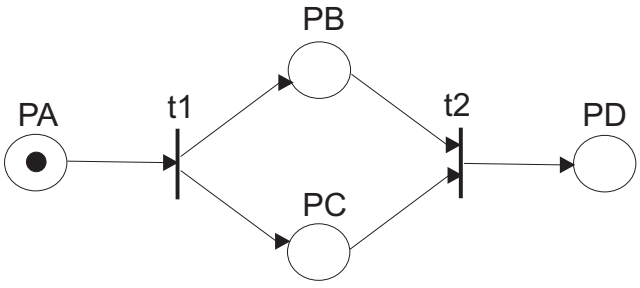

Figure 7.21: Petri net example for construction of the Linear Logic proof tree

An example of a Petri net to be used for the construction of the Linear Logic proof tree is shown in Figure 7.21. The sequent to be proved is: $PA, t1, t2 \vdash$

PD. This means that from place PA, after firing transitions t1 and t2, the final marking PD is reached. The linear formula for transitions t1 and t2 are respectively: $PA \multimap PB \otimes PC$ and $PB \otimes PC \multimap PD$. The proof tree is given as follows:

$$\dfrac{\dfrac{PA \vdash PA \qquad \dfrac{\dfrac{\dfrac{PB \vdash PB \qquad PC \vdash PC}{PB, PC \vdash PB \otimes PC} \otimes_R \qquad PD \vdash PD}{PB, PC, (PB \otimes PC \multimap PD) \vdash PD} \multimap_L}{PB \otimes PC, (PB \otimes PC \multimap PD) \vdash PD} \multimap_L}{PA, (PA \multimap PB \otimes PC), t2 \vdash PD}}{PA, t1, t2 \vdash PD}$$

From the proof tree generated, it is clear that the sequent is proved, as all leaves are identity sequents.

7.6.3 Analysis of Scenarios

In dynamic systems, an important issue is whether a specific state is reachable. Reasoning about reachable states allows one to verify, for instance, whether an unsafe state may occur.

The coverability graph is a method for analyzing Petri nets which consists of the enumeration of all reachable markings. When a model is bounded, the coverability graph is called reachability graph, since it contains all possible reachable markings. The algorithm to construct the coverability graph is shown in (Murata, 1989). Basically, the algorithm obtains, from the initial marking, a graph of the reachable markings of a Petri net when firing the enabled transitions.

Reachability analysis was applied to the Petri net of figure 7.20, but without considering the detector interface, which would characterize a specific scenario. The analysis resulted in 32 states, from state $S0 = \{$I2 RC RD$\}$ to state $S31 = \{$C_startRed End_YC I2 RD$\}$. The result is that the Petri net is live and bounded.

The reachability graph allows the verification of good properties, as reversibility, and verification of deadlock absence. Also, it can demonstrate that unsafe states are never reached. Nevertheless, the reachability analysis presents some disadvantages. The most important is that its construction is a very hard problem from a computational point of view. This is because the size of the state-space may grow more than exponentially with respect to the size of the

Petri net model. Another problem is that it is not possible to extract the causality links between the different firings, which is fundamental in scenario deriving (Demmou et al., 2004). Finally, the reachability graph presents some difficulties in analyzing properties in which concurrency plays a fundamental role (Girault and Valk, 2002).

The equivalence between Petri nets reachability and the proof of sequents in Linear Logic was proved in (Girault et al., 1997). When a sequent is proved by applying the rules of the sequent calculus of Linear Logic, equivalently it can be stated that the final marking is reached from the initial one by firing a set of transitions. As a direct consequence, if a sequent is not provable, then the marking is not reachable. The result is that Linear Logic can be seen as an analysis tool for Petri nets and can be applied to give some important results about the models.

Within Linear Logic, the reachability problem is in fact the problem of sequent proving. It gives a formal and logical framework that assures the coherence of causality between transitions firing. Also, it is possible to extract from the proof tree some information about the order of transition firing and the evaluation of the scenario (with or without time associated). The other advantage is that it is possible to derive specific scenarios that one wants to study directly from the Petri net without constructing the reachability graph (Demmou et al., 2004). Finally, when changing the initial marking or the final marking, it is not necessary to create the reachability graph again, provided that these states belong to the set of states present in the Linear Logic proof tree. The main purpose in this chapter is not to check all possibilities, which can be done using the reachability graph, but to study specific scenarios using theorem proving.

The considered scenario (Fig. 7.22) is the one in which a green phase is given first to road section C (token in I2C) and then to road section D (token in I2D) after a vehicle is detected. These places were created by the refinement of place I2 of Fig. 7.20. The same approach can be easily applied to other scenarios.

Scenario Analysis - Possible reachability of an unsafe state

Considering the proposed scenario, one needs to verify if there is a possibility of reaching an unsafe state, such as the reachability of simultaneous green phases for two conflicting road sections, as for instance, C and D, which is characterized by a token present simultaneously in GC and GD. From the Linear Logic point of view, this is equivalent to prove that the state GC $\otimes$ GD is not reachable from another state. What is to be proved is that from one chosen initial state the final state GC $\otimes$ GD is not reached. The equivalent sequent is

RC $\otimes$ RD $\otimes$ I2C $\otimes$ VD, tc4, tsc1, tc1, tsd1 $\multimap$ Go_GC $\otimes$ GC $\otimes$ Go_GD $\otimes$ GD.

This sequent states that from red states for both sections, it will be allowed

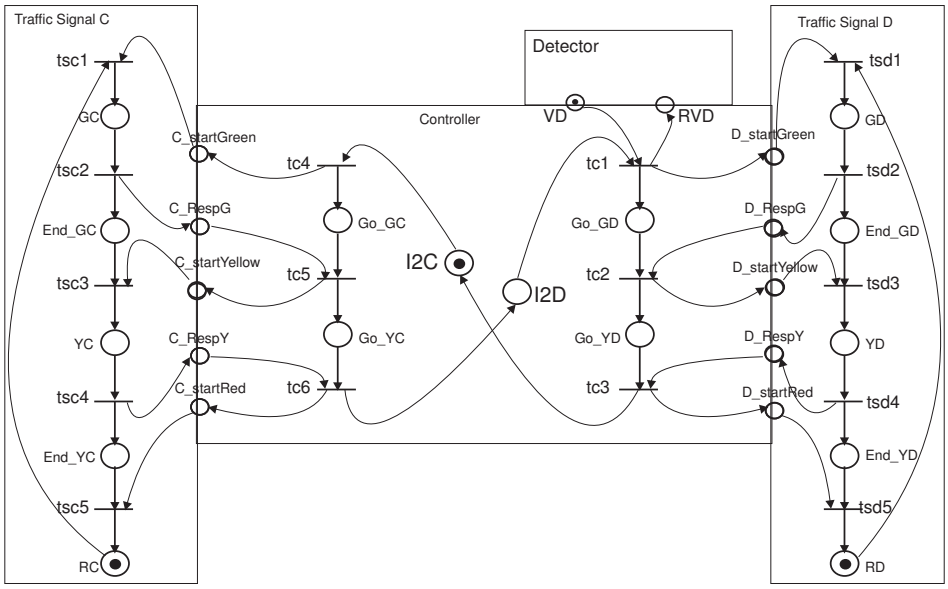

Figure 7.22: Petri net for scenario analysis

Table 7.1: Linear Logic representation of transitions

$tc4 = I2C \multimap C\_startGreen \otimes Go\_GC$
$tsc1 = C\_startGreen \otimes RC \multimap GC$
$tc1 = I2D \otimes VD \multimap D\_startGreen \otimes RVD \otimes Go\_GD$
$tsd1 = D\_startGreen \otimes RD \multimap GD$

green for sections C and D simultaneously.

Table 7.1 shows the Linear Logic representation of the transitions. The proof tree is given as follows (places C_startGreen and D_startGreen were respectively renamed to CsG and DsG due to lack of space in the proof tree).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{VD,RD,Go\_GC(I2D\otimes VD\multimap Go\_GD\otimes DsG\otimes RVD),tsd_1\vdash Go\_GC\otimes GC\otimes Go\_GD\otimes GD}\multimap_L}}{RC\vdash RC\ \ CsG\vdash CsG\ \ VD,RD,Go\_GC(I2D\otimes VD\multimap Go\_GD\otimes DsG\otimes RVD),tsd_1\vdash Go\_GC\otimes GC\otimes Go\_GD\otimes GD}\multimap_L}}{I2C\vdash I2C\qquad VD,RC,RD,(CsG\otimes RC\multimap GC),tc_1,tsd_1\vdash Go\_GC\otimes GC\otimes Go\_GD\otimes GD}\multimap_L}}{VD,RC,RD,I2C,(I2C\multimap Go\_GC\otimes CsG),tsc_1,tc_1,tsd_1\vdash Go\_GC\otimes GC\otimes Go\_GD\otimes GD}\multimap_L}}{VD,RC,RD,I2C,tc_4,tsc_1,tc_1,tsd_1\vdash Go\_GC\otimes GC\otimes Go\_GD\otimes GD}\otimes_L}}{VD\otimes RC,RD,I2C,tc_4,tsc_1,tc_1,tsd_1\vdash Go\_GC\otimes GC\otimes Go\_GD\otimes GD}\otimes_L}}{VD\otimes RC\otimes RD,I2C,tc_4,tsc_1,tc_1,tsd_1\vdash Go\_GC\otimes GC\otimes Go\_GD\otimes GD}\otimes_L}$$

$$VD\otimes RC\otimes RD\otimes I2C,tc_4,tsc_1,tc_1,tsd_1\vdash Go\_GC\otimes GC\otimes Go\_GD\otimes GD$$

From the proof tree, it is clear that the final state is not reached, as there are no further Linear Logic rules that can be applied to transform the branch $VD,RD,Go\_GC(I2D\otimes VD\multimap Go\_GD\otimes DsG\otimes RVD),tsd_1\vdash Go\_GC\otimes GC\otimes Go\_GD\otimes GD$ of the proof tree into identity sequents. The same reasoning can be done if the green phase is allowed first to road section D. In this case, a similar tree is generated with the same final result.

Other results can be inferred from the proof tree. The unsafe state is not reached because it is not possible to fire transition tc1, which is not enabled. In the proof tree, this is represented by not having the token I2D, as the VD is present. Supposing that this unsafe state is reached due to some error, then it is clear that transition tc1 was fired because it was enabled by a token in I2D, and there is a design problem. In addition, it is clear that the green phase is only reached in this scenario for road section D when there is a token in place VD, which means that a vehicle was detected. Supposing that it is not possible to fire the transition to allow the green phase, one possible reason is that the detector failed or is damaged.

Scenario Analysis - Reversibility

It is fundamental to prove that the controller and the traffic signals are capable of switching from red to green and yellow and returning to red, without possibilities of giving both red and green phases to the same road section. Considering $M_i = M_f$ = RC $\otimes$ RD $\otimes$ I2C as both the initial and final marking and t_s = {tc4, tsc1, tsc2, tc5, tsc3, tsc4, tc6, tsc5, tc1, tsd1, tsd2, tc2, tsd3, tsd4, tc3, tsd5} the set of transitions to be fired in the scenario, a Linear Logic sequent representing the reversible scenario is given as follows:

$$RC\otimes RD\otimes I2C, t_s \vdash RC\otimes RD\otimes I2C$$

An important remark is that the t_s set of transitions is not ordered. For instance, tsc5 and tc1 can be fired simultaneously, or tsc5 first or tc1 first. This

is an advantage of using Linear Logic sequent calculus instead of a reachability graph, in which all the possibilities are evaluated. The same remark holds for transitions tsd5 and tc4.

Only part of the proof tree is given as follows (place C_startGreen was renamed to CsG due to lack of space in the proof tree, and only the current enabled transition appears on the proof tree).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{RC \vdash RC \quad RD \vdash RD \quad I2C \vdash I2C}{RC, RD, I2C \vdash RC \otimes RD \otimes I2C}\otimes_R}{\vdots \quad \vdots \quad \vdots}\otimes_L}{\vdots \quad \vdots \quad \vdots}}{RC, RD, I2C, (I2C \multimap Go\_GC \otimes CsG) \vdash RC \otimes RD \otimes I2C}\multimap_L}{RC, RD, I2C, t_s \vdash RC \otimes RD \otimes I2C}\otimes_L}{RC \otimes RD, I2C, t_s \vdash RC \otimes RD \otimes I2C}\otimes_L}{RC \otimes RD \otimes I2C, t_s \vdash RC \otimes RD \otimes I2C}$$

All leaves of the tree are identity ones. This means that the final state is reached from the initial one after firing the set of transitions. As a matter of fact, the Petri net is reversible. The practical implication is that the designed net is capable of going back to its initial state after each cycle and then repeat the cycle, which is according to the requirements and exactly the expected behavior of a traffic signal.

7.7 Conclusion

Urban traffic signals control systems are critical software-intensive systems, in the sense that several elements are involved, and they are applied to a critical infrastructure. The design must address performance and safety, as human life is directly involved and transportation infrastructure is fundamental to the economy of a country.

In this chapter, the application of Petri nets and extensions for the design and analysis of urban traffic control systems is proposed. A divide and conquer approach is followed in order to address system complexity. Thus, from a network of roads, a subnetwork of two road sections are considered each time, but due to modularity aspects used in this chapter, this is scalable to greater networks. The modularity is based on components whose internal behavior is

specified by Petri nets. Another Petri net extension used is the association of a time interval to each place, allowing traffic signals phase durations to be specified from a minimum to a maximum value.

The verification of Petri net models is initially performed based on structural analysis. This is done by applying the invariant theory to a Petri net model. Reachability analysis is applied to the generated net in order to give results about some good properties. Finally, scenario analysis is based on theorem proving using Linear Logic proof trees. The advantage of using Linear Logic for scenario evaluation is that it is possible to derive specific scenarios that one wants to study, without constructing complete reachability graphs.

An advantage of Petri nets is that the same model can be used as input for diverse Systems Engineering activities. As a graphical model, Petri nets can be simulated through the common token player algorithm in a computer based tool. The same model can be formally verified using one of the many methods provided by the Petri net theory. Once the model is shown reliable, it can be used to evaluate the systems' performance. Finally, when all these activities are successful according to defined requirements and criteria, the models can be used for automatic code generation. The activities of modeling and verification have been considered in this chapter. Performance evaluation and code generation are activities to be considered in future research.

Chapter 8

Evaluation

This chapter is about the evaluation of the suitability of including and combining semi-formal modeling languages (UML and SysML) and a formal modeling language (Petri nets) in the 4+1 View Model of Architecture, as well as the applicability of this approach to design software-intensive systems. In order to test the hypotheses of Section 8.1, our choice was to use the "industry as laboratory" approach (Section 8.2). Thus, the first step was to lecture on Model-Driven Software Engineering (Section 8.3) to the employees of Trinité that would be the subjects for the evaluation. The evaluation itself was a process in which a survey was proposed followed by individual interviews (Section 8.4). With these data, an analysis of findings (Section 8.5) was performed.

8.1 Hypotheses

The general hypothesis to be tested is given as follows:

"The use of multiple views, modeling languages and well-defined domain and software architecture has a positive impact on the design of Software-Intensive Systems."

The general hypothesis can be refined as the following four derived ones.

1. SysML adds value in the design of Software-Intensive Systems compared to using only UML.

   SysML is a modeling language created from UML, considered both a subset of and an extension to UML (OMG, 2008a). We want to evaluate the advantages of using SysML, i.e., what modeling capabilities SysML offers that are not offered by UML.

2. Including SysML in the 4+1 View Model of Architecture adds value in the design of Software-Intensive Systems.

   We want to evaluate what added value SysML can bring when designing models for each view of the 4+1 View Model of Architecture. Thus, we would like to know which SysML diagrams can be used for each view, and what advantages this will bring in the design of software-intensive systems.

3. Including Petri nets in the 4+1 View Model of Architecture adds value in the design of the dynamic behavior of Software-Intensive Systems.

   The 4+1 View Model of Architecture is mostly associated with object-oriented modeling languages/methods for designing models for each view, such as the Booch method and the UML language. We would like to include a formal method for designing dynamic models for the architecture views and evaluate the added advantages, for instance, in terms of expressivity for modeling software-intensive systems. We also would like to know what shortcomings the use of formal methods can bring, and how to overcome identified issues.

4. Using the 4+1 View Model of Architecture facilitates the maintainability of Software-Intensive Systems.

   Dealing with legacy systems is a certainty in software-intensive systems development. Thus, the software architecture should provide means to facilitate the maintenance of software-intensive systems. We would like to know to what extent the 4+1 View Model of Architecture facilitates the maintainability of Software-Intensive Systems.

8.2 Test Design

The research presented in this thesis is about Software Engineering, which is a discipline that is most necessary to develop large, complex software. This is most likely to happen in practice, with real world problems. Thus, the choice was to test the hypotheses in an industrial environment.

The industry as laboratory approach uses the actual industrial setting as test environment (Potts, 1993). Within this approach, the research team builds an intimate relationship with an industrial product creation team, which is mutually beneficial (Muller and van de Laar, 2009). The research team gets inspiration from real industrial challenges, and at the same time it gets a means to verify research results in industrial settings. The industrial partner gets inspiration from results, and is continuously challenged by unbiased and critical people.

The testing was performed at Trinité, a company specializing in developing RTMS. A number of reasons support this choice and makes clear why Trinité is suitable for the testing.

Trinité is the company that developed HARS, and where part of the PhD research was performed. Besides, Trinité is a company that deals with complex problems, and is always eager to learn better and improved ways to solve these problems. As a company, Trinité is enthusiastic in improving the software development methods in use. Finally, Trinité employees' and managers' find the relationship with academia very positive.

In practice, the introduction of a new technology in industry is often done by a course. A common format for a course is that an expert from outside the company lecture to the employees. This saves time, as the best practices are already introduced through an experienced person.

8.3 Course on Model-Driven Software Engineering

The proposed course had a total duration of 12 hours, divided into 2 days (6 hours each day) and was followed by ten employees of Trinité, varying from developers to managers. The content of the course was divided into four parts as follows:

Part 1 - The 4+1 Views of Software Architecture Definition of software architecture. The five views. Examples with UML.

The purpose of Part 1 was to present the importance of having a defined software architecture to develop software-intensive systems. The 4+1 View Model of Architecture was presented, and for each view examples were given using UML diagrams and notations used by the company. It is important to note that Trinité already uses the 4+1 View Model of Architecture. Thus, the purpose with Part 1 was to identify possible problems with the architecture, and to discuss how the company uses the architecture in practice.

Part 2 - UML 2.0 - Dynamic diagrams Use Cases. Sequence diagrams. Activity diagrams. State-Machine diagrams. Relationships between diagrams.

The company has been using UML 1.x for many years, but had not yet switched to the UML 2 version, which was the version explained during the course. The emphasis of Part 2 was on UML dynamic diagrams. The main reasons for that are: (i) dynamic diagrams are more important than static diagrams for the type of systems being developed at Trinité, (ii)

knowing the relationship between dynamic diagrams is not trivial (Dobing and Parsons, 2006), and (iii) dynamic diagrams are often incorrectly used (Agarwal and Sinha, 2003; Anda et al., 2006). After the explanation about each of the most used dynamic diagrams, the relationship between them was explored. For instance, how Sequence diagrams and Use Cases are related to each other. In addition, best practices of when to use each diagram were presented, as due to the number of diagrams, knowing which one should be used in a specific situation is difficult (Anda et al., 2006).

Part 3 - SysML 1.0 Use Cases. Requirements diagram. Tables. Block diagrams.

Part 3 had two main objectives. The first one was to present the SysML diagrams relevant for the thesis (the ones actually used in this thesis and that needed to be evaluated). The emphasis was on the SysML diagrams used for Requirements Engineering, including the extensions to the basic SysML Requirements diagram (see Chapter 4) and the SysML Block diagram used for Software Architecture design (see Chapter 6). The second objective was to show where and how SysML could be included into the 4+1 View Model of Architecture, including discussion on the advantages of combining UML and SysML diagrams for describing software architecture views.

Part 4 - Petri nets Definition. Modeling capabilities. Tool support.

The purpose of Part 4 was to briefly present Petri nets. The emphasis was on the modeling capabilities and the advantages over UML dynamic diagrams (particularly the Sequence diagram). For instance, the possibility of modeling in a top-down or bottom-up way (see Chapter 7) was explained. In addition, the representation of time interval was presented. To finalize, tool support was shown, including the token game simulation and verification using reachability and invariant analysis.

During the course the participants had the opportunity to learn the theory combined with examples in a variety of domains. A number of exercises were proposed during the course. The purpose of these exercises was twofold: (i) employees could practice what they have just learnt, and (ii) as an evaluation of the difficulties they had with the modeling languages. The exercises were tailored as entry-level ones, with low level of complexity and considered feasible to be done individually. The list of exercises is given in Appendix C. In addition to the examples, a simple project was proposed by the company to each participant, in which they had to use the knowledge gained during the course to create a design for the project.

An important discussion during the course was about software development activities at Trinité that were performed in the past and were not performed

anymore, or that were not uniformly performed. For instance, the UML Activity diagram was used in the past at the company, and for no apparent reason was no longer being used. In addition, it was recognized that an effort on standardization was necessary such that all employees would use the same patterns for documents and design. An example is the use of the UML State Machine diagram, which was considered useful, but was not used by all employees.

There were many discussions on software architecture and its importance for software development and evolution. All employees agreed that having a well-defined architecture is fundamental. They discovered during the discussions about architecture that implementation layers were being mixed at the company. They realized the need for improved separation of concerns.

8.4 Evaluation

Unvalidated methods and techniques have been routinely used in practice throughout the entire spectrum of design, implementation and evaluation activities of software development (Davis, 1989; Venkatesh et al., 2003). Identifying interventions that could influence adoption and use of new information technologies (IT) can aid managerial decision making on successful IT implementation strategies (Jasperson et al., 2005).

Many models have been proposed to explain and predict the use of a system (Venkatesh and Bala, 2008). The most widely employed model of IT adoption and use is the Technology Acceptance Model (TAM) (Davis, 1989; Adams et al., 1992). The model suggests that when users are presented with a new technology, a number of factors influence their decision about how and when they will use it. It posits that individuals' behavioral intention to use an IT is determined by two beliefs: *perceived usefulness*, defined as the degree to which a person believes that using an IT will enhance his or her job performance, and *perceived ease of use*, defined as the degree to which a person believes that using a particular system will be free of effort. A third variable is considered in this research: *perceived usage* (Adams et al., 1992), defined as the degree to which the user will actually use the technology.

It should be emphasized that perceived usefulness and ease of use are people's subjective appraisal of performance and effort, respectively, and do not necessarily reflect objective reality (Davis, 1989). Though different individuals may attribute slightly different meaning to particular statements, the goal of the multi-item approach is to reduce any extraneous effects of individual items (Davis, 1989).

Another model created after the introduction of TAM is the Technology Transi-

tion Model (TTM) (Briggs et al., 1998). TTM was created by extending TAM. The purpose of TTM is to be a broader model of technology acceptance than TAM that may be useful for reducing the duration and risk of technology transition projects. However, TTM does not replace TAM. As a matter of fact, TTM can be considered in further steps, when researchers want to know what causes a group of technology users to become self-sustaining, i.e., the effectiveness of technology transference in a community.

In this research we are interested in the individuals acceptance process, which is a good starting point to view how a new technology might be accepted in a community. Therefore, extensions to TAM in which the social context is taken into account (Venkatesh and Davis, 2000), although useful, are not investigated here. In addition, the purpose is to evaluate initial acceptance of technology, which makes TAM a suitable model.

TAM was initially proposed to evaluate information systems tools (Davis, 1989; Adams et al., 1992), but it has also been used in other contexts, such as to evaluate the adoption of the BPMN notation to teach business process at the university level (Rozman et al., 2008), the intention to continue using a process modeling technique (Recker, 2007), and data modeling techniques (Moody, 2002). In our case, we want to evaluate software and systems engineering modeling languages (UML, SysML and Petri nets) in general, and more specifically their adaptation and their integration into a well-known architecture model, the 4+1 View Model of Architecture.

The evaluation was performed using two techniques: surveys, based on the TAM theory, and interviews.

8.4.1 The Questionnaire

All the participants that returned the questionnaire had followed the Model-Driven Software Engineering course and had had some practice after making a design for a simple project. Thus, the questionnaire was not proposed as soon as the course ended.

The participants answered a questionnaire composed of 43 statements in which they made their opinions explicit. A 5-point Likert scale (Likert, 1932) was proposed to measure perceived attitudes of the employees by providing a range of responses to each statement. The scale ranged from (1) strongly disagree, (2) disagree, (3) neutral, (4) agree, and (5) strongly agree. For the negative statements, it was inferred from low scores that the participants are positive about a statement in the questionnaire, while for the positive statements, a high score was inferred that the participants were positive about the statement.

The response categories in Likert scales have rank order, but the intervals between values cannot be presumed equal (Jamieson, 2004). As a result, Likert scales fall within the ordinal level of measurement. When treated as ordinal data, Likert responses can be analyzed using non-parametric tests, such as the Mann-Whitney test, the Wilcoxon signed rank test, and the Kruskal-Wallis test (Corder and Foreman, 2009). Non-parametric statistical methods were used to analyze the findings because the subjects involved in the study were few and not chosen randomly from a large population (they are all developers and managers in a company). The objective was to find the representative power of the results, and not necessarily the statistical significance (Yin, 2003). It is worthwhile to note that non-parametric techniques do not solve the problem of potential dependency between the answers of the participants. As a matter of fact, the results are interpreted carefully.

8.4.2 Questionnaire Results

The questionnaire was responded by ten employees. The answers regarding each question are shown in tables 8.1 to 8.11, in which "m" represents the mean, "s" represents the standard deviation, and "pos" indicates the number of positive answers, given in modulus because of negative statements. We arbitrarily considered as positive the answers "Agree" or "Strongly Agree" (values 4 or 5). For negative sentences we considered 1 and 2 for a positive response.

One table was created for each group of sentences (UML, SysML, Petri nets, Architecture). Perceived usage was also considered during interviews. The outcomes were evaluated in order to confirm or to refute the hypotheses. Conclusions were derived from the questionnaire and interviews. The purpose was to understand the perceived benefits of integrating UML, SysML and Petri nets into the 4+1 View Model of Architecture.

In statement 1, the fragments mentioned are "Optional", "Alternatives", "Loop", and "Parallelism". In statement 2, the mentioned relationships are "include" and "extend". In statement 3, the mentioned stereotypes are "boundary", "entity" and "control". In item 4, the Activity diagram is considered not only for process but also for detailed algorithm specification.

The answers relative to statements 1 to 5 indicate that constructions considered advanced and rarely applied in software design with UML, such as Use Case relationships and fragments of Sequence diagrams were considered useful. On the other hand, the response about class stereotypes was not conclusive. The Activity diagram was considered useful for representing the overall dynamics of a system. In addition, UML was considered useful for software design at the company by 9 out of 10 respondents.

Table 8.1: Perceived usefulness of UML - statements 1 to 5

Statement	1	2	3	4	5	m	s	pos
1 - The UML 2 Sequence Diagram fragments will make my models more expressive.			3	6	1	3.8	0.63	7
2 - The Use Case Diagram relationships will make my models more expressive.			4	6		3.6	0.51	6
3 - The Class Diagram stereotypes will facilitate separation of concerns.			6	4		3.4	0.51	4
4 - The Activity Diagram helps me to better understand the system.			1	7	2	4.1	0.56	9
5 - Overall using UML is useful in my job.			1	2	7	4.6	0.56	9

Table 8.2: Perceived ease of use of UML - statements 6 to 11

Statement	1	2	3	4	5	m	s	pos
6 - I think UML diagrams are difficult to understand.	1	7	1	1		2.2	0.78	8
7 - I know when to use each UML diagram.		5	1	3	1	3	1.15	4
8 - I know how to relate Use Cases and Sequence Diagrams.		1	1	8		3.7	0.67	8
9 - I know how to relate Use Cases and Class Diagrams.		1	1	8		3.7	0.67	8
10 - I know how to relate Class Diagrams and Sequence Diagrams.	1	1		7	1	3.6	1.17	8
11 - It is easy for me to understand a UML design made by others.		2	3	5		3.3	0.82	5

With the answers to statements 6 to 11, it is clear that UML is not considered, by most respondents, a difficult modeling language. Even the relationship between diagrams, which is normally not well-documented in articles and books, was considered easy for most of the respondents. Two main concerns can be recognized. The first one is that the actual process of when to use each UML diagram is not clear. The second concern in terms of usability was related to UML design made by others. However, that would probably be true as well for other artifacts generated during software development, such as source code and even documentation.

Table 8.3: Perceived usage of UML - statements 12 to 13

Statement	1	2	3	4	5	m	s	pos
12 - I am interested in knowing more about UML 2.x.		3	3	4		3.1	0.87	4
13 - I worry about the amount of time needed to learn UML extensions from version 1.x to 2.x.		3	4	2	1	3.1	0.99	3

Concluding the questionnaire about UML, statements 12 and 13 are about the

usage of UML. Specifically, the statements were about the transition from versions 1.x to 2.x. Updating versions of tools and modeling languages is normally a concern, because companies have developed many legacy systems, and updating all documents after every change is a time consuming task with little actual gain. The result was not conclusive, with almost equal answers supporting updating to the new version, or keeping using the current version.

Table 8.4: Perceived usefulness of SysML - statements 14 to 18

Statement	1	2	3	4	5	m	s	pos
14 - Using the SysML Requirements Diagram will make my models more expressive.			2	8		3.8	0.42	8
15 - Using the SysML Requirements Diagram relationships will make it easier to trace requirements.			2	7	1	3.9	0.56	8
16 - Using the SysML Requirements Diagram extensions will improve the requirements engineering phase.			4	6		3.6	0.51	6
17 - Using the SysML Requirements Diagram to model all types of requirement will improve the requirements engineering phase.			7	3		3.3	0.48	3
18 - I think it is useful to introduce SysML in the 4+1 view model of architecture.			3	7		3.7	0.48	7

The answers relative to statements 14 to 18 indicate that the SysML Requirements diagram has potential of usefulness in general and within the 4+1 Model of Architecture in particular. Most respondents considered that the diagram is useful to make the models more expressive and that it improves traceability between requirements and other design models, which is fundamental for designing software-intensive systems. SysML was considered to be useful to be used within the 4+1 View Model of Architecture by 7 respondents. However, a remarkable result was that 7 respondents were neutral with one important capability of SysML Requirements diagram, which is to model other types of requirements besides the functional ones.

Table 8.5: Perceived ease of use of SysML - statements 19 to 22

Statement	1	2	3	4	5	m	s	pos
19 - It is easy for me to know the difference between UML and SysML Use Cases.		3	5	2		2.9	0.73	2
20 - It is easy for me to know the difference between each SysML Requirements diagram relationship.		2	6	1	1	3.1	0.87	2
21 - I find it easy to relate SysML Use Cases and SysML Requirements Diagrams.		1	8	1		3	0.47	1
22 - It is easy for me to model requirements using the SysML Requirements diagram.		3	3	4		3.1	0.87	4

Even being a UML profile and sharing UML concepts, constructs and diagrams, details related specifically to SysML were not considered easy to understand and use. Moreover, the relationship and the differences between UML and SysML diagrams for Requirements Engineering are not clear. The results of statements 19 to 22 indicate that even for developers that are knowledgeable about UML, a specific training in SysML is highly recommendable. This is reflected in statements 23 to 25, about the usage of SysML. Although there was an interest in knowing more, there was also concern about the acceptance in real projects and the amount of time needed to learn the language.

Table 8.6: Perceived usage of SysML - statements 23 to 25

Statement	1	2	3	4	5	m	s	pos
23 - I worry about the acceptance of SysML in a real project.		4	4	2		2.8	0.78	4
24 - I worry about the amount of time needed to learn SysML.		7	2	1		2.4	0.7	7
25 - I am interested in knowing more about SysML.			4	6		3.6	0.51	6

The answers relative to statements 26 to 30 indicate that Petri nets are considered by most respondents a useful modeling language. Before the proposed course, only one respondent had had previous contact with Petri nets. Nevertheless, even after a short introduction, the advantages of Petri nets compared to UML were recognized. The capacity of modeling important features of software-intensive systems, such as shared resources and time constraints, and the tool support used to verify, simulate and validate models were appreciated. The introduction of Petri nets in the 4+1 View Model of Architecture was considered positive by 7 respondents. However, from the results it is not clear that the combination of modeling languages is useful.

Table 8.7: Perceived usefulness of Petri nets - statements 26 to 30

Statement	1	2	3	4	5	m	s	pos
26 - Petri nets provide modeling capabilities that are not within UML/SysML.			2	7	1	3.9	0.56	8
27 - Using Petri nets will make my dynamic models more expressive.			2	7	1	3.9	0.56	8
28 - Simulating Petri nets will help me to validate my dynamic models.			2	5	3	4.1	0.73	8
29 - I think it is useful to introduce Petri nets in the 4+1 view model of architecture.			3	6	1	3.8	0.63	7
30 - I think it is useful to combine UML/SysML with Petri nets.			6	4		3.4	0.51	4

Statements 31 and 32 were about the perceived easy of use of Petri nets. According to the answers, it is difficult to draw a clear conclusion, as most employees

responded "neutral" to these questions. Yet, three of them acknowledge that using Petri nets was easy, and that modeling time constraints with Petri nets was easy as well.

Table 8.8: Perceived ease of use of Petri nets - statements 31 to 32

Statement	1	2	3	4	5	m	s	pos
31 - It is easy for me to use Petri nets.		1	6	3		3.2	0.63	3
32 - It is easy for me to model time aspects using Petri nets.		1	6	3		3.2	0.63	3

According to statements 33 to 35, the use of Petri nets generated interest. However, they are worried about the amount of time needed to become competent users of Petri nets, and how well-accepted it would be to have Petri nets in a real project.

Table 8.9: Perceived usage of Petri nets - statements 33 to 35

Statement	1	2	3	4	5	m	s	pos
33 - I worry about the acceptance of Petri nets in a real project.		3	3	4		3.1	0.87	4
34 - I am interested in knowing more about Petri nets.			4	4	2	3.8	0.78	6
35 - I worry about the amount of time needed to learn Petri nets.	1	1	3	4	1	3.3	1.15	2

Statements 36 to 39 are about the usefulness of the 4+1 View Model of Architecture. The model is considered by most of the respondents sufficient, as architecture, to develop software-intensive systems. According to their experience, using this model of architecture improves future maintainability of software-intensive systems.

Table 8.10: Perceived usefulness of the 4+1 View Model of Architecture - statements 36 to 39

Statement	1	2	3	4	5	m	s	pos
36 - The 4+1 view model of architecture is sufficient, as architecture, for developing software-intensive systems.		2		7	1	3.7	0.94	8
37 - All views are necessary for developing software-intensive systems.		1	3	6		3.5	0.7	6
38 - Systems developed using the 4+1 architecture have improved maintainability.			2	7	1	3.9	0.56	8
39 - Overall using the 4+1 view model of architecture is useful in my job.			3	5	2	3.9	0.73	7

According to the responses given to statements 40 and 41, it is relatively easy to understand each view and to know which diagram to use for each view.

Nevertheless, the conclusion with the results of statements 42 and 43 is that the introduction of SysML and Petri nets into the 4+1 View Model of Architecture was not clear by most respondents. This issue should be clarified with intense training.

Table 8.11: Perceived ease of use of "integration into the 4+1 Architecture" - statements 40 to 43

Statement	1	2	3	4	5	m	s	pos
40 - It is easy for me to understand each view.		2	3	5		3.3	0.82	5
41 - It is easy for me to know which diagrams to use for each view.		3	1	6		3.3	0.94	6
42 - It is easy for me to know where to use SysML in the 4+1 views.		6	2	2		2.6	0.84	2
43 - It is easy for me to know where to use Petri nets in the 4+1 views.	1	4	4	1		2.5	0.84	1

8.4.3 Results from the Interviews

Individual interviews were conducted with some of the participants. The purpose was to better understand the responses and ask for more details of the process of software development at the company. Not all participants could be interviewed due to vacation periods or because they were no longer working at the company. In addition, the focus was on interviewing, when possible, those employees whose answers deviate from the group. For instance, in statement 36, the two respondents that do not considered the 4+1 View Model of Architecture sufficient, as architecture, for developing software-intensive systems. These two employees actually believe that additional information specific to the customer viewpoint, without technical details, should be taken into account.

A major concern mentioned during the interviews was related to the customer and the requirements. That confirms the many publications cited in this thesis that consider Requirements Engineering a critical phase in software development. For instance, according to one developer, the customer comes with very specific requirements, making it difficult to change or to adapt. This can be an issue because functional requirements may be possible to implement from a theoretical point of view, but hard to implement in practice. In order to facilitate the implementation, developers may interpret the requirements, which is considered "dangerous" by one manager. Incorrect interpretations may lead to different implementation of certain functionalities and displease the customer.

From the project management point of view, it is difficult to make estimations based only on initial requirements. A solution followed at the company is to make at least part of the design before returning to the customer any informa-

tion about duration and costs. The main issue with this approach is that it is not always feasible with the allocated time.

A major problem faced mainly by managers is that design is normally not done according to the company's standards. Although the importance of having a good design before starting to implement the software is widely recognized both in literature and in practice, developers are eager to start coding too early. This was confirmed by developers during the interview.

The company as a whole makes intense use of UML as modeling language. The language is considered by most of the employees easy to learn and use. Not all diagrams are used, but just the most common ones (Use Case, Sequence, Class), which actually is often seen in practice in other companies. Activity diagrams are occasionally used for process modeling. The company has made modifications to the Sequence diagram. For example, a specific symbol is inserted in the diagram to represent initiation time, and a graphical user interface is added in order to show to the customer the results of each processing step.

According to managers and developers, one major problem faced with UML is the lack of really useful tools. Although many tools are available, normally commercial tools are too expensive, and open source tools are only good for toy examples, but not for real design of large projects, which is common at the company. In addition, the company uses Linux as the platform of development, limiting the number of UML tools.

The applicability of formal methods for system design is a debated theme at the company. Formal methods are not considered useful for some activities such as designing the user interface, but are suitable for communicating systems and processes specification in which the design must be free of errors. More specifically, Petri nets are considered useful, but it is still unclear how they fit into the 4+1 View Model of Architecture and the current company's documentation. As a result, most developers are reluctant to use Petri nets in practice, and think that training is fundamental before they can become competent users.

On the other hand, SysML was considered easy to use and almost directly applicable, as it is an extension of UML. Another advantage of SysML is the flexibility provided by the language and the possibility of creating specific profiles with important company-specific additions. Specifically about the SysML Requirements diagram and Tables, the developers and managers see potential in SysML compared to their old requirements specification method, mainly because it clearly shows relations between requirements, and between requirements and design.

SysML Requirements diagrams and Tables were already applied to two projects at Trinité. A third project, considered large (around 200 user requirements) by the company, is currently being developed using SysML and UML. Trinité is

currently trying to find the best way to incorporate SysML in the company's development process.

8.5 Analysis of Findings

The gap between new academic methods, techniques and processes and their application in industry is common in Systems and Software Engineering (Parnas, 1997; Reifer, 2003; Eckstein and Baumeister, 2004; Connor et al., 2009). The challenge is not only to develop better theories, but also to effectively introduce and use these theories in practice. One way to address this gap is to use industry as a testing environment. Thus, researchers learn what real world problems exist, and if the proposed solutions really contribute to solve problems.

The SysML Requirements diagram may become the language of choice for requirements specification. There is a lack of languages for requirements specification and documentation, which is one reason why natural language is often used almost exclusively to document requirements. There are advantages in using natural language, for instance, for being the primary means of communication among humans. The problem occurs when natural language is the only means of description of requirements, due to its well-known problems, such as ambiguity and lack of standards.

As SysML is a language, not a methodology, it can be added without many problems into the current development process in an organization. There is no need to perform a radical change in the current methodology, which would involve too many risks. The language can be adapted and integrated into the existing methodology and processes. In addition, SysML is based on UML, which is widely known and used, both in academia and industry. As a matter of fact, SysML is considered easily introduced to teams that are already using UML. The major concern is to find the correct relationship between UML and SysML diagrams. During the course, it became clear that the employees could understand the proposed SysML diagrams of the examples. In addition, they could create the proposed models themselves with little help.

SysML is highly customizable and can be extended into families of languages, specific for various domains. Business organizations that develop systems for several different domains may create a family of languages based on a specific standard, and apply them to each domain. Profiles may specialize language semantics, provide new graphical icons and domain-specific model libraries.

Petri nets are useful to design models of software-intensive systems, providing the capacity of modeling important features such as shared resources and time

constraints. Another important characteristic is the availability of tool support that may be used for model verification and validation. Thus, non-functional requirements and constraints can be assessed during the early stages of software development.

The major issue with the applicability of Petri nets is common to other formal methods, as was already discussed in Chapter 2. Despite having few basic elements, mastering the modeling and interpreting results of verification algorithms, even when using tool support, is challenging. This became clear during the course, as the participants in general had difficulties in creating their own models.

Developers normally worry about the amount of time needed to become competent users of Petri nets. In order to master Petri nets as a modeling language and verification tool, intensive training and practical application are both fundamental.

The 4+1 View Model of Architecture is considered easy to understand and to use, and useful to improve the maintainability of systems. However, integrating other modeling languages besides UML and its profiles is an issue. Introducing SysML to create models for each view was considered positive by most employees. On the other hand, it was difficult to understand how Petri nets would fit in the set of models for each view. As a matter of fact, not only it was difficult for the participants to create their own Petri net models, but also the introduction of Petri nets in the architecture was a challenge.

Finally, although training was mentioned as highly recommended, one has to keep in mind that developers are always very busy and have little time to stop their work and learn new technologies. Formal training, in which groups of employees attend classes, is often not what they prefer. In practice, the learning process is frequently done "on the job", by interacting with more experienced employees. As a result, informal training, in which one or more employees explain not at once, but daily during the effective execution of real tasks, has more potential to be successful.

Chapter 9

Epilogue

This chapter closes this thesis. It contains the responses to the research questions (Section 9.1), the final conclusions and recommendations (Section 9.2) and topics for future work (Section 9.3).

9.1 Research Questions Revisited

The main research questions proposed in Chapter 1 are discussed in this section.

9.1.1 Research Question 1

How to improve requirements specification and analysis for Software-Intensive Systems?

Stakeholders frequently change requirements due to various factors. For example, stakeholders may be unsure about their own needs in the beginning of a project, and laws and business processes may change. The main issue is not related to changes in requirements, but to uncontrolled changes. Through correct requirements management, whenever stakeholders ask for changes in requirements, developers have the possibility to uncover where and how this change will impact the system design.

The early introduction of graphical models in Requirements Engineering through a common modeling language (SysML) is proposed in this thesis. Although the UML Use Case diagram has been used to model functional requirements, the SysML Requirements diagram closes a gap by modeling other types of requirements, such as non-functional ones. This is fundamental in the design of software-intensive systems, as non-functional requirements are crucial quality

factors for their success.

Two subareas of the Requirements Engineering process are investigated in this thesis: the Specification and the Analysis.

Specification is improved by identifying, classifying, and relating requirements to each other. The graphical modeling of each requirement helps in improving the requirements documentation. This is performed using the SysML Requirements diagram and the SysML Tables. SysML was introduced at Trinité in the Requirements Engineering phase of their software development process. According to their developers, it clearly adds value when compared to their old method for requirements specification, mainly because the SysML Requirements diagram shows relations between requirements, and between requirements and design. As a result, the introduction of SysML as a language for Requirements Engineering provides a bridge between the text-based requirements and the models of the system. For instance, SysML Requirements can be related to a Use Case or to a SysML Block.

The basic SysML Requirements diagram was extended with new properties. These properties are optional, but may be useful in activities related to requirements analysis and project management, such as release planning and risk evaluation and mitigation. Requirements analysis is improved by identifying relationships between requirements, the type of each relationship, and by tracing requirements through development.

An important phase of the system life cycle is maintenance. Tracing back design models to requirements is desirable in order to facilitate the maintenance phase. The possibility of tracing a requirement during the system development phases is fundamental. The modeling approach presented in this thesis shows that this is possible using the SysML Requirements diagram, including the many relationships provided by SysML, and by using SysML Tables. According to (IEEE, 1998), an important quality of a requirements specification document is that its requirements are traceable. In addition, traceable requirements conform to another important quality factor of a requirements document, which is that the requirements document is modifiable (IEEE, 1998).

9.1.2 Research Question 2

How to specify a Software-Intensive System's Architecture that enables reusability?

The importance of having a well-defined architecture for developing software-intensive systems has been recognized by several authors, as discussed in this thesis. The architecture of the system is an essential component to help in

maintaining the system. Getting it right is a pre-requisite for success. Wrong architecture decisions will lengthen the development process or even derail the final software products.

Architecture can be seen as a high level design of a system. It is useful to express the overall structure at a high level of abstraction. The architecture of a software-intensive system makes the structure of these systems more transparent and systematic.

In this thesis, two types of architectures were proposed: domain architectures and software architectures. Both types are relevant to software-intensive systems. Due to the complexity of software-intensive systems, both types are necessary and complementary.

The domain architecture expresses the organizational procedures, information and business structure of the system for customers. It can be used as an input for the software architecture, and for communicating and making explicit decisions related to business. Typically the domain architecture proposes a family of systems. This is necessary in order to avoid stovepipe systems, as the target family of systems should share many assets. However, the domain architecture is too high-level to be used as a basis for detailed design. The basis for software design is better represented by a software architecture.

The software architecture is useful to identify subsystems, components and their interfaces. In large, software-intensive systems, with multi-disciplinary teams involved in the development phases, important decisions must be documented to be referred in the future. Hence, the design of the software architecture is an important activity in order to facilitate future maintenance of software-intensive systems. In this thesis, a multiple-view architecture is used for the design of systems.

Another important characteristic of the proposed software architecture is that it is the basis for the design of a family of products belonging to the same domain, as a software product line. Therefore, the same architecture is the basis for a family of products, which increases the reusing of artifacts that can vary from source code to complete components and subsystems. Thus, commonalities are easily identified and reuse increased, leading to economic and quality benefits. Additionally, maintainability is enabled by facilitating the creation of multiple systems versions.

9.1.3 Research Question 3

How to model and verify reliability of Software-Intensive Systems?

In order to answer this question, in this thesis a formal method was applied.

Formal methods are key to the activities of verification and validation, which are related to the quality of the final products.

Petri nets present important characteristics that are useful to model distributed real-time systems. Models created with Petri nets can be simulated in a computer-based tool, providing validations of specific scenarios. Although these simulations do not guarantee that models are error-free, they can already detect many design flaws. Thus, when the behavior of the chosen scenarios is as expected, the same models can be formally verified using one or more of the variety of verification tools provided by the Petri net theory. The advantage is that the same models can be used in diverse activities during system design. In this thesis, analysis of Petri nets models is done by using invariant analysis, the reachability graph, and theorem proving with Linear Logic.

An approach in which there is integration of formal and semi-formal languages and methods has been proven to be successful by many authors, in many domains, as already discussed in this thesis. This was the approach used in this thesis. The advantages are clear. Using semi-formal methods in the early stages of the design of software-intensive systems allows one to not spend too much time on formalization, as there are good chances that the initial models are going to be changed. This approach avoids that resources (time, money, personnel) are unnecessarily invested in the early stages of design. Formal methods have, compared to semi-formal methods, a steeper learning curve, and are less suitable to be shown to non-technical stakeholders. Finally, not all models of the design can be or need to be formally specified. Formal methods can be used in phases of the design in which they add value, such as the specification of real-time constraints, resource sharing, and concurrency.

9.2 Final Conclusions and Recommendations

In this section, a number of topics regarding limitations and recommendations based on the thesis are discussed.

9.2.1 Industry as Laboratory

The approach followed in this thesis is to have a close relationship with industry, in what is often called “Industry as Laboratory”. This approach offers advantages for both academia and industry. Working in cooperation with partners from industry bring real problems and cases to researchers, and show them what Software Engineering looks like in practice. The effect is that researchers have real challenges to confirm/refute their theories. In addition, their results are stronger when tested in a real industry environment, not only with toy

projects. The advantage for industry is that researchers are most likely capable to think "outside the box", proposing non-conventional solutions that may have a high impact on problem solving. The industrial partners gets inspiration from results, and are continuously challenged by unbiased and critical people.

Naturally the approach also involve risks. From the industry side, having researchers working with them cannot be seen as cheap consultancy. It must be clear for researchers that industry will only be interested in changes if they see added value for their business. And researchers have to be prepared to learn about industry particularities. As a matter of fact, involved parts must be honest and state clearly what is expected from each other.

9.2.2 Introducing New Technologies

The introduction of new techniques, methods, tools or procedures is challenging in Software Engineering. Not only people are taken out of their comfort zone, but it also takes an effort to adapt legacy systems to the new approaches. As a result, risks may increase, as well as developers dissatisfaction and financial losses. In order to avoid these problems, intensive training is necessary. The common approach for training in many companies is to have a formal course to be held at a specific date and time, followed by everyone at the same time, lectured by an expert. The main issue in this case is the assumption that every person has the same ability to follow the course, which is not true in most cases.

Training should be considered in a more broad aspect than normally proposed, and performed in a gradual, interactive way. New concepts should be introduced not at once, during a whole day course, but smoothly. Training on the job, guided by one or more experts, respecting the level and ability of each person, is most likely to succeed.

9.2.3 Legacy Systems

Maintaining legacy systems involves not only updating them to new hardware and software platforms. Actually these systems were designed to conform to business processes and organizational policies that may have changed or even no longer exist. Thus, new requirements may come up, which have to be implemented respecting requirements already deployed in the current system. There is always the possibility that new requirements are in conflict with the already implemented ones. These factors must be considered when evolving legacy systems.

9.2.4 Software Product Line Architecture

An interesting problem in software architecture is that many software products were developed without a clear architecture definition. The architecture does exists, but only in the designers' minds. Thus, it has to be recovered when the architecture has to be the basis of a family of software products. In this case, architects can not start from scratch anymore. Therefore, defining the architecture of a family of software products is a challenge.

9.2.5 Formal Methods

The applicability of formal methods has been widely discussed. It seems that the application of formal methods to critical systems is already well-recognized by researchers. As discussed in this thesis, some research studies were published arguing that there are so many advantages in using formal methods that it is worth trying. However, practitioners still have doubts about formal methods, considering that they are of limited scope and scalability, and are too difficult and too costly to be used.

The advantages of using formal methods should be made explicit by researchers. Thus, practitioners can understand the gain in terms of software quality. In addition, it should be made clear that it costs too much in downtime and maintenance not to formally prove the correctness of software-intensive systems.

There is a wide range of formal methods and their types. For instance, they can be classified as algebraic or state-based. Choosing the right formal method is currently an issue. A clear example is choosing which type of Petri net to use. The variety is so great that the choice is not obvious. For instance, time can be associated to places, arcs, or transitions, and can vary in terms of semantics. As a result, the choice is often non-optimum, being made by previous experience or the current availability of tools. A more straightforward method for choosing a formal method based on characteristics of the problem is necessary. Thus, the probability of success in applying formal methods in practice may become higher.

9.3 Future Research

In this section, future research is proposed. The topics were chosen based on discussed issues presented previously in the thesis.

9.3.1 Reengineering Legacy Systems

The importance of legacy systems is well-recognized in practice. Because of their importance, Software Engineering researchers should pay more attention to the reengineering of legacy systems. These systems are still very common in many domains such as finance, transportation, health care, and manufacturing. Frequently, legacy systems have to be maintained or ported to different hardware in order to keep being useful. The reason is that in many cases it is too costly and too risky to turn-off an application that has been used with variety degree of success for many years. Another common issue is that these systems need to inter-operate with new applications.

A number of research directions can be followed when reengineering legacy systems. Many systems were developed in the past without a clear architecture definition, or the architecture was implicit. The software architecture has to be retrieved in order to better understand these systems. This is an important step before looking deeply into design models and source code. Documentation of legacy systems (if existing) is frequently ill-structured. Legacy systems were often developed by personnel that may no longer be available to work on them. During maintenance, new requirements may emerge, leading to new design efforts. Currently the most used modeling language is UML, but many legacy systems were designed using other modeling languages. Thus, some integration of modeling languages should be expected. However, without a clear definition of the software architecture, the whole effort has a high probability of failure.

9.3.2 Evaluation of UML and Profiles

UML is recognized by many authors as insufficient to model real-time systems. In order to address this concern, the UML profile MARTE was proposed. According to the OMG, MARTE consists of defining foundations for model-based description of real-time and embedded systems. MARTE is currently an OMG Adopted Beta Specification. The advantages of MARTE will become clear when the profile is tested in real projects.

Another UML profile that is gaining attention in the software and systems development community is SysML. The language has been investigated by many researchers. The first results are promising, but there is still a long way to go to make SysML acceptable and used as widely for Systems Engineering as UML is for Software Engineering. The challenge is even greater than was the acceptance of UML. With SysML, not only the Software Engineering community has to approve, but also many other engineering communities.

9.3.3 Software Architecture

The importance of software architecture in the software life-cycle is widely recognized in theory and practice by many authors, as discussed previously in this thesis. The discipline of Software Architecture has emerged, and plenty of textbooks, journal and conference articles are available, as well as associations and working groups (Kruchten et al., 2006). Specific courses on software architecture are being taught at universities.

Many directions can be followed in software architecture research. Architecture Description Languages (ADLs) are not as well-standardized and used in practice as modeling languages for software design. The reasons behind it are not clear, and should be discovered in order to address this problem. UML, although used as an ADL (Garland and Anthony, 2002), is actually a modeling language whose main target is software design, not architecture.

Pure agilists refuse to recognize the importance of architecture. They have named architecture BDUF (Big Design Up-Front). Agile methods have shown benefits widely described in conferences such as the Agile Conference and Conference on Agile Software Development. A reconciliation of the two communities, each one acknowledging the accomplishments and collaborating with each other can be of benefit for Software Engineering.

9.3.4 Empirical Software Engineering

"If you can not measure it, you can not improve it." - "To measure is to know." (Lord Byron).

Empirical evidence is still lacking in Software Engineering research. Measuring aspects such as quality, usefulness, and utility of Software Engineering artifacts, which include modeling languages, architecture, methods, and techniques, is essential to recognize software development as a real engineering discipline. In addition, knowing what is worth paying attention to can help researchers in improving the body of knowledge of Software Engineering.

Bibliography

ABET, 1941. The Engineers' Council for Professional Development. Science, 94, 456.

Abran, A., Bourque, P., Dupuis, R., Moore, J. W., Tripp, L. L. (Eds.), 2004. Guide to the Software Engineering Body of Knowledge - SWEBOK, 2004th Edition. IEEE Press, Piscataway, NJ, USA.

Abrial, J.-R., 2006. Formal Methods in Industry: Achievements, Problems, Future. In: ICSE '06: Proceedings of the 28th International Conference on Software Engineering.

Abrial, J.-R., 2007. Formal Methods: Theory Becoming Practice. Journal of Universal Computer Science, 13 (5), 619–628.

Adams, D. A., Nelson, R. R., Todd, P. A., 1992. Perceived Usefulness, Ease of Use, and Usage of Information Technology: a Replication. MIS Quarterly, 16 (2), 227–247.

Agarwal, R., Sinha, A. P., 2003. Object-oriented Modeling with UML: a Study of Developers' Perceptions. Communications of the ACM, 46 (9), 248–256.

Almendros-Jiménez, J. M., Iribarne, L., 2005. Describing Use Cases with Activity Diagrams. In: Proceedings of the Metainformatics Symposium.

Almendros-Jiménez, J. M., Iribarne, L., 2007. Describing Use-Case Relationships with Sequence Diagrams. Computer Journal, 50 (1), 116–128.

Anda, B., Hansen, K., Gullesen, I., Thorsen, H. K., 2006. Experiences from Introducing UML-based Development in a Large Safety-Critical Project. Empirical Software Engineering, 11 (4), 555–581.

ANSI/IEEE, 2000. ANSI/IEEE Std 1471 Recommended Practice for Architectural Description of Software-Intensive Systems.

Avison, D. E., Lau, F., Myers, M. D., Nielsen, P. A., 1999. Action Research. Communications of the ACM, 42 (1), 94–97.

AVV, 2006. Auditing on RTMS, private document made by *Adviesdienst Verkeer en Vervoer*.

Balmelli, L., Brown, D., Cantor, M., Mott, M., 2006. Model-driven Systems Development. IBM Systems Journal, 45 (3), 569–586.

Bar-Yam, Y., 2003. When Systems Engineering Fails - Toward Complex Systems Engineering. In: Proceedings of the International Conference on Systems, Man & Cybernetics. Vol. 2.

Baskerville, R., 1999. Investigating Information Systems with Action Research. Communications of the AIS, 2 (4), 1–32.

Baskerville, R., Wood-Harper, A., 1996. A Critical Perspective on Action Research as a Method for Information Systems Research. Journal of Information Technology, 11 (3), 235–246.

Bass, L., Clements, P., Kazman, R., 2003. Software Architecture in Practice. Addison-Wesley Professional, Reading, MA, USA.

Bause, F., Kritzinger, P. S., 2002. Stochastic Petri Nets - An Introduction to the Theory. Vieweg Verlag, Berlin, Germany.

Beck, K., 1999. Extreme Programming Explained: Embrace Change. Addison-Wesley Professional, Boston, MA, USA.

Berry, D. M., 2004. The Inevitable Pain of Software Development: Why There Is No Silver Bullet. In: Radical Innovations of Software and Systems Engineering in the Future. Lecture Notes in Computer Science.

Bézivin, J., 2005. On the Unification Power of Models. Software and System Modeling, 4 (2), 171–188.

Bézivin, J., 2006. Model Driven Engineering: An Emerging Technical Space. Vol. 4143 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany.

Bicarregui, J., Dick, J., Woods, E., 1996. Quantitative Analysis of an Application of Formal Methods. In: FME '96: Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods.

Boehm, B. W., 1973. Software and Its Impact: A Quantitative Assessment. Datamation, 19 (5), 48–59.

Boehm, B. W., 1983. Seven Basic Principles of Software Engineering. Journal of Systems and Software, 3 (1), 3–24.

Boehm, B. W., 2006. A View of 20th and 21st Century Software Engineering. In: ICSE '06: Proceedings of the 28th International Conference on Software Engineering.

Boehm, B. W., Abts, C., Chulani, S., 2000. Software Development Cost Estimation Approaches - A Survey. Annals of Software Engineering, 10, 177–205.

Boehm, B. W., Papaccio, P., 1988. Understanding and Controlling Software Costs. IEEE Transactions on Software Engineering, 14 (10), 1462–1477.

Boehm, B. W., Turner, R., 2003. Balancing Agility and Discipline: A Guide for the Perplexed. Addison-Wesley Professional, Boston, MA, USA.

Boehm, B. W., Valerdi, R., Honour, E., 2008. The ROI of Systems Engineering: Some Quantitative Results for Software-Intensive Systems. Systems Engineering, 11 (3), 221–234.

Booch, G., 1994. Object-Oriented Analysis and Design with Applications (2nd ed.). Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA.

Booch, G., 2007. The Economics of Architecture-First. IEEE Software, 24 (5), 18–20.

Booch, G., Maksimchuk, R. A., Engel, M. W., Young, B. J., Conallen, J., Houston, K. A., 2007. Object-Oriented Analysis and Design with Applications (3rd Edition). Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

Booch, G., Rumbaugh, J., Jacobson, I., 2005. The Unified Modeling Language User Guide, 2nd Edition. Addison-Wesley Professional, Boston, MA, USA.

Bosch, J., 2000. Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. ACM Press/Addison-Wesley Professional, New York, NY, USA.

Bourque, P., Dupuis, R., Abran, A., Moore, J. W., Tripp, L., Wolff, S., 2002. Fundamental Principles of Software Engineering - A Journey. Journal of Systems and Software, 62 (1), 59 – 70.

Bowen, J. P., Hinchey, M. G., 1995. Seven More Myths of Formal Methods. IEEE Software, 12 (4), 34–41.

Bowen, J. P., Hinchey, M. G., 2005. Ten Commandments Revisited: A Ten-Year Perspective on the Industrial Application of Formal Methods. In: FMICS '05: Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems.

Brave, Y., 1993. Control of Discrete Event Systems Modeled as Hierarchical State Machines. IEEE Transactions on Automatic Control, 38 (12), 1803–1819.

Briggs, R. O., Adkins, M., Mittleman, D., Kruse, J., Miller, S., Nunamaker, Jr., J. F., 1998. A Technology Transition Model Derived from Field Investigation of GSS Use Aboard the U.S.S. CORONADO. Journal of Management Information Systems, 15 (3), 151–195.

Broadfoot, G. H., Hopcroft, P. J., 2005. Introducing Formal Methods into Industry using Cleanroom and CSP. Last accessed on the 19th of November, 2009.

Brooks, F. P., 1987. No Silver Bullet: Essence and Accidents of Software Engineering. Computer, 20 (4), 10–19.

Brooks, F. P., 1995. The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley Professional, Boston, USA.

Brown, A. W., Conallen, J., Tropeano, D., 2005. Model-Driven Software Development. Springer-Verlag, Berlin, Germany, Ch. Introduction: Models, Modeling, and Model-Driven Architecture (MDA).

Brown, A. W., McDermid, J. A., 2007. The Art and Science of Software Architecture. International Journal of Cooperative Information Systems, 16 (3/4), 439–466.

Brown, C., 1989. Relating Petri Nets to Formulas of Linear Logic. Tech. Rep. ECS-LFCS-89-87, University of Edinburgh.

Broy, M., 2006. The 'Grand Challenge' in Informatics: Engineering Software-Intensive Systems. Computer, 39 (10), 72–80.

Broy, M., 2007. From Formal Methods to System Modeling. Lecture Notes in Computer Science. Springer, Berlin, Germany.

Buxton, J. N., Randell, B. (Eds.), 1970. Software Engineering Techniques: Report of a Conference Sponsored by the NATO Science Committee. NATO Science Committee, Brussels, Belgium.

Calhoun, C., 1995. Critical Social Theory: Culture, History, and the Challenge of Difference. Wiley Blackwell, Oxford, UK.

CALTRANS, 2007. Systems Engineering Guidebook for ITS. Tech. rep., California Department of Transportation - Division of Research and Innovation.

Cambridge Systematics, 2001. Twin Cities Ramp Meter Evaluation. Tech. rep., Highway Research Board.

Campos, J., Merseguer, J., 2006. Petri Nets and Other Models of Concurrency. Vol. 4024 of Lecture Notes in Computer Science. Springer, Berlin, Germany, Ch. On the Integration of UML and Petri nets in Software Development.

Carlshamre, P., Sandahl, K., Lindvall, M., Regnell, B., Natt och Dag, J., 2001. An Industrial Survey of Requirements Interdependencies in Software Product Release Planning.

Casicato, L., Cass, S., 1962. Pilot Study of the Automatic Control of Traffic Signals by a General Purpose Electric Computer. Tech. Rep. 338, Highway Research Board.

Cassandras, C. G., Lafortune, S., 1999. Introduction to Discrete Event Systems. The International Series on Discrete Event Dynamic Systems. Kluwer Academic Publishers, Norwell, MA, USA.

Charette, R. N., 2005. Why Software Fails. IEEE Spectrum, 42 (9), 42–49.

Cheng, S.-F., Epelman, M., Smith, R., 2006. CoSIGN: A Parallel Algorithm for Coordinated Traffic Signal Control. IEEE Transactions on Intelligent Transportation Systems, 7 (4), 551–564.

Cheung, K., Cheung, T., Chow, K., 2006. A Petri-Net-Based Synthesis Methodology for Use-Case-Driven System Design. Journal of Systems and Software, 79 (6), 772 – 790.

Choppy, C., Mayero, M., Petrucci, L., 2008. Experimenting Formal Proofs of Petri Nets Refinements. Electronic Notes Theoretical Computer Science, 214, 231–254.

Christensen, S., Petrucci, L., 2000. Modular Analysis of Petri Nets. The Computer Journal, 43 (3), 224–242.

Clarke, E. M., Wing, J. M., 1996. Formal Methods: State of the Art and Future Directions. ACM Computing Surveys, 28 (4), 626–643.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J., 2002. Documenting Software Architectures: Views and Beyond. Addison-Wesley Professional, Boston, MA, USA.

Clements, P., Northrop, L., 2001. Software Product Lines: Practices and Patterns, 3rd Edition. Addison-Wesley Professional, Boston, MA, USA.

Connor, A. M., Buchan, J., Petrova, K., 2009. Bridging the Research-Practice Gap in Requirements Engineering through Effective Teaching and Peer Learning. In: ITNG '09: Proceedings of the 2009 Sixth International Conference on Information Technology: New Generations. IEEE Computer Society.

Cooper, K., Ito, M., 2002. Formalizing a Structured Natural Language Requirements Specification Notation. In: Proceedings of the International Council on Systems Engineering Symposium. Vol. CDROM index 1.6.2.

Corder, G. W., Foreman, D. I., 2009. Nonparametric Statistics for Non-Statisticians: A Step-by-Step Approach. John Wiley & Sons, Hoboken, NJ, USA.

Creswell, J., 2008. Research Design: Qualitative, Quantitative and Mixed Methods Approaches. Sage Publications, Thousand Oaks, CA, USA.

Damian, D., Chisan, J., Vaidyanathasamy, L., Pal, Y., 2005. Requirements Engineering and Downstream Software Development: Findings from a Case Study. Empirical Software Engineering, 10 (3), 255–283.

Damian, D., Zowghi, D., Vaidyanathasamy, L., Pal, Y., 2004. An Industrial Case Study of Immediate Benefits of Requirements Engineering Process Improvement at the Australian Center for Unisys Software. Empirical Software Engineering, 9 (1-2), 45–75.

Davis, A. M., 2003. The Art of Requirements Triage. Computer, 36 (3), 42–49.

Davis, F. D., 1989. Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology. MIS Quarterly, 13 (3), 319–339.

Davis, J. F., 2005. The Affordable Application of Formal Methods to Software Engineering. ACM SIGAda Ada Letters, XXV (4), 57–62.

Dedrick, J., Gurbaxani, V., Kraemer, K. L., 2003. Information Technology and Economic Performance: A Critical Review of the Empirical Evidence. ACM Computing Surveys, 35 (1), 1–28.

Demmou, D., Khalfaoui, S., Guilhem, E., Valette, R., 2004. Critical Scenarios Derivation Methodology for Mechatronic Systems. Reliability Engineering & System Safety, 84 (1), 33–44.

Deng, L. Y., Liang, H. C., Wang, C.-T., Wang, C.-S., Hung, L.-P., 2005. The Development of the Adaptive Traffic Signal Control System. In: ICPADS '05: Proceedings of the 11th International Conference on Parallel and Distributed Systems.

Denning, P. J., 1997. A New Social Contract for Research. Communications of the ACM, 40 (2), 132–134.

DiCesare, F., Kulp, P. T., Gile, M., List, G., 1994. The Application of Petri Nets to the Modeling, Analysis and Control of Intelligent Urban Traffic Networks. In: Proceedings of the 15th International Conference on Application and Theory of Petri Nets.

Diev, S., 2006. Use Cases Modeling and Software Estimation: Applying Use Case Points. ACM Software Engineering Notes, 31 (6), 1–4.

Dijkstra, E. W., 1965. Cooperating Sequential Processes. Tech. Rep. EWD-123, Technical University of Eindhoven.

Dijkstra, E. W., 1970. Structured Programming. In: Buxton, J., Randell, B. (Eds.), Software Engineering Techniques. NATO Science Committee.

Dijkstra, E. W., 1972. The Humble Programmer. Communications of the ACM, 15 (10), 859–866.

Dijkstra, E. W., 1979. Programming Considered as a Human Activity. Yourdon Press, Upper Saddle River, NJ, USA.

Dijkstra, E. W., 1982. Selected Writings on Computing: A Personal Perspective. Springer-Verlag, New York, NY, USA.

Dobing, B., Parsons, J., 2006. How UML Is Used. Communications of the ACM, 49 (5), 109–113.

Dotolia, M., Fanti, M. P., 2006. An Urban Traffic Network Model Via Coloured Timed Petri Nets. Control Engineering Practice, 14 (10), 1213–1229.

Douglass, B. P., 2004. Real Time UML: Advances in the UML for Real-Time Systems (3rd Edition). Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

Dutilleul, S., Defossez, F., Bon, P., 2006. Safety Requirements and P-Time Petri Nets: A Level Crossing Case Study. IMACS Multiconference on Computational Engineering in Systems Applications,.

Easterbrook, S., Singer, J., Storey, M., Damian, D., 2007. Selecting Empirical Methods for Software Engineering Research. Springer Verlag, London, UK.

Eckstein, J., Baumeister, H. (Eds.), 2004. Extreme Programming and Agile Processes in Software Engineering. Vol. 3092 of Lecture Notes in Computer Science.

Eichner, C., Fleischhack, H., Meyer, R., Schrimpf, U., Stehno, C., 2005. SDL Forum. Ch. Compositional Semantics for UML 2.0 Sequence Diagrams Using Petri Nets.

Eisenhauer, G., Schwan, K., Bustamante, F., 2006. Publish-Subscribe for High-Performance Computing. IEEE Internet Computing, 10 (1), 40–47.

Engberg, U., Winskel, G., 1990. Petri Nets as Models of Linear Logic. In: Proceedings of Colloquium on Trees in Algebra and Programming.

Eugster, P. T., Felber, P. A., Guerraoui, R., Kermarrec, A.-M., 2003. The Many Faces of Publish/Subscribe. ACM Computing Surveys, 35 (2), 114–131.

Eurlings, C., 2008. Tweede Kamer der Staten-Generaal - Mobiliteitsbeleid. Last accessed on the 8th of September, 2009.

Eurostat, 2006. Keep Europe Moving - Sustainable Mobility for our Continent Mid-term Review of the European Commissions 2001 Transport White Paper. Tech. rep., European Commission - Directorate General Energy and Transport, last accessed on the 22th of August, 2009.

Eurostat, 2009. Road Freight Transport Vademecum. Tech. rep., European Commission - Directorate General Energy and Transport., last accessed on the 22th of August, 2009.

Febbraro, A. D., Giglio, D., 2006. Urban Traffic Control in Modular/Switching Deterministic-Timed Petri Nets. In: Proceedings of the Eleventh IFAC Symposium on Control in Transportation Systems.

Febbraro, A. D., Giglio, D., Sacco, N., December 2004. Urban Traffic Control Structure Based on Hybrid Petri Nets. IEEE Transactions on Intelligent Transportation Systems, 5 (4), 224–237.

FHWA, 1998. Developing Traffic Signal Control Systems Using the National ITS Architecture. Tech. rep., U. S. Department of Transportation - Federal Highway Administration, Washington, DC.

FHWA, 2006. Traffic Detector Handbook: Third Edition Volume I. Tech. Rep. FHWA-HRT-06-108, U.S. Department of Transportation - Federal Highway Administration.

Fiege, L., Cilia, M., Muhl, G., Buchmann, A., 2006a. Publish-Subscribe Grows Up: Support for Management, Visibility Control, and Heterogeneity. IEEE Internet Computing, 10 (1), 48–55.

Fiege, L., Muhl, G., Pietzuch, P. R., 2006b. Distributed Event-based Systems. Springer-Verlag, Berlin, Germany.

Flowers, S., 1996. Software Failure: Management Failure - Amazing Stories and Cautionary Tales. John Wiley & Sons, New York, NY, USA.

Galin, D., Avrahami, M., 2006. Are CMM Program Investments Beneficial? Analyzing Past Studies. IEEE Software, 23 (6), 81–87.

Gallego, J.-L., Farges, J.-L., Henry, J.-J., 1996. Design by Petri Nets of an Intersection Signal Controller. Transportation Research Part C: Emerging Technologies, 4 (4), 231–248.

Garlan, D., Shaw, M., 1993. An Introduction to Software Architecture. World Scientific Publishing, New Jersey, NJ, USA.

Garland, J., Anthony, R., 2002. Large-Scale Software Architecture: A Practical Guide using UML. John Wiley & Sons, Inc., New York, NY, USA.

Gehlot, V., Gunter, C. A., 1989. Nets as Tensor Theories. In: Michelis, G. D. (Ed.), Proceedings of the 10th International Conference on Application and Theory of Petri Nets. Bonn, Germany.

George, C., Haxthausen, A. E., Hughes, S., Milne, R., Prehn, S., Pedersen, J. S., 1995. The RAISE Development Method. The BCS Practitioners Series. Prentice Hall International, New York, NY, USA.

Girard, J.-Y., 1987. Linear Logic. Theoretical Computer Science, 50 (1), 1–102.

Girard, J.-Y., 1995. Linear Logic: Its Syntax and Semantics. In: Girard, J.-Y., Lafont, Y., Regnier, L. (Eds.), Advances in Linear Logic (Proc. of the Workshop on Linear Logic, Cornell University). No. 222. Cambridge University Press.

Girault, C., Valk, R., 2002. Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Girault, F., Pradier-Chezalviel, B., Valette, R., 1997. A Logic for Petri nets. Journal Européen des Systèmes Automatisés, 31 (3), 525–542.

Glass, R. L., 1999. The Realities of Software Technology Payoffs. Communications of the ACM, 42 (2), 74–79.

Glass, R. L., 2006. Software Conflict 2.0: The Art And Science of Software Engineering. Developer.* Books.

Glinz, M., 2000. Problems and Deficiencies of UML as a Requirements Specification Language. In: IWSSD '00: Proceedings of the 10th International Workshop on Software Specification and Design.

Gomaa, H., 2000. Designing Concurrent, Distributed, and Real-Time Applications with UML. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Gotel, O. C. Z., Finkelstein, C. W., 1994. An Analysis of the Requirements Traceability Problem. In: International Conference on Requirements Engineering.

Greer, D., 2005. Requirements Engineering for Sociotechnical Systems. Idea-Group, London, UK, Ch. Requirements Prioritisation for Incremental and Iterative Development.

Hall, A., 1990. Seven Myths of Formal Methods. IEEE Software, 7 (5), 11–19.

Hall, T., Beecham, S., Rainer, A., 2002. Requirements Problems in Twelve Companies: An Empirical Analysis. IEE Proceedings for Software, 149 (5), 153–160.

Harel, D., 1987. Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming, 8 (3), 231–274.

Hatley, D. J., Pirbhai, I. A., 1987. Strategies for Real-Time System Specification. John Wiley & Sons, New York, NY, USA.

Hecht, H., 1999. Systems Engineering for Software-Intensive Projects. In: ASSET '99: Proceedings of the 1999 IEEE Symposium on Application - Specific Systems and Software Engineering and Technology.

Henderson-Sellers, B., 2005. UML - the Good, the Bad or the Ugly? Perspectives from a Panel of Experts. Software and System Modeling, 4 (1), 4–13.

Hevner, A. R., March, S. T., Park, J., Ram, S., 2004. Design Science in Information Systems Research. MIS Quarterly, 28 (1), 75–105.

Hinchey, M., Jackson, M., Cousot, P., Cook, B., Bowen, J. P., Margaria, T., 2008. Software Engineering and Formal Methods. Communications of the ACM, 51 (9), 54–59.

Hoenicke, J., Olderog, E.-R., 2002. CSP-OZ-DC: A Combination of Specification Techniques for Processes, Data and Time. Nordic Journal of Computing, 9 (4), 301–334.

Hofmann, H. F., Lehner, F., 2001. Requirements Engineering as a Success Factor in Software Projects. IEEE Software, 18 (4), 58–66.

Hruz, B., Zhou, M. C., 2007. Modeling and Control of Discrete-Event Dynamic Systems: with Petri Nets and Other Tools. Springer Verlag, London, UK.

Huang, H., Kirchner, H., 2009. Policy Composition Based on Petri Nets. In: Proceedings of the Computer Software and Applications Conference. Vol. 2.

Huang, Y.-S., 2006. Design of Traffic Light Control Systems Using Statecharts. Computer Journal, 49 (6), 634–649.

Huddart, K., 1999. Advances in Mobile Information Systems. Artech House, London, UK, Ch. Traffic Control.

Hunt, P., Robertson, D., Bretherton, R., Winton, R., 1981. SCOOT - A Traffic Responsive Method of Coordinating Signals. Tech. Rep. TRRL Laboratory Report 1014, Transport and Road Research Laboratory, Crowthrone, Berkshire, UK.

IEEE, 1998. IEEE Recommended Practice for Software Requirements Specifications. Tech. rep.

IEEE/EIA, 1996. IEEE/EIA 12207.0-1996//ISO/ IEC12207:1995, Industry Implementation of Int. Std. ISO/IEC 12207:95 Standard for Information Technology- Software Life Cycle Processes. Tech. rep.

Jackson, M., 1995. Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices. Addison-Wesley Professional, New York, NY, USA.

Jackson, M., 2003. Why Software Writing is Difficult and Will Remain So. Information Processing Letters, 88 (1-2), 13 – 25.

Jackson, M. A., 1983. System Development. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Jacobson, I., 1992. Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley Professional, Reading, MA, USA.

Jacobson, I., 2004. Use Cases - Yesterday, Today, and Tomorrow. Software and System Modeling, 3 (3), 210–220.

Jamieson, S., 2004. Likert Scales: How to (ab)use them. Med Educ, 38 (12), 1217–1218.

Jasperson, J. S., Carter, P. E., Zmud, R. W., 2005. A Comprehensive Conceptualization of the Post-Adoptive Behaviors Associated with IT-enabled Work Systems. MIS Quarterly, 29 (3), 525–557.

Jeng, M., DiCesare, F., 1995. Synthesis Using Resource Control Nets for Modeling Shared-Resource Systems. IEEE Transactions on Robotics and Automation, 11 (3), 317–327.

Jensen, K., Kristensen, L., 2009. Coloured Petri Nets. Springer-Verlag, Berlin, Germany.

Jesty, P. H., Gaillet, J.-F., Burkert, A., Avontuur, V., Schulz, H. J., Franco, G., 2000. European ITS Framework Architecture. Tech. rep., KAREN.

Johnson, C., 2002. Forensic Software Engineering: Are Software Failures Symptomatic of Systemic Problems? Safety Science, 40 (9), 835–847.

Johnson, C., 2003. Failure in Safety-Critical Systems: A Handbook of Accident and Incident Reporting. University of Glasgow Press, Glasgow, Scotland.

Jones, C., February 2006. The Economics of Software Maintenance in the Twenty First Century, Version 3. Last accessed on the 20th of August, 2009.

Juristo, N., Moreno, A. M., Silva, A., 2002. Is the European Industry Moving Toward Solving Requirements Engineering Problems? IEEE Software, 19 (6), 70–77.

Kamsties, E., 2005. Understanding Ambiguity in Requirements Engineering. In: Aurum, A., Wohlin, C. (Eds.), Engineering and Managing Software Requirements. Springer-Verlag.

Karkinsky, D., Schneider, S. A., Treharne, H., 2007. Combining Mobility with State. In: Integrated Formal Methods.

Khansa, W., Aygaline, P., Denat, J., 1996. Structural Analysis of P-Time Petri Nets. In: Symposium on Discrete Events and Manufacturing Systems: Proceedings of the CESA'96 IMACS Multiconference.

Klein, H. K., Myers, M. D., 1999. A Set of Principles for Conducting and Evaluating Interpretive Field Studies in Information Systems. MIS Quarterly, 23 (1), 67–93.

Klein, L., 2001. Sensor Technologies and Data Requirements for ITS. Artech House, Boston, USA.

Komi-Sirviö, S., Tihinen, M., 2003. Great Challenges and Opportunities of Distributed Software Development - An Industrial Survey. In: Proceedings of the Fifteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2003).

Kronlöf, K. (Ed.), 1993. Method Integration: Concepts and Case Studies. John Wiley & Sons, New York, NY, USA.

Kruchten, P., 1995. The 4+1 View Model of Architecture. IEEE Software, 12 (6), 42–50.

Kruchten, P., 2003. The Rational Unified Process: An Introduction. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Kruchten, P., Obbink, H., Stafford, J., 2006. The Past, Present, and Future for Software Architecture. IEEE Software, 23 (2), 22–30.

Lano, K., 1991. Z++, An Object-orientated Extension to Z. In: Proceedings of the Fifth Annual Z User Meeting on Z User Workshop.

Larsen, P. G., Fitzgerald, J., Brookes, T., 1996. Applying Formal Specification in Industry. IEEE Software, 13 (3), 48–56.

Lavagno, L., Martin, G., Selic, B. (Eds.), 2003. UML for Real: Design of Embedded Real-Time Systems. Kluwer Academic Publishers, Norwell, MA, USA.

Lea, D., Vinoski, S., Vogels, W., 2006. Guest Editors' Introduction: Asynchronous Middleware and Services. IEEE Internet Computing, 10 (1), 14–17.

Lee, J., Chi, S., 2005. Using Symbolic DEVS Simulation to Generate Optimal Traffic Signal Timings. Simulation, 81 (2), 153–170.

Leist, S., Zellner, G., 2006. Evaluation of Current Architecture Frameworks. In: SAC '06: Proceedings of the 2006 ACM Symposium on Applied Computing.

Likert, R., 1932. A Technique for the Measurement of Attitudes. Archives of Psychology, 22 (140), 1–55.

Lin, L., Nan, T., Xiangyang, M., Fubing, S., 2003. Implementation of Traffic Lights Control Based on Petri Nets. In: Proceedings of the 6th IEEE International Conference on Intelligent Transportation Systems. Vol. 2.

Lindgren, M., Norstrom, C., Wall, A., Land, R., 2008. Importance of Software Architecture during Release Planning. In: Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008).

List, G. F., Cetin, M., 2004. Modeling Traffic Signal Control Using Petri Nets. IEEE Transactions on Intelligent Transportation Systems, 5 (3), 177–187.

Ludewig, J., 2003. Models in Software Engineering. Software and System Modeling, 2 (1), 5–14.

Luisa, M., Mariangela, F., Pierluigi, I., 2004. Market Research for Requirements Analysis Using Linguistic Tools. Requirements Engineering, 9 (1), 40–56.

Luqi, Goguen, J. A., 1997. Formal Methods: Promises and Problems. IEEE Software, 14 (1), 73–85.

Lutz, R. R., 1993. Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems. In: Proceedings of the IEEE International Symposium on Requirements Engineering.

Maccubbin, R., Staples, B., Mercer, M., Kabir, F., Abedon, D., Bunch, J., 2005. Intelligent Transportation Systems - Benefits, Costs and Lessons Learned. Tech. Rep. FHWA-OP-05-002, Mitretek Systems and Federal Highway Administration.

McDonald, M., Hall, R., Keller, H., Hecht, C., Fakler, O., Klijnhout, J., Mauro, V., Spence, A., 2006. Intelligent Transport Systems in Europe: Opportunities for Future Research. World Scientific Publishing, Singapore.

Mcilroy, D., 1968. Mass-Produced Software Components. In: Naur, P., Randell, B. (Eds.), Proceedings of the 1st International Conference on Software Engineering. Garmisch, Germany.

McQueen, B., McQueen, J., 1999. Intelligent Transportation Systems Architectures. Artech House Publishers, Norwood, MA, USA.

Menand, L., 1997. Pragmatism: A Reader. Vintage Press, New York, NY, USA.

Mernik, M., Heering, J., Sloane, A. M., 2005. When and How to Develop Domain-Specific Languages. ACM Computing Surveys, 37 (4), 316–344.

Minor, O., Armarego, J., 2005. Requirements Engineering: a Close Look at Industry Needs and Model Curricula. Australian Journal of Information Systems, 13 (1), 192–208.

Moody, D. L., 2002. Comparative Evaluation of Large Data Model Representation Methods: The Analyst's Perspective. In: Proceedings of the International Conference on Conceptual Modeling.

Mueller, E. A., 1970. Aspects of the History of Traffic Signals. IEEE Transactions on Vehicular Technology, 19 (1), 6–17.

Muller, G., van de Laar, P., 2009. Researching Reference Architectures and their Relationship with Frameworks, Methods, Techniques, and Tools. In: Proceedings of the 7th Annual Conference on Systems Engineering Research (CSER 2009).

Murata, T., 1989. Petri nets: Properties, Analysis and Applications. Proceedings of the IEEE, 77 (4), 541–580.

OMG, 2006. Meta-Object Facility (MOF) Core Specification - Version 2.0.

OMG, 2007. Unified Modeling Language (UML): Superstructure - version 2.1.2.

OMG, 2008a. Systems Modeling Language (SysML) - Version 1.1.

OMG, 2008b. UML Profile for MARTE, Beta 2.

OmniTRANS, 2006. OmniTRANS - Offering the Best of Both Worlds. Traffic Engineering & Control, 47 (9), 1–3.

Page, V., Dixon, M., Bielkowicz, P., 2003. Object-Oriented Graceful Evolution Monitors. Vol. 2817 of Lecture Notes in Computer Science. Springer.

Paige, R. F., 1997. Formal Method Integration Via Heterogeneous Notations. Ph.D. thesis, University of Toronto, Canada.

Paige, R. F., 1998. Comparing Extended Z with a Heterogeneous Notation for Reasoning about Time and Space. In: ZUM'98: Proceedings of the 11th International Conference of Z Users on The Z Formal Specification Notation.

Paige, R. F., 1999. When are Methods Complementary? Information & Software Technology, 41 (3), 157–162.

Parnas, D. L., 1972. On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM, 15 (12), 1053–1058.

Parnas, D. L., 1976. On the Design and Development of Program Families. IEEE Transactions on Software Engineering, 2 (1), 1–9.

Parnas, D. L., 1997. Software Engineering: an Unconsummated Marriage. Communications of the ACM, 40 (9), 128.

Parnas, D. L., Clements, P., 1986. A Rational Design Process: How and Why to Fake It. IEEE Transactions on Software Engineering, 12 (2), 251–257.

Parviainen, P., Tihinen, M., Lormans, M., van Solingen, R., 2004. Requirements Engineering: Dealing with the Complexity of Sociotechnical Systems Development. IdeaGroup Inc, Ch. 1.

Petri, C. A., 1962. Kommunikation mit Automaten. Ph.D. thesis, Institut für instrumentelle Mathematik, Bonn, Germany.

PIARC, 1999. ITS Handbook 2000 - Recommendations from the World Road Association (PIARC). Artech House, London, UK.

PMI, 2008. A Guide to the Project Management Body of Knowledge, 4th Edition. PMI, Pennsylvania, USA.

Pohl, K., Böckle, G., van der Linden, F. J., 2005. Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Berlin, Germany.

Polack, F., Whiston, M., Mander, K., 1993. The SAZ Project: Integrating SSADM and Z. In: FME '93: Proceedings of the First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods.

Pomello, L., Rozenberg, G., Simone, C., 1992. A Survey of Equivalence Notions for Net Based Systems. In: Advances in Petri Nets 1992, The DEMON Project.

Potts, C., 1993. Software-Engineering Research Revisited. IEEE Software, 10 (5), 19–28.

Pressman, R. S., 2005. Software Engineering: A Practitioner's Approach. McGraw-Hill, Inc., New York, NY, USA.

Ramesh, B., Jarke, M., 2001. Toward Reference Models for Requirements Traceability. IEEE Transactions on Software Engineering, 27 (1), 58–93.

Recker, J. C., 2007. Why Do We Keep Using A Process Modelling Technique? In: Proceddings of the 18th Australasian Conference on Information Systems.

Reifer, D. J., 2003. Is the Software Engineering State of the Practice Getting Closer to the State of the Art? IEEE Software, 20 (6), 78–83.

Reisig, W., 1985. Petri nets: An Introduction. Springer-Verlag New York, Inc., New York, NY, USA.

Ribeiro, O., Fernandes, J. M., 2006. Some Rules to Transform Sequence Diagrams into Coloured Petri Nets. In: K., J. (Ed.), Proceedings of the 7th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN 2006).

Rijkswaterstaat, 2003. Handbook for Sustainable Traffic Management - A Guide for Users. Tech. rep., Rijkswaterstaat, Rotterdam.

Robertson, D., 1969. TRANSYT: A Traffic Network Study Tool. Tech. Rep. LR 253, Transport and Road Research Laboratory, Crowthrone, Berkshire, UK.

Robertson, S., Robertson, J., 2006. Mastering the Requirements Process (2nd Edition). Addison-Wesley Professional, ACM Press/Addison-Wesley Publishing Co. New York, NY, USA.

Robinson, W. N., Pawlowski, S. D., Volkov, V., 2003. Requirements Interaction Management. ACM Computing Surveys, 35 (2), 132–190.

Roess, R. P., Prassas, E. S., McShane, W. R., 2003. Traffic Engineering, 3rd Edition. Prentice Hall, New Jersey, NJ, USA.

Rossi, M., Sein, M., 2003. Design Research Workshop: A Proactive Research Approach. In: Proceedings of the 26th Information Systems Research Seminar in Scandinavia (IRIS 2003).

Rouse, W., 2003. Engineering Complex Systems: Implications for Research in Systems Engineering. IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews, 33 (2), 154–156.

Rozman, T., Horvat, R. V., Rozman, I., 2008. Modeling the Standard Compliant Software Processes in the University Environment. Business Process Management Journal, 14 (1), 53 – 64.

Sahraoui, A.-E.-K., 2005. Requirements Traceability Issues: Generic Model, Methodology And Formal Basis. International Journal of Information Technology and Decision Making, 4 (1), 59–80.

Saiedian, H., 1996. An Invitation to Formal Methods. Computer, 29 (4), 16–17.

Semmens, L., France, R. B., Docker, T. W. G., 1992. Integrated Structured Analysis and Formal Specification Techniques. Computer Journal, 35 (6), 600–610.

Simons, A. J. H., 1999. Use Cases Considered Harmful. In: TOOLS '99: Proceedings of the Technology of Object-Oriented Languages and Systems.

Sjoberg, D., Dyba, T., Jorgensen, M., 2007. The Future of Empirical Methods in Software Engineering Research. In: FOSE '07: Future of Software Engineering.

Smith, G., 2008. Extending Formal Methods for Software-Intensive Systems. Springer-Verlag, Berlin, Germany.

Snook, C., Butler, M., 2006. UML-B: Formal Modeling and Design Aided by UML. ACM Transactions on Software Engineering Methodologies, 15 (1), 92–122.

Soares, M. S., Julia, S., Vrancken, J., 2008. Real-time Scheduling of Batch Systems using Petri Nets and Linear Logic. Journal of Systems and Software, 81 (11), 1983–1996.

Soares, M. S., Vrancken, J., 2007a. A Multi-Agent Distributed Architecture for Road Traffic Control. In: Proceedings of the 6th European Congress and Exhibition on Intelligent Transport Systems and Services (ITS 2007).

Soares, M. S., Vrancken, J., 2007b. An Integrated Method based on Multi-Models and Levels of Modeling for Design and Analysis of Complex Engineering Systems. In: Dig, D. (Ed.), Proceedings of the 17th ECOOP Doctoral Symposium and PhD Workshop.

Soares, M. S., Vrancken, J., 2007c. Requirements Specification and Modeling Through SysML. In: Proceedings of the 2007 IEEE International Conference on Systems, Man and Cybernetics.

Soares, M. S., Vrancken, J., 2007d. Road Traffic Signals Modeling and Analysis with Petri nets and Linear Logic. In: Proceedings of the 2007 IEEE International Conference on Networking, Sensing and Control (ICNSC 2007).

Soares, M. S., Vrancken, J., 2007e. Urban Transport XIII. Urban Transport and the Environment in the 21st Century. Wessex Institute of Technology, UK,

Ch. Scenario Analysis of a Network of Traffic Signals Designed with Petri nets.

Soares, M. S., Vrancken, J., 2008a. A Metamodeling Approach to Transform UML 2.0 Sequence Diagrams to Time Petri nets. In: Pahl, C. (Ed.), Proceedings of the IASTED International Conference on Software Engineering 2008.

Soares, M. S., Vrancken, J., 2008b. A Proposed Extension to the SysML Requirements Diagram. In: Pahl, C. (Ed.), Proceedings of the IASTED International Conference on Software Engineering 2008.

Soares, M. S., Vrancken, J., 2008c. Model-Driven User Requirements Specification using SysML. Journal of Software, 3 (6), 57–68.

Soares, M. S., Vrancken, J., 2008d. Responsive Traffic Signals Designed with Time Petri nets. In: Hing, G.-S. (Ed.), Proceedings of the IEEE International Conference on Systems, Man and Cybernetics (SMC).

Soares, M. S., Vrancken, J., 2009a. Evaluation of UML in Practice - Experiences in a Traffic Management Systems Company. In: Cordeiro, J., Filipe, J. (Eds.), Proceedings of the 11th International Conference on Enterprise Information Systems (ICEIS 2009).

Soares, M. S., Vrancken, J., 2009b. Including SysML in the 4+1 View Model of Architecture for Software-Intensive Systems. In: Kalawsky, R., O'Brien, J., Goonetilleke, T., Grocott, C. (Eds.), Proceedings of the 7th Annual Conference on Systems Engineering Research (CSER 2009).

Soares, M. S., Vrancken, J., Wang, Y., 2009a. A Common Architecture to Develop Road Traffic Management Systems. In: Proceedings of the 16th ITS World Congress.

Soares, M. S., Vrancken, J., Wang, Y., 2009b. Application of Publish-Subscribe Middleware for Road Traffic Measures Visualization. In: Chen, X., Kanda, T. (Eds.), Proceedings of the 2009 IEEE International Conference on Networking, Sensing and Control.

Soares, M. S., Vrancken, J., Wang, Y., 2009c. Software Product Line Architecture to Distributed Real-Time Systems. In: Sadiq, M. K., Sophatsathit, P., Xu, H. (Eds.), Proceedings of the 2009 International Conference on Software Engineering Theory and Practice (SETP 2009).

Soares, M. S., Vrancken, J., Wang, Y., 2010. Architecture-Based Development of Road Traffic Management Systems. In: Proceedings of the 2010 IEEE International Conference on Networking, Sensing and Control.

Sommerville, I., 2007. Software Engineering, 8th Edition. Addison Wesley, Essex, UK.

Soni, D., Nord, R. L., Hofmeister, C., 1995. Software Architecture in Industrial Applications. In: ICSE '95: Proceedings of the 17th International Conference on Software Engineering.

Srinivasan, D., Choy, M., 2006. Cooperative Multi-Agent System for Coordinated Traffic Signal Control. IEE Proceedings - Intelligent Transport Systems, 153 (1), 41–50.

Stoelhorst, H., Middelham, F., 2006. State of the Art in Regional Traffic Management Planning in the Netherlands. In: Proceedings of the Eleventh IFAC Symposium on Control in Transportation Systems.

Stough, R., 2001. Intelligent Transport Systems: Cases and Policies. Edward Elgar Publishing, Northampton, Massachusetts, USA.

Suzuki, I., Murata, T., 1983. A Method for Stepwise Refinement and Abstraction of Petri Nets. Journal of Computer and System Sciences, 27 (1), 51–76.

Taale, H., Westerman, M., Stoelhorst, H., van Amelsfort, D., 2004. Regional and Sustainable Traffic Management in The Netherlands: Methodology and Applications. In: Proceedings of the European Transport Conference.

Taguchi, K., Dong, J. S., Ciobanu, G., 2004. Relating pi-calculus to Object-Z. In: Proceedings of the 9th IEEE International Conference on Engineering Complex Computer Systems.

Tanenbaum, A. S., van Steen, M., 2006. Distributed Systems: Principles and Paradigms (2nd Edition). Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

The Standish Group, 2003. CHAOS Chronicles v3.0. Tech. rep., The Standish Group, last accessed on the 20th of August, 2009.

Tiako, P. F., 2008. Designing Software-Intensive Systems: Methods and Principles, 1st Edition. IGI Global, Hershey, New York, USA.

Tolba, C., Thomas, P., ElMoudni, A., Lefebvre, D., 2003. Performances Evaluation of the Traffic Control in a Single Crossroad by Petri Nets. In: Proceedings of the ETFA - Emerging Technologies and Factory Automation. Vol. 2.

Tsichritzis, D., 1997. The Dynamics of Innovation. Copernicus, New York, NY, US.

USDOT, October 1997. ITS Benefits: Continuing Successes and Operational Test Results. Tech. rep., U.S. Department of Transportation - Federal Highway Administration, Washington, DC.

van der Linden, F. J., Schmid, K., Rommes, E., 2007. Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering. Springer-Verlag, Secaucus, NJ, USA.

van Genuchten, M., 1991. Why is Software Late? An Empirical Study of Reasons For Delay in Software Development. IEEE Transactions on Software Engineering, 17 (6), 582–590.

van Nes, N., Brandenburg, S., Twisk, D., 2008. Dynamic Speed Limits; Effects on Homogeneity of Driving Speed. In: Proceedings of the IEEE Intelligent Vehicles Symposium.

van Ommering, R. C., Bosch, J., 2002. Widening the Scope of Software Product Lines - From Variation to Composition. In: SPLC 2: Proceedings of the Second International Conference on Software Product Lines.

van Vliet, H., 2008. Software Engineering: Principles and Practice. John Wiley & Sons, New York, NY, USA.

Venkatesh, V., Bala, H., 2008. Technology Acceptance Model 3 and a Research Agenda on Interventions. Decision Sciences, 39 (2), 273–315.

Venkatesh, V., Davis, F. D., 2000. A Theoretical Extension of the Technology Acceptance Model: Four Longitudinal Field Studies. Management Science, 46 (2), 186–204.

Venkatesh, V., Morris, M. G., Davis, G. B., Davis, F. D., 2003. User Acceptance of Information Technology: Toward a Unified View. MIS Quarterly, 27 (3), 425–478.

VERTIS, 1999. System Architecture for ITS in Japan. Tech. rep., last accessed on the 21th of August, 2009.

Vogler, W., 1992. Modular Construction and Partial Order Semantics of Petri Nets. Springer-Verlag, Secaucus, NJ, USA.

Vrancken, J., 2006. Layered Models in IT Standardization. In: IEEE International Conference on Systems, Man and Cybernetics. Vol. 5.

Vrancken, J., Avontuur, V., Westerman, M., Blonk, J., 1998. Architecture Development for Traffic Control on the Dutch Motorways. In: Proceedings of the ITS World Congress.

Vrancken, J., Soares, M. S., 2009a. A Hierarchical Network Model for Road Traffic Control. In: Chen, X., Kanda, T. (Eds.), Proceedings of the 2009 IEEE International Conference on Networking, Sensing and Control.

Vrancken, J., Soares, M. S., 2009b. A Real-life Test Bed for Multi-agent Monitoring of Road Network Performance. International Journal of Critical Infrastructures, 4 (5), 357–367.

Vrancken, J., Soares, M. S., 2010. Intelligent Infrastructures. Springer, The Netherlands, Ch. Intelligent Road Network Control.

Vrancken, J., van den Berg, J., Soares, M. S., 2008. Human factors in System Reliability: Lessons Learnt from the Maeslant Storm Surge Barrier in The Netherlands. International Journal of Critical Infrastructures, 4 (4), 418–429.

Wang, J., 1998. Timed Petri Nets, Theory and Application. Kluwer Academic Publishers, Norwell, MA, USA.

Wang, Y., 2007. Software Engineering Foundations: A Software Science Perspective, 1st Edition. AUERBACH / CRC Press, New York, NY, USA.

Wassyng, A., Lawford, M., 2003. Lessons Learned from a Successful Implementation of Formal Methods in an Industrial Project. In: Araki, K., Gnesi, S., Mandrioli, D. (Eds.), FME 2003: International Symposium of Formal Methods Europe Proceedings.

Wing, J. M., 1990. A Specifier's Introduction to Formal Methods. Computer, 23 (9), 8–23.

Wirsing, M., Banâtre, J.-P., Hölzl, M. M., Rauschmayer, A. (Eds.), 2008. Software-Intensive Systems and New Computing Paradigms - Challenges and Visions. Vol. 5380 of Lecture Notes in Computer Science. Springer.

Wirsing, M., Holzl, M., 2006. Software Intensive Systems. Tech. rep., ERCIM EEIG.

Wirth, N., 2008. A Brief History of Software Engineering. IEEE Annals of the History of Computing, 30 (3), 32–39.

Woodcock, J., Larsen, P. G., Bicarregui, J., Fitzgerald, J., 2009. Formal Methods: Practice and Experience . ACM Computing Surveys, 41 (4), 1–36.

Wooldridge, M., 2009. Introduction to Multiagent Systems. John Wiley & Sons, Inc., New York, NY, USA.

Yin, R. K., 2003. Case Study Research. Design and Methods, 3rd Edition. Vol. 5 of Applied Social Research Method Series. Sage Publications, California, USA.

Zeigler, B. P., Praehofer, H., Kim, T. G., 2000. Theory of Modeling and Simulation, 2nd Edition. Academic Press, San Diego, California, USA.

Zeng, R., Li, G., Lin, L., 2007. Adaptive Traffic Signals Control by Using Fuzzy Logic. In: ICICIC '07: Proceedings of the Second International Conference on Innovative Computing, Information and Control.

Zhou, M. C., Dicesare, F., Rudolph, D., 1992. Design and Implementation of a Petri Net Supervisory for a Flexible Manufacturing System. IEEE Transactions on Robotics Automation, 28 (6), 1199–1208.

Appendix A

Visualization of Junction Measurements

A.1 List of requirements

1. For each direction, the user wants to visualize information about splitfactor and waiting time.
2. The number of directions depends on the number of Accessor links.
3. The symbol "-" should be used in the place of numerical values when there is no direction.
4. Junction must have an automatic created image based on Links information
5. Junction image must be by default on the geographical map (Trinivision).
6. Junction image on the geographical map (Trinivision) must be shown in a small form
7. Clicking in the junction image will result in an overview from this image containing the junction information
8. The junction image must have an understandable and representable layout
9. The information shown shall be from all Accessor links into the junction
10. The information shall be shown for each traffic stream into the junction
11. Accessor link information should contain the splitfactor
12. Information about intensity and waiting time must be shown when there is a Direction with an actuator or sensor from the Accessor link.

A.2 Algorithm specifications

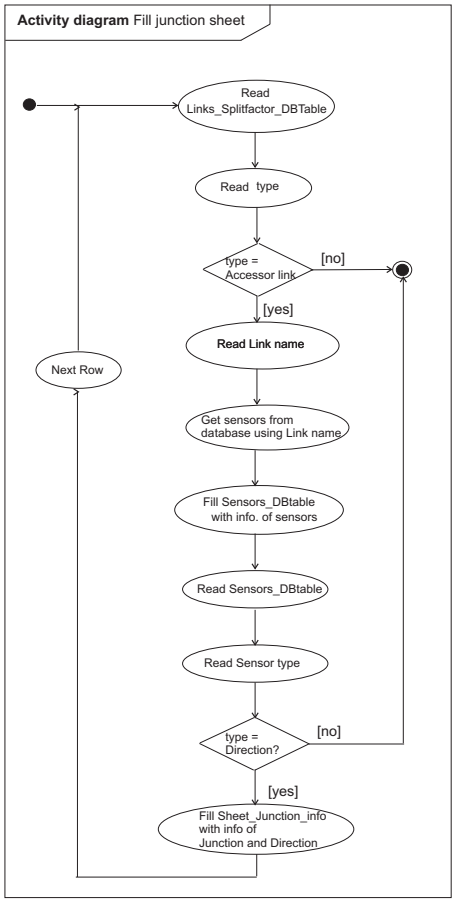

Figure A.1: Activity diagram: Fill Junction Sheet

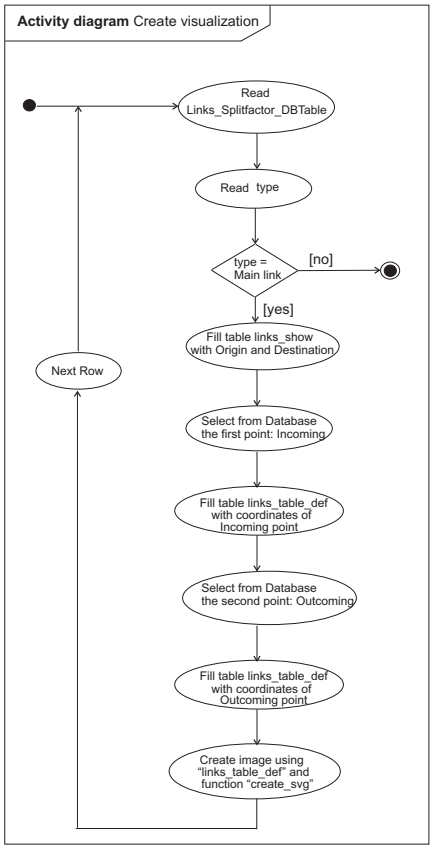

Figure A.2: Activity diagram: Create Visualization

Appendix B

List of User Requirements for RTMS

Road User

1. RU1 - The losses due to inefficient usage of the road network in the Netherlands must be minimized. In addition, the traveling time must be predictable.

Ministry

1. M2 - The utilization of the road network in the Netherlands must be maximized (optimally utilized).
2. M3 - The initial and yearly investment costs of the management of the traffic-flow in the Netherlands must be minimized.

Traffic Manager

1. TM4 - It is expected that software systems will be increasingly more intelligent for managing the traffic-flow in a more effective and efficient manner.
2. TM5 - To optimize traffic flow, it is expected that gradually, region-wide traffic management methods will be introduced.
3. TM6 - The traffic management systems must have a convenient access to region-wide, nation-wide, or even European-wide parameters so that the traffic-flow can be managed optimally.

4. TM7 - It must be possible for the traffic managers/experts to express (strategic) "task and scenario management frames", conveniently.

5. TM8 - The system should effectively gather and interpret all kinds of information for the purpose of conveniently assessing the performance of the responsible companies/organizations that have carried out the construction of the related traffic systems and/or infrastructure.

6. TM9 - The system must support the traffic managers/experts so that they can express various experimental simulations and analytical models.

7. TM10 - The system must enable the traffic managers/experts to access various kinds of statistical data.

8. TM11 - The system must enable the traffic managers/experts to access different kinds of data for transient cases such as incidents.

9. TM12 - The system must provide means for expressing a wide range of tasks and scenarios.

10. TM13 - The traffic management will gradually evolve from object management towards task and scenario management.

Traffic Manager Center

1. TMC14 - The operational costs of the traffic management centers and shared resources must be minimized.

2. TMC15 - The operators' reaction speed must be improved, especially in critical and unanticipated situations.

3. TMC16 - The operators' decision accuracy must be improved, especially in critical and unanticipated situations.

4. TMC17 - The system must provide means to manage various "traffic management configuration information" conveniently.

5. TMC18 - The system must provide tools so that the operators can perform their work more efficiently.

6. TMC19 - The system must provide tools so that the operators can perform their work more effectively.

7. TMC20 - The system must make it intuitively obvious in which function/-context the operator is working in.

8. TMC21 - The education material and process necessary to train the operators must be simplified, standardized and supported. This should improve the effectiveness of tutoring.

The task, scenario and operator manager

1. OM22 - The system must provide convenient means to manage the task and scenario allocation processes.
2. OM23 - The system must provide means to optimally allocate tasks and scenarios to the operators.
3. OM24 - The task and scenario definition and allocation strategies must be conveniently expressible to deal with the density, urgency and the range of the management activities.
4. OM25 - The system must provide convenient means to distribute the tasks and scenarios according to the dynamic context of the traffic management, for example, to effectively cope with the incidents.
5. OM26 The system must be able to express various task and scenario allocation strategies, for example based on hierarchical and/or team-based organizational structures.
6. OM27 - The system must provide means to conveniently transfer an operators task and scenario to another.
7. OM28 - The system must provide means to conveniently separate operators tasks and scenarios from each other.
8. OM29 - The system must provide means to flexibly allocate tasks and scenarios to the operators.
9. OM30 - The system must provide expressive and convenient means to authorize operators.
10. OM31 - The system must conveniently enable exchanging personnel among different traffic management centers.
11. OM32 - The system must conveniently enable migrating an operator to a different traffic management center.
12. OM33 - The system must provide expressive and convenient means for introducing different kinds of operators.
13. OM34 - The system must provide expressible and flexible means to authorize tasks and scenarios.

14. OM35 - The system must provide expressible and convenient means for defining various different kinds of concurrent task and scenario execution and synchronization.

15. OM36 - The system must not allow introducing unsafe and/or incorrect task and scenario definitions. In fact, the operator's actions must never cause unsafe actions at the roadside.

16. OM37 - The system must support graceful degradation of tasks and scenarios, where applicable.

17. OM38 - The system must be able to express different dynamic tasks and scenario prioritization schemes.

18. OM39 - The system must provide expressive and flexible means to steer tasks and scenarios in different modalities, such as management by awareness, management by exception and management by anticipation.

19. OM40 - The tasks and scenarios that are defined at a higher-level of management must be consistent with the tasks and scenarios that are supported/allowed by the roadside systems.

20. OM41 - The system must provide expressive and convenient means for introducing tasks and scenarios that utilize multiple and different (roadside) systems.

21. OM42 - The system must provide expressive and convenient means to define various kinds of tasks and scenarios such as observing, tracing, preparing, enabling time critical and non time critical activities, and various supporting activities such as report generation.

22. OM43 - The system must be able to support unanticipated tasks and scenarios (evolvability).

Operator

1. O44 - The operators working environment must be flexible enough so that he/she can conveniently adapt to changing tasks and scenarios.

2. O45 - The system must be more usable than the current systems deployed at the traffic centers.

3. O46 - The operators must be able to comprehend the situation effectively despite the use of different kinds of data.

4. O47 - The system must provide a uniform and consistent user-interface.

The Designer of Operators' Supporting Functions

1. DOSF48 - The system must be expressive and flexible enough to present all the necessary operations (including control parameters) to the operator.
2. DOSF49 - The system must be expressive and flexible enough to present all the necessary information to the operator.
3. DOSF50 - The system must provide expressive and comprehensible high-level constructs for abstracting the frequently used operations, where applicable.
4. DOSF51 - The system must provide consistent, convenient and expressive means for combining various information sources and types obtained from various different systems.
5. DOSF52 - The system must provide expressive and convenient means to filter out the unnecessary data according to the context of the task and scenario, and the operators interest and capability. Depending on the needs, filtering must be realized statically and dynamically.
6. DOSF53 - To ease the operators task and scenario, the system must provide expressive and convenient means for eliminating the unnecessary control functions, where applicable.
7. DOSF54 - The system must conveniently support applications that provide means for directing and managing the traffic flow.
8. DOSF55 - The system must conveniently support applications that provide means for controlling and protecting objects (tunnels, bridges, etc.).
9. DOSF56 - The system must conveniently support applications that provide means for controlling and protecting objects (tunnels, bridges, etc.) as a subtask of overall traffic management.
10. DOSF57 - The system must conveniently support applications that provide means for coordinating the necessary actions in case of incidents so that the traffic flow gets normalized in the fastest possible way.
11. DOSF58 - The system must conveniently support applications that provide means for coordinating road construction and/or repair activities at planning and/or operational phases.
12. DOSF59 - The system must conveniently support applications that provide means for displaying the relevant information to any user (drivers, traffic centers, police, etc.) of the road network.

13. DOSF60 - The system must be flexible and expressible enough to cope with different kinds of information provided by various sources such as the roadside systems, external geographic databases, meteo, teletext and radio dispatching.

14. DOSF61 - The system must support introducing unanticipated functions and data types (evolvability).

15. DOSF62 - The system must effectively and conveniently monitor and protect the availability of all the dynamic traffic management systems.

The Technical Quality Manager

1. TQM63 - Reduced costs: The effort that is necessary for designing and implementing, configuring and instantiating the system at different traffic management centers must be minimized.

2. TQM64 - Manageability: There must be an effective and convenient support for managing the technical applications in an integrated way, so that it must be conveniently possible to incorporate, migrate and validate different versions and configurations.

3. TQM65 - Configurability (or composability): There must be an effective and convenient means for system configuration.

4. TQM66 - Effective data management: There must be an effective and convenient support for the management of technical data for system configuration.

5. TQM67 - Reduced (managed) complexity: The current systems are very complex. This is because many system components have complex data and control dependencies with each other. The complexity of the systems must be reduced effectively without loosing their desired capabilities.

6. TQM68 - Improved relevance and correctness: The system must provide all the technical means to support the required functions and/or quality factors defined in this section

7. TQM69 - Information relevance: The information gathered by the (roadside) systems must be relevant for the traffic management centers and high-level traffic management needs.

8. TQM70 - Information availability: The information gathered by the (roadside) systems must be made available at the right place.

9. TQM71 - Information timing: The information gathered by the (roadside) systems must be made available at the right time.

10. TQM72 - Reduced redundancy: The data and functions in the system must not be unnecessarily replicated.

11. TQM73 - Heterogeneity: The system must be able to cope with diverse IT (software, hardware and/or communication) systems.

12. TQM74 - Well-defined architecture: To manage the costs, to organize and direct the activities, and to increase their effectiveness, a system architecture concept must be defined.

13. TQM75 - Improved separation and composition of concerns: The concerns that are necessary to implement the system must be effectively separated and composed with each other.

14. TQM76 - Relevance: To define a correct architecture, it is necessary to identify and specify the fundamental concerns (or modules) that are necessary to implement the system.

15. TQM77 - Availability: The system must provide a high degree of availability especially in the implementation of the basic requirements.

16. TQM78 - Continuous operation: The services, components and subsystems that implement the basic requirements must be continuously operational. All kinds of required changes must be carried out at run time without disrupting the operation of the system.

17. TQM79 - Acceptable time performance: The system must perform within the required timing constraints.

Appendix C

Exercises Proposed for the Course at Trinité

C.1 Exercise 1 - Use Case diagram

Given the following list of requirements, design a Use Case diagram.

1. The system shall manage all information about employees and guests.
2. Before hiring an employee, information about his/her criminal record must be searched at the Police Office.
3. Employees shall be classified as operational and managers.
4. Only managers shall operate the system.
5. A guest shall make a reservation with minimum of 4 hours and a maximum of 45 days in advance.
6. A guest shall choose to pay using a credit card or in cash to the manager.
7. Guests that stay in the hotel for more than 12 nights during the period of one year are classified as special and must have 3% of discount during the next year.
8. Guests that stay in the hotel for more than 24 nights during the period of one year are classified as very special and must have 7% of discounts during the next year.
9. For payments with credit card, the client profile and card must be checked with the Credit Card Company.

10. Payment by credit card must be secure.
11. All guests shall be registered with a minimum information about nationality, name, address and contact details.
12. The system shall accept check-ins starting only from 11am.
13. The system shall accept check-outs continuously.
14. The guests can optionally fill an electronic form about the hotel and services evaluation when checking-out.

C.2 Exercise 2 - Sequence diagram

Design a Sequence Diagram for the use case scenario borrow a book, according to the following steps:

Client	**Librarian**	**System**
1. Gives library card		
	2. Enters library card	
		3. Check client
		4. Client is OK
5. Gives book to librarian		
	6. Enters book code	
		7. Check if book is available
		8. Book is available
		9. Borrow book

C.3 Exercise 3 - SysML Requirements diagram

Design a SysML Requirements diagram for the following user requirements.

Traffic Manager Center

1. TMC14 - The operational costs of the traffic management centers and shared resources must be minimized.
2. TMC15 - The operators' reaction speed must be improved, especially in critical and unanticipated situations.
3. TMC16 - The operators' decision accuracy must be improved, especially in critical and unanticipated situations.

4. TMC17 - The system must provide means to manage various "traffic management configuration information" conveniently.
5. TMC18 - The system must provide tools so that the operators can perform their work more efficiently.
6. TMC19 - The system must provide tools so that the operators can perform their work more effectively.
7. TMC20 - The system must make it intuitively obvious in which function/-context the operator is working in.
8. TMC21 - The education material and process necessary to train the operators must be simplified, standardized and supported. This should improve the effectiveness of tutoring.

C.4 Exercise 4 - Petri nets

Model the states and the states transition of a traffic signal using Petri nets according to the following requirements:

1. Green time: minimum 30, maximum 40 seconds.
2. Yellow: minimum and maximum: 5 seconds.
3. From red to green to yellow to red.

C.5 Exercise 5 - Petri nets

Model the system using Petri nets according to the following requirements:

1. Pieces of raw material stored in a tank T1 are processed at machine M1 when the machine is available.
2. At M1, the process consists of splitting raw pieces into two parts.
3. One part goes to machine M2, where it is processed during a period of 4 to 6 minutes.
4. Another part of the pieces are processed at a Machine M3, during a period of 6 to 8 minutes.
5. After the processing, each part of the original piece is stored in a separate tank T2 or T3.

6. A robot R1 transports the pieces from M1 to M2 or to M3.
7. The robot can only transport one piece each time.
8. Machines M1, M2 and M3 can be used for other purposes when they are available.
9. Processing at machines M2 and M3 are independent of each other.

Glossary

List of abbreviations

The following abbreviations are used in this thesis:

ADL	Architecture Description Languages
AR	Action Research
ATC	Architecture for Traffic Control
BPMN	Business Process Modeling Notation
DES	Discrete Event Systems
DTCA	Distributed Traffic Control Architecture
ETC	Electronic Toll Collection
EU	European Union
FSM	Finite State Machine
GDP	Gross Domestic Product
HARS	*Het Alkmaar Regelsysteem*
INCOSE	International Council on Systems Engineering
IT	Information Technologies
ITS	Intelligent Transportation Systems
KA	Knowledge Area
KAREN	Keystone Architecture Required for European Networks
MARTE	Modeling and Analysis of Real-time and Embedded Systems
MOF	Meta Object Facility
NATO	North Atlantic Treaty Organization
NL	Natural language
ODMGR	Origin-Destination Managers
OMT	Object Modeling Technique
OMG	Object Management Group
OOSE	Object-Oriented Software Engineering
PMBOK	Project Management Body of Knowledge
PLC	Programmable Logic Controller
PN	Petri Net
PNC	Petri Net Component
RTC	Road Traffic Control

RTMS	Road Traffic Management Systems
RUP	Rational Unified Process
SRS	Software Requirements Specification
SNL	Structured Natural language
SysML	Systems Modeling Language
SWEBOK	Software Engineering Body of Knowledge
TAM	Technology Acceptance Model
TD	Time Domain
TMC	Traffic Management Center
TTM	Technology Transition Model
UML	Unified Modeling Language
VMS	Variable Message Signs
VSL	Variable Speed Limits
XP	eXtreme Programming
...	

Summary

Architecture-Driven Integration of Modeling Languages for the Design of Software-Intensive Systems

Software-intensive systems are large, complex systems in which software is an essential component, that interacts with other software, systems, devices, actuators, sensors and with people. Being an essential component, software influences the design, construction, deployment, and evolution of the system as a whole. These systems are in widespread use and their impact on society is still increasing. Examples of software-intensive systems can be found in many sectors, such as manufacturing plants, transportation, military, telecommunication and health care.

A great challenge in modern society is to develop successful software-intensive systems respecting constraints such as costs and deadlines, and to be able to maintain and evolve these systems. This challenge is associated with another important one: developing practically useful and theoretically well-founded principles, methods, algorithms and tools for programming and engineering reliable, secure, cost-effective, and efficient software-intensive systems throughout their whole life-cycle. The proper environment in which software-intensive systems act poses great challenges. Software-intensive systems are frequently used to control critical infrastructures in which any error, non-conformance or even response delays may cause enormous financial damage and jeopardize human life.

An interesting fact that can not be overlooked is that software has to evolve in order to remain useful. In fact, software is hardly developed from scratch. Green field projects, in which software systems are developed without any constraints imposed by prior work, are rare. Thus, software must be flexible in order to facilitate its change.

More specifically, the type of software-intensive systems that are investigated in this thesis are the Distributed Real-Time Systems. The term Real-Time System usually refers to systems with timing constraints. In

concurrent problems, there is no way of predicting which system component will provide the next input, which increases design complexity. Moreover, system components, such as sensors and actuators, are often geographically distributed in a network and need to communicate according to specific timing constraints described in system requirements documents.

The case studies provided in the thesis are related to Road Traffic Management Systems (RTMS). RTMS are Distributed Real-Time Systems that influence traffic by using a variety of actuators, such as traffic signals and Variable Message Signs, based on acquired data using various types of sensors, such as video cameras and inductive loops. All the difficulties of development and maintenance of software-intensive systems hold for RTMS. For example, after deployment these systems are used for many years, which means that they must be maintained in order to cope with hardware and policy changes. This makes it very unlikely that a new RTMS project will start from scratch. In most cases, legacy systems have been in operation for many years and must be taken into account. For instance, all cities have deployed urban traffic control systems with a varying degree of sophistication, control algorithms and hardware.

Design for software-intensive systems requires adequate methodology in order to support the development of these systems. However, the theory of modeling for software-intensive systems remains incomplete, and methodologies for specifying and verifying software-intensive systems pose a grand challenge that a broad stream of research must address. In order to develop software-intensive systems, modeling tasks have to cover different development phases such as requirements analysis, architectural design, and detailed design. Other phases, such as implementation, testing and integration are direct consequences of the modeling phases. The focus of this thesis is on the modeling phases of software development.

This thesis contributes to Software Engineering research and practice by proposing the extension and integration of formal and semi-formal modeling languages in a multiple-view software architecture, combined with domain architecture, which are used in practice to develop a family of distributed real-time systems in the road traffic domain.

Research Objective

In the research that led to this thesis a multi-disciplinary approach, combining Traffic Engineering and Software Engineering, was used. Traffic engineers come up with new control strategies and algorithms for improving traffic. Once new solutions are defined from a Traffic Engineering

point of view, there is the problem of obtaining operational systems that address all requirements. Knowing what to build is just the first step that must be followed by the how to build. Both are problematic and they depend on each other.

The research objective can be stated as follows:

Improve Systems and Software Engineering methodology (what), by using, adapting, extending and combining modeling languages (how), to be used by systems and software developers (to whom), for designing Distributed Real-Time Systems (for which purpose) that are in line with requirements, flexible, and reliable (with which quality factors).

Summary of Chapters

Chapter 1 introduces the problem. The importance of software for modern society and the difficulties of software development and maintenance are discussed in this chapter. HARS (*Het Alkmaar Regelsysteem*) is introduced to illustrate the type of software systems that are the target of this thesis.

Chapters 2 and 3 are both about theoretical background, written after extensive literature research on, respectively, Software-Intensive Systems and Intelligent Transportation Systems. In these Chapters the scope of the thesis is delimited.

The early introduction of graphical models in Requirements Engineering through a common modeling language (SysML) is proposed in Chapter 4. Although the UML Use Case diagram has been used to model functional requirements, the SysML Requirements diagram closes a gap by modeling other types of requirements, such as non-functional ones. A good knowledge of requirements is essential for designing the system architecture. Two sub-areas of the Requirements Engineering process are investigated in this chapter: the Specification and the Analysis. Specification is improved by identifying, classifying, and relating requirements to each other. The graphical modeling of each requirement helps in improving the requirements documentation. This is performed using the SysML Requirements diagram and the SysML Tables. As a result, the introduction of SysML as a language for Requirements Engineering provides a bridge between the text-based requirements and the design models of the system. The basic SysML Requirements diagram was extended with new properties. These properties are optional, but are useful in activities related to requirements analysis and project management, such as release planning and risk evaluation and mitigation. Requirements analysis is improved by identifying

relationships between requirements, the type of each relationship, and by tracing requirements through development. Requirements are related to each other and also directly related to design.

Two types of system architecture were proposed in this thesis: the domain architecture and the software architecture. Both types are relevant to software-intensive systems. Due to the complexity of software-intensive systems, both types are necessary and complementary. The domain architecture was introduced in Chapter 5. It expresses the organizational procedures, information and business structure of the system for customers. It can be used as an input for the software architecture, and for communicating and making explicit decisions related to business. Typically the domain architecture proposes a family of systems. This is necessary in order to avoid stovepipe systems, as the target family of systems should share many assets. However, the domain architecture is too high-level to be used as a basis for software design.

The software architecture (Chapter 6) is useful to identify subsystems, components and their interfaces. In large systems, with multi-disciplinary teams involved in the development phases, important decisions must be documented to be referred to in the future. Hence, the design of the software architecture is an important activity in order to facilitate software maintenance. Another important characteristic of the proposed software architecture is that it is the basis for the design of a family of products belonging to the same domain. The advantages are well-known. The same architecture can be the basis for a family of products, which increases the possibility of reusing artifacts that can vary from source code to complete components and subsystems. Thus, commonalities can be easily identified and reuse increased, leading to economic and quality benefits. Additionally, maintainability is enabled by facilitating the creation of multiple systems versions.

Chapter 7 is about detailed design and verification of system components and software objects using formal methods. In this chapter, models of urban traffic control signals were created with Petri nets and simulated in a computer-based tool, providing validations of specific scenarios. Although these simulations do not guarantee that models are error-free, they can already detect many design flaws. Thus, when the behavior of the chosen scenarios is as intended, the same models can be formally verified using one or more of the variety of verification tools provided by the Petri net theory. The advantage is that the same models can be used in various activities during system design. In this thesis, analysis of Petri net models is done by using invariant analysis, the reachability graph, and theorem proving

with Linear Logic.

Chapter 8 is about the evaluation of the suitability of including and combining semi-formal modeling languages (UML and SysML) and a formal modeling language (Petri nets) in the 4+1 View Model of Architecture, as well as the applicability of this approach to design Distributed Real-Time Systems. The approach for the evaluation includes the application in a real environment, in what is known as "industry as laboratory". The results of this evaluation are presented at the end of this chapter. These results are used in Chapter 9 in which conclusions are presented together with limitations, recommendations and proposals for future work.

Samenvatting

Architectuur-gedreven integratie van modelleertalen voor het ontwerp van programmatuur-intensieve systemen

Programmatuur-intensieve systemen zijn grote, complexe systemen waarvan programmatuur een essentieel onderdeel vormt dat interageert met andere programmatuur, systemen, apparaten, actuatoren, sensoren en met mensen. Als essentieel onderdeel beïnvloedt programmatuur het ontwerp, de constructie, de uitrol en de evolutie van het systeem waar het onderdeel van uitmaakt. Dergelijke systemen worden op grote schaal toegepast en hun invloed op de maatschappij is nog steeds groeiende. Voorbeelden van programmatuur-intensieve systemen zijn te vinden in vele sectoren zoals de industrie, het transport, het leger, de telecommunicatie en de gezondheidszorg.

Het is een grote uitdaging in de huidige maatschappij om succesvolle, programmatuur-intensieve systemen te ontwikkelen, binnen gestelde grenzen van, onder meer, tijd en budget, en die onderhouden en verder ontwikkeld kunnen worden. Deze uitdaging hangt samen met een andere belangrijke uitdaging: het ontwikkelen van praktisch bruikbare, en theoretisch goed gefundeerde principes, methoden, algoritmen en gereedschap voor het ontwikkelen en bouwen van betrouwbare, veilige, kosten-effectieve en efficiënte programmatuur-intensieve systemen, en dit gedurende hun gehele levenscyclus. De omgeving waarin programmatuur-intensieve systemen toegepast worden, stelt grote uitdagingen. Deze systemen worden regelmatig toegepast voor de besturing van kritische infrastructuren waarin elke fout of afwijking of zelfs vertraging in reactietijd, grote financiële schade kan veroorzaken en mensenlevens in gevaar kan brengen.

Een belangrijk feit, dat niet over het hoofd gezien kan worden, is dat programmatuur moet evolueren om nuttig te blijven. Het is zelfs zo dat pro-

grammatuur nauwelijks nog van de grond af ontwikkeld wordt. Projecten waarin programmatuur ontwikkeld wordt zonder enige beperking vanuit eerder ontwikkelwerk zijn zeldzaam. Derhalve moet programmatuur flexibel zijn ten einde dergelijke aanpassingen mogelijk te maken.

De programmatuur-intensieve systemen die hier onderzocht zijn, zijn met name de gedistribueerde, tijd-kritische systemen. De term tijd-kritisch verwijst meestal naar systemen die aan tijdigheidseisen moeten voldoen. In situaties met verschillende gelijktijdig werkende systeemonderdelen is er geen manier om te voorspellen welke onderdeel de volgende input gaat leveren, waardoor de ontwerpcomplexiteit toeneemt. Bovendien zijn systeemonderdelen, zoals sensoren en actuatoren, vaak geografisch gespreid in een netwerk en moeten deze communiceren binnen strikte tijdseisen, beschreven in het programma van eisen.

De praktijkvoorbeelden die worden beschreven in dit proefschrift hebben betrekking op systemen voor dynamisch verkeersmanagement (DVM-systemen). DVM-systemen zijn gedistribueerde, tijd-kritische systemen die het verkeer beïnvloeden door een keur aan actuatoren, zoals verkeerslichten en matrix-signaalgevers, op basis van data, ingewonnen met behulp van diverse typen sensoren, zoals video-camera's en inductielussen in de weg.

Alle bekende problemen van ontwikkeling en onderhoud van programmatuur-intsieve systemen zijn van toepassing op DVM-systemen. Bijvoorbeeld, na ontwikkeling worden deze systemen vele jaren gebruikt, wat betekent dat ze onderhouden moeten worden teneinde het hoofd te bieden aan allerlei omgevingsveranderingen zoals andere hardware en beleidswijzigingen. Dit maakt het heel onwaarschijnlijk dat een nieuw DVM-systeem van de grond af kan beginnen. In de meeste gevallen is er sprake van een erfenis aan systemen die al jaren in gebruik zijn en waar rekening mee gehouden moet worden. Bijvoorbeeld, alle steden hebben verkeerslichten geïnstalleerd, maar met grote verschillen in geavanceerdheid, besturingsalgoritmen en hardware.

Het ontwerp van programmatuur-intensieve systemen vereist adequate methoden om de ontwikkeling van deze systemen te ondersteunen. Maar, de theorie van het modelleren van programmatuur-intensieve systemen is nog steeds incompleet en praktisch bruikbare methoden voor specificatie en verificatie van programmatuur-intensieve systemen vormen een grote uitdaging waarvoor onderzoek wordt vereist met een brede aanpak. Voor de ontwikkeling van programmatuur-intensieve systemen dienen de modelleertaken zich over verschillende ontwikkelingsfasen uit te strekken,

zoals analyse van eisen, architecturaal ontwerp en detailontwerp. Andere fasen, zoals implementatie, testen en integratie, zijn directe afgeleiden van deze modelleerfasen. Dit proefschrift gaat vooral over de modelleerfasen.

Dit proefschrift draagt bij aan het onderzoek in en de praktische toepassing van Programmatuurkunde door het uitbreiden en integreren van formele en semi-formele modelleertalen, ingebed in een software-architectuur met verschillende gezichtspunten. In combinatie met een domein-architectuur, wordt deze in de praktijk gebruikt voor de ontwikkeling van families van gedistribueerde, tijd-kritische systemen voor wegverkeersmanagement.

Het onderzoeksdoel

In het onderzoek voor dit proefschrift is een multi-disciplinaire aanpak gehanteerd door het combineren van Verkeerskunde en Programmatuurkunde. Verkeerskundigen bedenken nieuwe besturingsconcepten en algoritmen ter verbetering van het verkeer. Zodra nieuwe oplossingen zijn bedacht in verkeerskundige zin, volgt het probleem om praktisch inzetbare systemen te verkrijgen die aan alle eisen voldoen. Weten wat je wilt bouwen is alleen de eerste stap, die onvermijdelijk gevolgd wordt door de vraag hoe te bouwen. Beide vragen zijn lastig en ze zijn onderling afhankelijk.

Het onderzoeksdoel kan als volgt geformuleerd worden:

Verbeter systemen en programmatuurkundige methoden, door aanpassing, uitbreiding en combinatie van modelleertalen (het *hoe*), voor gebruik door systeem- en programmatuurontwikkelaars (door *wie*), voor het ontwerpen van gedistribueerde, tijd-kritische systemen (voor *welk doel*) die voldoen aan de eisen en die flexibel en betrouwbaar zijn (met *welke kwaliteitsaspecten*).

Samenvatting van de hoofdstukken

Hoofdstuk 1 geeft een inleiding tot het probleem, het belang van programmatuur voor de moderne maatschappij en de moeilijkheden van programmatuurontwikkeling en -onderhoud. HARS (Het Alkmaar Regelsysteem) wordt geïntroduceerd teneinde het type programmatuur te illustreren dat onderwerp vormt van dit proefschrift.

Hoofdstukken 2 en 3 geven beide de theoretische achtergrond, op basis van uitgebreid literatuuronderzoek, van respectievelijk programmatuurintensieve systemen en intelligente transportsystemen. In deze hoofdstukken wordt ook de reikwijdte van dit proefschrift nader afgebakend.

Hoofdstuk 4 pleit voor het vroegtijdig introduceren van grafische modellen bij het opstellen van het programma van eisen, uitgedrukt in een gemeenschappelijke modelleertaal (SysML). Alhoewel het UML Use Case-diagram gebruikt wordt voor het beschrijven van functionele eisen, voorziet het SysML Requirements diagram in een leemte doordat daarmee andersoortige eisen beschreven kunnen worden, zoals niet-functionele. Een goede kennis van de eisen is noodzakelijk om de systeemarchitectuur te ontwerpen. Twee deelaktiviteiten binnen het opstellen van eisen worden onderzocht in dit hoofdstuk: specificatie en analyse. Specificatie wordt verbeterd door het identificeren, classificeren en het aan elkaar relateren van eisen. Het grafisch modelleren van elke eis helpt bij het verbeteren van het documenteren van eisen. Dit gebeurt aan de hand van het SysML Requirements diagram en de SysML Tabellen. Hierdoor vormt de invoering van SysML, als taal voor het opstellen van eisen, een brug tussen tekstuele eisen en de ontwerpmodellen van een systeem. Het basis Requirements Diagram van SysML is uitgebreid met nieuwe eigenschappen. Deze eigenschappen zijn optioneel, maar zijn nuttig bij activiteiten die gerelateerd zijn aan analyse van eisen en projectmanagement, zoals uitrolplanning en risico-evaluatie en -vermindering. De analyse van eisen wordt verbeterd door het identificeren van relaties tussen eisen, het typeren van deze relaties, en door het volgen van eisen door het hele ontwikkelproces. Er worden relaties gelegd tussen eisen, en tussen eisen en het ontwerp.

Er worden twee vormen van systeemarchitectuur behandeld in dit proefschrift: de domeinarchitectuur en de software-architectuur. Beide typen zijn van belang voor programmatuur-intensieve systemen. Door de complexiteit van deze systemen zijn ze zelfs noodzakelijk en ze zijn complementair. De domeinarchitectuur is behandeld in hoofdstuk 5. Deze beschrijft de organisatorische procedures en de informatie- en business-structuur voor de afnemers van het systeem. Deze architectuur kan input leveren voor de software-architectuur, en voor het communiceren over en het expliciet maken van beslissingen binnen het business-domein. Meestal gaat de domeinarchitectuur over een familie van systemen. Dit is noodzakelijk om monolithische, kachel-pijpsystemen te voorkomen, want de beoogde familie van systemen hoort een groot aantal componenten gemeenschappelijk te gebruiken. Maar de domeinarchitectuur is te abstract om direct te gebruiken als basis voor programmatuurontwerp.

De software-architectuur (hoofdstuk 6) is nuttig om subsystemen en componenten en hun interfaces te bepalen. Bij grote systemen, met multidisciplinaire ontwikkelteams, moeten belangrijke beslissingen gedocumenteerd worden teneinde er later naar te kunnen verwijzen. Vandaar dat

het ontwerp van de software-architectuur een belangrijke activiteit is om onderhoud van het systeem mogelijk te maken. Een tweede belangrijke eigenschap van de voorgestelde software-architectuur is dat het de basis vormt voor het ontwerp van een familie van producten binnen één domein. De voordelen zijn bekend. Dezelfde architectuur kan gebruikt worden voor de hele lijn van producten, wat de kans vergroot op herbruikbaarheid van artefacten, variërend van broncode tot complete componenten en subsystemen. Aldus kunnen overeenkomsten gemakkelijk onderkend worden en neemt hergebruik toe, wat weer leidt tot lagere kosten en betere kwaliteit. Bovendien onstaat er onderhoudbaarheid doordat het creëren van verschillende systeemversies ondersteund wordt.

Hoofdstuk 7 gaat over detailontwerp en verificatie van systeemcomponenten en programma-objecten, met gebruik van formele methoden. In dit hoofdstuk worden modellen van stedelijke verkeerslichtregelingen gecreëerd met behulp van Petri-netten en gesimuleerd in een geautomatiseerde tool, waarmee specifieke scenario's gevalideerd kunnen worden. Deze simulaties kunnen niet garanderen dat een model geheel foutvrij is, maar ze kunnen wel al veel ontwerpfouten opsporen. Dus als de modellen zich gedragen zoals bedoeld, dan kunnen deze modellen vervolgens formeel geverifieerd worden met een of meer van de vele verificatiegereedschappen die door de theorie van Petri-netten geboden worden. Het voordeel is dan dat dezelfde modellen gehanteerd kunnen worden binnen verschillende activiteiten binnen systeemontwerp. In dit proefschrift worden Petri-netten geanalyseerd door middel van invariantenanalyse, de bereikbaarheidsgraaf en bewijsvoering met Lineaire Logica.

Hoofdstuk 8 gaat over de evaluatie van de geschiktheid van het combineren van semi-formele modelleertalen (UML en SysML) met een formele modelleertaal (Petri-netten) binnen het 4+1 View Model of Architecture, evenals de toepasbaarheid van deze benadering voor het ontwerp van gedistribueerde, tijd-kritische systemen. De aanpak voor de evaluatie omvatte de toepassing ervan in een echte ontwikkelomgeving. Dit staat bekend als "de industrie als laboratorium". De resultaten van deze evaluatie worden beschreven aan het einde van dit hoofdstuk. Deze resultaten worden gebruikt in hoofdstuk 9, waar conclusies worden geformuleerd, samen met de beperkingen van dit onderzoek, aanbevelingen, en voorstellen voor toekomstig onderzoek.

Curriculum Vitae

Michel dos Santos Soares was born on the 13th of April, 1978, in Rio de Janeiro, Brazil. After completing his pre-universitary education in 1995, he studied Computer Science at the Federal University of São Carlos (UFSCar), receiving the Bachelor degree in 2000. Then he worked as a System Analyst and Consultant in financial companies, before starting his Master degree in Computer Science at the Federal University of Uberlândia. He graduated in 2004 with the MSc thesis entitled "An approach based on a p-time Petri net Token Player and on the sequent calculus of Linear Logic for scenario verification of Real Time Systems specified by UML dynamic diagrams". Then he worked as an Assistant Professor and Consultant.

In 2006 he moved to The Netherlands to work on his PhD research at the faculty of Technology, Policy and Management, at the Delft University of Technology (TU Delft). As a member of the Information and Communication Section and participant of the Intelligent Infrastructures subprogram of the Next Generation Infrastructures Foundation, he worked on three projects (IHRTC, IHRTC-E, and C4C), all related to the application of Software Engineering methods, modeling languages and architectures in the design and analysis of Distributed Real-Time Systems, with Road Traffic Management Systems as case studies. He supervised one master student and was teaching assistant for three years for the course SPM2410. He published more than 40 articles in international conferences, books, and journals. He is co-author of the book "Qualidade de Software" (Software Quality), published in 2006.

NGInfra PhD Thesis Series on Infrastructures

1. Strategic behavior and regulatory styles in the Netherlands energy industry
   Martijn Kuit, 2002, Delft University of Technology, the Netherlands.

2. Securing the public interest in electricity generation markets, The myths of the invisible hand and the copper plate
   Laurens de Vries, 2004, Delft University of Technology, the Netherlands.

3. Quality of service routing in the internet: theory, complexity and algorithms
   Fernando Kuipers, 2004, Delft University of Technology, the Netherlands.

4. The role of power exchanges for the creation of a single European electricity market: market design and market regulation
   Franć cois Boisseleau, 2004, Delft University of Technology, the Netherlands, and University of Paris IX Dauphine, France.

5. The ecology of metals
   Ewoud Verhoef, 2004, Delft University of Technology, the Netherlands.

6. MEDUSA, Survivable information security in critical infrastructures
   Semir Daskapan, 2005,Delft University of Technology, the Netherlands.

7. Transport infrastructure slot allocation
   Kaspar Koolstra, 2005, Delft University of Technology, the Netherlands.

8. Understanding open source communities: an organizational perspective

Ruben van Wendel de Joode, 2005, Delft University of Technology, the Netherlands.

9. Regulating beyond price, integrated price-quality regulation for electricity distribution networks
   Viren Ajodhia, 2006, Delft University of Technology, the Netherlands.

10. Networked Reliability, Institutional fragmentation and the reliability of service provision in critical infrastructures
    Mark de Bruijne, 2006, Delft University of Technology, the Netherlands.

11. Regional regulation as a new form of telecom sector governance: the interactions with technological socio-economic systems and market performance
    Andrew Barendse, 2006, Delft University of Technology, the Netherlands.

12. The Internet bubble - the impact on the development path of the telecommunications sector
    Wolter Lemstra, 2006, Delft University of Technology, the Netherlands.

13. Multi-agent model predictive control with applications to power networks
    Rudy Negenborn, 2007, Delft University of Technology, the Netherlands.

14. Dynamic bi-level optimal toll design approach for dynamic traffic networks
    Dusica Joksimovic, 2007, Delft University of Technology, the Netherlands.

15. Intertwining uncertainty analysis and decision-making about drinking water infrastructure
    Machtelt Meijer, 2007, Delft University of Technology, the Netherlands.

16. The new EU approach to sector regulation in the network infrastructure industries
    Richard Cawley, 2007, Delft University of Technology, the Netherlands.

17. A functional legal design for reliable electricity supply, How technology affects law
Hamilcar Knops, 2008, Delft University of Technology, the Netherlands and Leiden University, the Netherlands.

18. Improving real-rime train dispatching: models, algorithms and applications
Andrea DAriano, 2008, Delft University of Technology, the Netherlands.

19. Exploratory modeling and analysis: A promising method to deal with deep uncertainty
Datu Buyung Agusdinata, 2008, Delft University of Technology, the Netherlands.

20. Characterization of complex networks: application to robustness analysis
Almerima Jamakovic, 2008, Delft University of Technology, Delft, the Netherlands.

21. Shedding light on the black hole, The roll-out of broadband access networks by private operators
Marieke Fijnvandraat, 2008, Delft University of Technology, Delft, the Netherlands.

22. On stackelberg and inverse stackelberg games & their applications in the optimal toll design problem, the energy markets liberalization problem, and in the theory of incentives
Katerina Stankova, 2009, Delft University of Technology, Delft, the Netherlands.

23. On the conceptual design of large-scale process & energy infrastructure systems: integrating flexibility,reliability, availability,maintainability and economics (FRAME) performance metrics
Austine Ajah, 2009, Delft University of Technology, Delft, the Netherlands.

24. Comprehensive models for security analysis of critical infrastructure as complex systems
Fei Xue, 2009, Politecnico di Torino, Torino, Italy.

25. Towards a single European electricity market, A structured approach for regulatory mode decision-making
Hanneke de Jong, 2009, Delft University of Technology, the Netherlands.

26. Co-evolutionary process for modeling large scale socio-technical systems evolution
Igor Nikolic, 2009, Delft University of Technology, the Netherlands.

27. Regulation in splendid isolation: A framework to promote effective and efficient performance of the electricity industry in small isolated monopoly systems
Steven Martina, 2009, Delft University of Technology, the Netherlands.

28. Reliability-based dynamic network design with stochastic networks
Hao Li, 2009, Delft University of Technology, the Netherlands.

29. Competing public values
Bauke Steenhuisen, 2009, Delft University of Technology, the Netherlands.

30. Innovative contracting practices in the road sector: cross-national lessons in dealing with opportunistic behaviour
Monica Altamirano, 2009, Delft University of Technology, the Netherlands.

31. Reliability in urban public transport network assessment and design
Shahram Tahmasseby, 2009, Delft University of Technology, the Netherlands.

32. Capturing socio-technical systems with agent-based modelling
Koen van Dam, 2009, Delft University of Technology, the Netherlands.

33. Road incidents and network dynamics, Effects on driving behaviour and traffic congestion
Victor Knoop, 2009, Delft University of Technology, the Netherlands.

34. Governing mobile service innovation in co-evolving value networks
Mark de Reuver, 2009, Delft University of Technology, the Netherlands.

35. Modelling Risk Control Measures in Railways
Jaap van den Top, 2009, Delft University of Technology, the Netherlands.

36. Smart heat and power: Utilizing the flexibility of micro cogeneration
Michiel Houwing, 2010, Delft University of Technology, the Netherlands.

37. Architecture-Driven Integration of Modeling Languages for the Design of Software-Intensive Systems
Michel dos Santos Soares, 2010, Delft University of Technology, the Netherlands.

Order information: info@nginfra.nl