Mälardalen University Licentiate Thesis
No.256

Pattern-based Specification and Formal Analysis of Embedded Systems Requirements and Behavioral Models

Predrag Filipovikj

April 2017

MÄLARDALEN UNIVERSITY

School of Innovation, Design and Engineering
Mälardalen University
Västerås, Sweden

ISSN 1651-9256
ISBN 978-91-7485-319-3
Printed by E-Print AB, Stockholm, Sweden
Distribution: Mälardalen University Press

Abstract

Since the first lines of code were introduced in the automotive domain, vehicles have transitioned from being predominantly mechanical systems to software intensive systems. With the ever-increasing computational power and memory of vehicular embedded systems, a set of new, more powerful and more complex software functions are installed into vehicles to realize core functionalities. This trend impacts all phases of the system development including requirements specification, design and architecture of the system, as well as the integration and testing phases. In such settings, creating and managing different artifacts during the system development process by using traditional, human-intensive techniques becomes increasingly difficult. One problem stems from the high number and intricacy of system requirements that combine functional and possibly timing or other types of constraints. Another problem is related to the fact that industrial development relies on models, e.g. developed in Simulink, from which code may be generated, so the correctness of such models needs to be ensured. A potential way to address of the mentioned problems is by applying computer-aided specification, analysis and verification techniques already at the requirements stage, but also further at later development stages. Despite the high degree of automation, exhaustiveness and rigor of formal specification and analysis techniques, their integration with industrial practice remains a challenge.

To address this challenge, in this thesis, we develop the foundation of a framework, tailored for industrial adoption, for formal specification and analysis of system requirements specifications and behavioral system models. First, we study the expressiveness of existing pattern-based techniques for creating formal requirements specifications, on a relevant industrial case study. Next, in order to enable practitioners to create formal system specification by using pattern-based techniques, we propose a tool called SeSAMM Specifier. Further, we provide an automated Satisfiability Modulo Theories (SMT)-based consistency analysis approach for the formally encoded system requirements specifications. The proposed SMT-based approach is suitable for early phases of the development for debugging the specifications. For the formal analysis of behavioral models, we provide an approach for statistical model checking of Simulink models by using the UPPAAL SMC tool. To facilitate the adoption of the approach, we provide the SIMPPAAL tool that automates procedure of generating network of stochastic timed automata for a given Simulink model. For validation, we apply our approach on a complex industrial model, namely the Brake-by-Wire function from Volvo GTT.

Sammanfattning

Från att de första kodraderna infördes inom fordonsindustrin så har fordon gått från att vara övervägande mekaniska till alltmer programvaruintensiva. Den ständigt ökande beräkningskraften och minnet i de inbyggda systemen i fordonen har lett till att nya mer kraftfulla och komplexa programvarufunktioner installeras för att realisera kärnfunktioner. Denna trend påverkar alla faser av systemutvecklingen, inklusive kravspecificering, design och arkitektur av systemet, samt integration och testning. I en sådan kontext så blir det allt svårare att skapa och hantera olika artefakter under utvecklingsfaserna genom att använda traditionella, människo-intensiva tekniker. Ett problem härrör från det stora antalet komplicerade systemkrav som kombinerar funktionella egenskaper med timing eller annat. Ett annat problem är relaterat till det faktum att systemutveckling i industrin grundar sig mycket i användandet av modeller, t.ex. utvecklade i Simulink, från vilken kod kan genereras, så det är viktigt att man kan garantera att sådana modeller är korrekta. Ett angreppssätt för att potentiellt lösa de nämnda problemen är att använda datorstöd för specificering, analys och verifiering redan vid kravspecificeringsfasen, men även vid senare utvecklingsfaser. Trots att ett sådant datorstöds höga grad av automatisering, fullständighet och stringens i formell specificering och analystekniker, är dess integration med industriell praxis en utmaning.

För att adressera denna utmaning så utvecklar vi i denna avhandling grunden för ett industriellt anpassat ramverk för formell specificering och analys av systemkrav och beteendemodeller. Först studerar vi uttrycksfullheten i befintliga mönsterbaserade tekniker för formella kravspecifikationer i kontexten av en relevant industriell fallstudie. Vidare, i syfte att göra det möjligt för utövare att skapa formella systemspecifikationer med mönsterbaserade tekniker, föreslår vi ett verktyg som heter SeSAMM Specifier. Därefter tillhandahåller vi en Satisfiability Modulo Theories (SMT)-baserad metod för automatisk konsistensanalys av formella systemkravsspecifikationer. Den föreslagna SMT-baserade metoden är lämplig för felsökning av specifikationer i tidiga utvecklingsfaser. För formell analys av beteendemodeller så tillhandahåller vi en strategi för statistisk model checking av Simulink-modeller med hjälp av verktyget UPPAAL SMC. För att underlätta införandet av metoden så presenterar vi verktyget simppaal som automatiserar generereringen av ett nätverk av tidsbaserade automater för en given Simulink modell. Vi validerar metoden genom att tillämpar den på en komplex industriell modell, nämligen funktionen Brake-by-Wire från Volvo GTT.

To my parents

Acknowledgements

I would like to express my deepest gratitude to my main advisor Associate Professor Cristina Seceleanu, my co-advisor Dr. Guillermo Rodriguez-Navas and my industrial co-advisor Professor Mattias Nyberg from Scania. Thank you for your patience, guidance and support. This thesis would not have been possible without you.

I would like to thank all the professors and lecturers at the department for unselfishly sharing their knowledge. It was a pleasure attending the courses and learning from you. Especially I would like to thank Prof. Ivica Crnkovic for introducing me to the world of research.

Many thanks to my fellow PhD students and rest of the staff at the department. I really enjoyed the time spent with you guys!

This thesis really would not have been the same without the support from our industrial partners from Volvo Group Trucks Technology, Sweden and Scania, Sweden. During the past three years, I had the pleasure and privilege to work with amazing people from Scania. I would like to thank all of them, and especially my managers Jon Andersson and Peter Rödin.

I would like to thank the grading committee members Professor Jan Carlson and Associate Professor Patrizio Pelliccione, and especially the faculty examiner Professor Stefania Gnesi for reviewing this thesis. Thank you for your time and effort.

Last but not least, I would like to thank my parents. There are no words that can express what I feel for them, how much I love them, and how proud I am to be their son. I dedicate all my work to you!

Predrag Filipovikj
Västerås, March, 2017

List of publications

Publications included in the licentiate thesis[1]

Paper A *Reassessing the Pattern-Based Approach for Formalizing Requirements in the Automotive Domain.* Predrag Filipovikj, Mattias Nyberg, Guillermo Rodriguez-Navas. In Proceedings of the 22nd IEEE International Requirements Engineering Conference (RE'14), pages 444-450. Karlskrona, Sweden. August 2014. IEEE Computer Society.

Paper B *Integrating Pattern-based Formal Requirements Specification in an Industrial Tool-chain.* Predrag Filipovikj, Trevor Jagerfield, Mattias Nyberg, Guillermo Rodriguez-Navas, Cristina Seceleanu. In Proceedings of the 10th IEEE International Workshop on Quality Oriented Reuse of Software (QUORS'16), collocated with COMPSAC 2016, pages 167-173. Atlanta, USA. June 2016. IEEE Computer Society.

Paper C *SMT-based Consistency Analysis of Industrial Systems Requirements.* Predrag Filipovikj, Guillermo Rodriguez-Navas, Mattias Nyberg, Cristina Seceleanu. In Proceedings of the 32nd ACM SIGAPP Symposium On Applied Computing (SAC 2017). **Best Paper Award**. Marrakesh, Morocco. April 2017. ACM.

Paper D *Analyzing Industrial Simulink Models by Statistical Model Checking.* Predrag Filipovikj, Nesredin Mahmud, Raluca Marinescu, Guillermo Rodriguez-Navas, Cristina Seceleanu, Oscar Ljungkrantz , Henrik Lönn. MRTC Report, Mälardalen Real-Time Research Center, Mälardalen University. March, 2017. *NOTE:* This publication is an extended version of the following article: *Simulink to UPPAAL Statistical Model Checker:*

[1]The included articles have been reformatted to comply with the licentiate thesis layout.

Analyzing Automotive Industrial Systems. Predrag Filipovikj, Nesredin Mahmud, Raluca Marinescu, Cristina Seceleanu, Oscar Ljungkrantz, Henrik Lönn. In Proceedings of the 21st International Symposium on Formal Methods (FM2016). Limassol, Cyprus. November 2016. Springer, LNCS.

Additional publications, not included in the licentiate thesis

1. *Simulink to UPPAAL Statistical Model Checker: Analyzing Automotive Industrial Systems.* Predrag Filipovikj, Nesredin Mahmud, Raluca Marinescu, Cristina Seceleanu, Oscar Ljungkrantz, Henrik Lönn. In Proceedings of the 21st International Symposium on Formal Methods (FM2016). Limassol, Cyprus. November 2016. Springer, LNCS.

2. *Increasing Embedded Systems Quality through Automated Specification and Analysis of Requirements and Behavioral Models.* Predrag Filipovikj. 43rd International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM2017), Student Research Forum. **Best Student Research Proposal Award**. Limerick, Ireland. January 2017.

Contents

I

Thesis

Chapter 1

Introduction

Embedded systems are a special class of computer systems that consists of dedicated hardware and software designed for specific purposes, preprogrammed to perform a predefined set of tasks. We interact with such systems on a regular basis, as they are integral parts of our lives, in forms of home appliances, mobile phones, entertainment systems, hand-held computers, etc. Embedded systems are also used in more specialized and safety-critical contexts such as chemical and nuclear power plants, robotic production lines, all kinds of transportation systems, etc. Nowadays, with the ever-increasing memory capacity and computational power of embedded hardware platforms, we are producing embedded software so big and complex, that it practically revolutionizes and drives entire industries.

The automotive industry is one of the many that has been greatly influenced by the increased versatility and power of the embedded systems. Traditionally, vehicles were predominantly mechanical systems that, in addition, used simple electronics mostly for infotainment that included features such as radio, media player, air-conditioner, etc. However, with the introduction of a new generation of vehicular embedded systems with increased computational power and operational memory, a new trend has started (often regarded as *"drive-by-wire"* trend) in the automotive industry. In this trend, vehicular features traditionally implemented using mechanical parts are being replaced by complex software functions. Since then, embedded systems have become the centerpiece of the vehicle development, with major expenses for producing new vehicles being associated to software and the hardware that runs it [1]. Additionally, embedded systems, especially the embedded software functions have become the major area of innovations within the domain [2], meaning that better, faster and more reliable vehicular software is seen as main advantage of a vehicular manufacturer over its competitors.

The software in vehicles runs on distributed processing units called Electrical Control Units (ECUs). The set of all ECUs, plus the sensors, actuators and the communication lines constitute the Electrical/Electronic (E/E) system of a vehicle. Typically, the E/E system of a modern vehicle consists of more than 70 independent ECUs, which combined run more than 10 million lines of code [3, 4]. The size and complexity of the vehicular software functions impact all the phases of system development, including the requirements specification, design and architecture of the system, as well as the integration and testing phases [3].

A significant number of the software functions have implications on the overall safety of the vehicle, thus they are being categorized as *safety critical*. For developing high quality software the engineers in the automotive domain rely on different state-of-the-practice techniques including manual peer-review for requirements quality assurance [5], model-based development (MBD) for problem abstraction, documenting the solution, as well as testing at various levels (unit, integration testing, etc.). However, the effectiveness of such techniques is limited as they do not scale well with the size and complexity of the systems. The limitation is due to the fact that these techniques are performed manually. Additionally, the verification effort for the traditional verification techniques (testing and simulation) grows proportionally with the size and complexity of the systems. A way to assure high level of correctness in such settings is to use specialized computer programs that automatically and with high rigor assess the correctness of the system. Such techniques are called *formal verification methods*. The goal is that a system represented as a precise mathematical model (formal model) can be checked that it does not deviate from its intended behavior, expressed as a set of logical properties (formal system specification). The potential benefit of adopting formal verification techniques for analyzing automotive software has been acknowledged in the latest standard for automotive safety [6], according to which utilization of formal verification techniques is highly recommended. However, the adoption of formal techniques in industrial settings is limited by the difficulty of creating formal system specifications and generating formal system models.

In this thesis, we develop the foundation of a framework for formal specification, analysis and verification of system requirements and behavioral models of embedded systems, suitable for adoption in industrial settings, with special emphasis on the automotive domain. The goal of the framework is to enable the engineers who are not experts in formal methods to use formal techniques for increasing the confidence in the correctness of software. In order to support our goal, we propose contributions on three fronts. First, we focus on providing an engineer-friendly way for creating formal system requirements specifications (further in the thesis referred as system specifications). To achieve this, as a first research endeavor, we assess the suitability and expressiveness of an existing technique, called real-time specification patterns (RTSPS) [7] for formalizing system specifications in the automotive domain [8]. To evaluate the expressiveness of the given technique, we conduct a case study in which we formalize a set of requirements gathered from several functions installed in vehicles produced by Scania, Sweden. The results of the case study are aligned with the results that emerged from earlier attempts carried out by research fellows [9], which reveal that RTSPS patterns, in principle, are expressive enough for formalizing automotive requirements, yet they bear important challenges to be addressed, such as validation of the formal requirements encoding and the steep learning curve on how to use the patterns. To address these challenges, we propose a tool called *SeSAMM Specifier* as our second contribution [10]. The tool enables formal specification of requirements by using the specification patterns without being specifically bound to a predefined catalog of patterns. This feature allows the tool to be extensible and customizable in order to fulfill the needs of specific users. Additionally, the tool provides a validation mechanism by visualizing the behavior of the formalized requirements using different mechanisms. To make it applicable in industry, we integrate the tool into an existing toolchain (SeSAMM) [11] developed at Scania.

Next, we propose an approach for automated consistency analysis of formally encoded requirements of industrial embedded systems based on Satisfiability Modulo Theories (SMT) [12], by using the state-of-the-art SMT solver Z3 [13]. The proposed approach belongs to the class of *"model-free"* sanity checks [14], as it does not require details of the system's behavioral or architectural model, and is suitable for early debugging of system requirements formalized using the specification patterns. To enable the SMT-based consistency analysis, we provide: a) a pattern-based transformation of the Timed Computation Tree Logic (TCTL) formulas into Z3 assertions, b) a set of rules for simplifying the original Z3 assertions by reducing the number of quantified variables and quantifiers in the assertions and c) a mitigation strategy for non-solvable requirements such that the procedure terminates [15].

As our final contribution, we propose a methodology for the formal analysis of Simulink models by means of statistical model checking [16] using the statistical model checker UPPAAL SMC [17]. To enable this, we propose a pattern-based transformation of Simulink models into stochastic timed automata. To enable the transformation, first, we define the formal semantics of the most frequently used Simulink blocks into stochastic timed automata. Second, we propose stochastic timed automata patterns for discrete- and continuous-time Simulink blocks. Third, we propose a flattening algorithm for composite Simulink blocks and a synchronization mechanism for preserving the block execution order generated by Simulink. Finally, we validate our approach on an industrial use-case, namely the prototype Brake-by-Wire (BBW) system.

1.1 Thesis Overview

The thesis is divided into two major parts. The first part is an overall summary of the thesis, organized as follows. In Chapter 2, we give a short overview of the preliminaries; in Chapter 3, we describe the research method used for conducting the research and producing the research results described in the thesis. Chapter 4 introduces the research goals of the thesis. In Chapter 5, we briefly describe the contributions of the thesis, and map them to the corresponding research goals, respectively. The overview and comparison to the related work is given in Chapter 6, after which we conclude the first part of the thesis and present the directions for future work in Chapter 7.

The second part of the thesis is given as a collection of publications that encompass all the thesis contributions. The included papers are:

Paper A. *Reassessing the Pattern-Based Approach for Formalizing Requirements in the Automotive Domain.* Predrag Filipovikj, Mattias Nyberg, Guillermo Rodriguez-Navas. In Proceedings of the 22nd IEEE International Requirements Engineering Conference (RE'14), pages 444-450. Karlskrona, Sweden. August 2014. IEEE Computer Society.

Abstract. The importance of using formal methods and techniques for verification of requirements in the automotive industry has been greatly emphasized with the introduction of the new ISO26262 standard for road vehicles functional safety. The lack of support for formal modeling of requirements still represents an obstacle for the adoption of the formal methods in industry. This paper presents a case study that has been conducted in order to evaluate the difficulties inherent to the process of transforming the system requirements from their traditional written form into semi-formal notation. The case study focuses on a set of non-structured functional requirements for the Electrical and Electronic (E/E) systems inside heavy road vehicles, written in natural language, and reassesses the applicability of the extended Specification Pattern System (SPS) represented in a restricted English grammar. Correlating this experience with former studies, we observe that, as previously claimed, the concept of patterns is likely to be generally applicable for the automotive domain. Additionally, we have identified some potential difficulties in the transformation process, which were not reported by the previous studies and will be used as a basis for further research.

Contributions. I was the main driver of the paper. I have performed most of the activities related to the case study, including the requirements gathering and extraction, applying the patterns and drawing conclusions. I also wrote most of the paper. Guillermo Rodriguez-Navas and Mattias Nyberg participated in discussions and contributed with ideas and comments on the patterning process.

Paper B. *Integrating Pattern-based Formal Requirements Specification in an Industrial Tool-chain.* Predrag Filipovikj, Trevor Jagerfield, Mattias Nyberg, Guillermo Rodriguez-Navas, Cristina Seceleanu. In Proceedings of the 10th IEEE International Workshop on Quality Oriented Reuse of Software (QUORS'16), collocated with COMPSAC 2016, pages 167-173. Atlanta, Georgia, USA. June 2016. IEEE Computer Society.

Abstract. The lack of formal system specifications is a major obstacle to the widespread adoption of formal verification techniques in industrial settings. Specification patterns represent a promising approach that can fill this gap by enabling non-expert practitioners to write formal specifications based on reusing solutions to commonly occurring problems. Despite the fact that the specification patterns have been proven suitable for specification of industrial systems, there is no engineer-friendly tool support adequate for industrial adoption. In this paper, we present a tool called SESAMM Specifier in which we integrate a subset of the specification patterns for formal requirements specification, called SPS, into an existing industrial tool-chain. The tool provides the necessary means for the formal specification of system requirements and the later validation of the formally expressed behavior.

Contributions. I was the main driver of the work. The SeSAMM Specifier tool was implemented by Trevor Jagerfield, while I was the main architect of the tool. The rest of the coauthors participated in discussions contributing with ideas about the tool implementation and the paper. I wrote most of the paper, with the rest of the coauthors writing some parts and giving valuable feedback.

Paper C. *SMT-based Consistency Analysis of Industrial Systems Requirements*. Predrag Filipovikj, Guillermo Rodriguez-Navas, Mattias Nyberg, Cristina Seceleanu. In Proceedings of the 32nd ACM SIGAPP Symposium On Applied Computing (SAC 2017). Marrakesh, Morocco. April 2017. ACM.

Abstract. As the complexity of industrial systems increases, it becomes difficult to ensure the correctness of system requirements specifications with respect to certain criteria such as consistency. Automated techniques for consistency checking of requirements, mostly by means of model checking, have been proposed in academia. However, such approaches can sometimes be costly in terms of modeling and analysis time or not applicable for certain types of properties. In this paper, we present a complementary method that relies on pattern-based formalization of requirements and automated consistency checking using the state-of-the-art SMT tool Z3. For validation, we apply our method on a set of timed computation tree logic requirements of an industrial automotive subsystem called the Fuel Level Display.

Contributions. I was the main driver of the paper. I collected the requirements which were included in the case study. Also, I did the requirements formalization via specification patterns and encoding them into Z3 assertions. Additionally, I wrote most of the paper. Cristina Seceleanu and Guillermo Rodrigues-Navas contributed with useful comments for the proofs and structure of the paper. Mattias Nyberg provided feedback on the formalization of the requirements from the FLD system.

Paper D. *Analyzing Industrial Simulink Models by Statistical Model Checking*. Predrag Filipovikj, Nesredin Mahmud, Raluca Marinescu, Guillermo Rodriguez-Navas, Cristina Seceleanu, Oscar Ljungkrantz, Henrik Lönn. Mälardalen Real-Time Research Centre, Mälardalen University (MRTC Technical Report). March 2017. Mälardalen University Press.

Abstract. The evolution of automotive systems has been rapid. Nowadays, electronic brains control dozens of functions in vehicles, like braking, cruising, etc. Model-based design approaches, in environments such as

MATLAB Simulink, seem to help in addressing the ever-increasing need to enhance quality, and manage complexity, by supporting functional design from predefined block libraries, which can be simulated and analyzed for hidden errors, but also used for code generation. For this reason, providing assurance that Simulink models fulfill given functional and timing requirements is desirable. In this paper, we propose a pattern-based, execution-order preserving automatic transformation of Simulink atomic and composite blocks into stochastic timed automata that can then be analyzed formally with UPPAAL Statistical Model Checker (UPPPAAL SMC). Our method is supported by the tool SIMPPAAL, which we also introduce and apply on an industrial prototype called the Brake-by-Wire system. This work enables the formal analysis of industrial Simulink models, by automatically generating their semantic counterpart. .

Contributions. Together with Nesredin Mahmud I was the main driver and contributor to the paper. I have designed and implemented the SIMPPAAL tool as well as most of the plug-ins for generating the block routines. I wrote three complete sections and three additional subsections in the paper. Further, I applied the SIMPPAAL tool on the Brake-by-Wire Simulink model to generate the network of STA. Nesredin Mahmud has developed a subset of the block routine plug-ins. Raluca Marinescu validated the correctness of the generated Brake-by-Wire UPPAAL model and performed the SMC analysis on the same. Cristina Seceleanu developed the proof of correctness for the transformation and wrote one section of the paper, also providing useful comments. Guillermo Rodriguez-Navas wrote one section in the paper and provided useful comments. Oscar Ljungkrantz and Henrik Lönn provided valuable feedback both on the approach and the final version of the paper.

Chapter 2

Preliminaries

In this chapter, we introduce the preliminary concepts that are used throughout the thesis. First, in Section 2.1 we present the model-based development paradigm. Next, in Section 2.2 we give an overview of the Simulink tool. In Section 2.3, we present an overview of the formal modeling, verification and analysis techniques. In Section 2.4, we give an overview of the Satisfiability-Modulo Theories (SMT) and the Z3 SMT solver. Next, in Section 2.5 we describe the specification patterns, and finally, we conclude the chapter with Section 2.6 with an overview of the sanity checking techniques.

2.1 Model-based Development

The biggest problem in all engineering disciplines is the complexity of the underlying systems, and software engineering is no exception. The complexity of any engineering problem can be observed from two standpoints [18]: first, there is the inherent complexity of the problem itself, and second, the additional complexity arising from the tools and methods which are used for solving the problem. Model-based development (MBD) has proven to be an effective paradigm for developing complex systems. It facilitates system modeling through multiple abstractions or views, corresponding to the system's development phases. This enables the seamless integration of design and analysis techniques and tools throughout the system development.

Abstraction refers to the process of removing irrelevant details in order to focus on the essential parts of the system. An abstracted version of the system, as observed from a particular point of view is called *model*. The main characteristic of MBD is raising the abstraction level of the software systems by shifting the focus from coding to modeling activities. To be useful, any software model should be [19]:

- *Abstract* - the main purpose of the model is to hide all the irrelevant details, such that the important features stand out;
- *Understandable* - the model must convey the information of interest in a clear and unambiguous way;
- *Accurate* - the model must be an accurate abstraction of the system, that is, it has to correctly reflect the properties of interest;
- *Predictable* - the model should behave in the expected way.

Any software model that has the above listed characteristics is usable in different ways. Apart from abstracting the problem, the models that are understandable for the domain engineers can be used for documenting the solution. The accuracy of the models, combined with their predictability allow one to treat them as executable specifications. This is possible only if the language used for developing the model has well-defined semantics, such that specialized tools can be used to generate executable code from the model directly.

Given all the above, the MBD paradigm, in essence, provides two key benefits [19]: first, MBD raises the levels of abstraction of the specification to be closer to the problem domain while hiding the implementation details, and second, it facilitates the automation by bridging the gap between the specification (the model) and the actual implementation (the code).

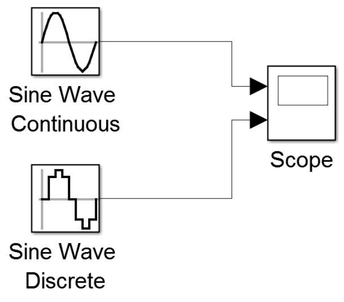
MBD paradigm has become the “go-to” way for developing software in the automotive domain. This is due to the fact that still, most engineers who develop automotive software are experts in domains such as electrical engineering, hydraulics, mechanics, etc., but have limited skills in coding and software development in general. The MBD paradigm allows them to abstract away the implementation of their solutions, by using specialized tools, which let them to model their solutions in easy and intuitive ways. One such tool for modeling, simulation and code generation in industrial settings is Simulink, which we introduce in the following section.

2.2 Simulink

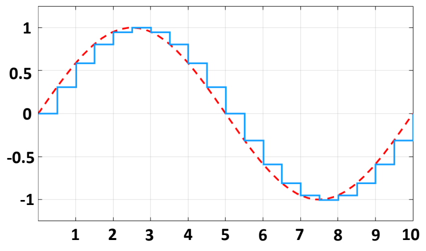
Simulink (often referred to as Matlab Simulink) is a graphical programming environment for modeling, simulation and code generation for multi-domain dynamic systems, integrated into the Matlab environment developed by MathWorks [20]. Due to its versatility, Simulink has become the de-facto standard for MBD in the automotive domain. A Simulink model is a hierarchical representation of the system, composed of sets of blocks that communicate via signals. Simulink blocks can be either *atomic* or *composite*. An atomic Simulink block represents a basic module that computes an equation or another modeling concept in order to produce an output, ether continuously or at predefined time points.

Simulink by default provides a library that contains a number of atomic blocks. These atomic blocks have a predefined behavior and allow customization only via block specific parameters. For example, the `Gain` Simulink block allows multiplication of the value of an input signal by a given value (scalar or vector). The input-output relation of the `Gain` block is predefined, and as such cannot be modified. In order to facilitate extension of the existing library of atomic blocks, Simulink introduces the concept of `S-function`, which allows one to define an atomic Simulink block by specifying its behavior in Matlab, C, C++ or Fortran. One can additionally apply the concept of masking, by using a special extension called `Mask`, to define the interface of the newly-introduced block and encapsulate its behavior as a black-box.

The composite Simulink blocks are used to describe the hierarchical structure of the model. They are usually represented via the `Subsystem` block, which has multiple variations such as the `Triggered Subsystem`, `Referenced Subsystem`, etc. Unlike the atomic ones, composite Simulink blocks do not have predefined behavior. Instead, the behavior of the composite blocks is modeled as a set of atomic blocks. Simulink allows arbitrary levels of nesting composite blocks inside of a model. Composite blocks can be either *virtual* or *atomic*. The encapsulated blocks of the virtual subsystem blocks are invoked according to the overall system model, whereas for the the atomic (also called non-virtual) subsystems the encapsulated blocks are executed as a single unit that can be conditionally executed based on an external triggering, function-call, or enabling input. Once triggered, all the blocks inside an atomic subsystem are executed as an atomic unit, meaning that there is no interleaving with other blocks outside the subsystem. All the blocks inside a triggered subsystem are discrete, as they update their outputs only when the external triggering event occurs. To facilitate reuse, Simulink allows the



(a)



(b)

Figure 2.1: Example Sine Wave Block: (a) Simulink Diagram and (b) Simulation Result

contents of a given subsystem to be saved into a separate model file.

Based on how they update their output, we classify all Simulink blocks, be they composite or atomic into two categories: *continuous-time* and *discrete-time*. The continuous-time blocks produce new outputs continuously, whereas the discrete-time blocks produce new outputs at predefined points in time, determined based on *sample time* of the block. Another exclusive feature of the discrete-time Simulink blocks is the possibility to delay the first execution, specified as *offset* of the block. In case when the offset of a given block is greater than zero, the subsequent outputs are produced relative to the offset period and not the beginning of the simulation. Figure 2.1 shows an example of `Sine` block as modeled in Simulink (Figure 2.1a) and simulation traces for both continuous (dashed red line) and discrete (full blue line) behavior of the same (Figure 2.1b). During simulation, Simulink uses an internal algorithm to determine the order at which the blocks inside the model are executed. The list of execution order of the blocks during simulation is called the *sorted order list* or *slist* for short. It can be obtained by running the `slist` command while the Matlab is in debug mode.

Simulink capabilities are extended by two supportive tools: Simulink Design Verifier (SDV) and Simulink Coder, both provided by Matlab. As advertised by the vendor, the SDV tool uses formal methods to detect design errors, such as integer overflow, division by zero, dead logic, array access violations, etc. It can also verify system requirements expressed as verification objectives, which are in fact simple Simulink models. The Simulink Coder is used for automatic generation of executable `C` or `C++` code, which is then deployed and run on the vehicle.

2.3 Formal Modeling and Verification

Formal verification is a set of techniques based on mathematics, which are used to rigorously prove the correctness of a system model expressed in some formal notation. Compared to other verification techniques such as simulation and testing, formal verification techniques are deemed to deliver a higher degree of assurance. Due to this, formal verification techniques can be used for proving the absence of certain types of errors. There is a number of formal verification techniques, which in principle can be divided into two categories: *deductive*, used for proving the correctness of the system based on a number of axioms and a set of proof rules, and *model checking*, which is a technique that performs a systematic and exhaustive state space exploration to determine whether the system model conforms to a set of defined logical properties. In practice, model checking is preferred over deductive techniques as the verification algorithm is fully automated.

2.3.1 Model Checking

As mentioned above, *model checking* is an automated technique that checks a finite-state abstract system model in a systematic and exhaustive manner, to prove whether it satisfies a given property modeled in logic. Model checking if fully automated and is performed by a verifier tool called *model checker*.

The core of model checking is the verification algorithm, performed by the model checker. The input to the model checker is a system model expressed in a formal notation and a set of formally specified logical properties. For verification of qualitative properties (that admit a yes/no answer) there are two possible outcomes of the model checking procedure. If the model conforms to a given property, the model checker returns a positive answer. For reachability and some liveness properties (e.g., something good will eventually happen) the model checker returns a witness trace in case of fulfillment. Then, the model checking activity can be continued for the rest of system properties. When a safety property is not satisfied, the model checker generates a counter example, which is usually a path (error trace) to the state that violates the property.

Due to its systematic approach and the exhaustiveness of the state space exploration, the model checking procedure can handle models with state spaces up to a certain size, above which there is not enough memory to store new

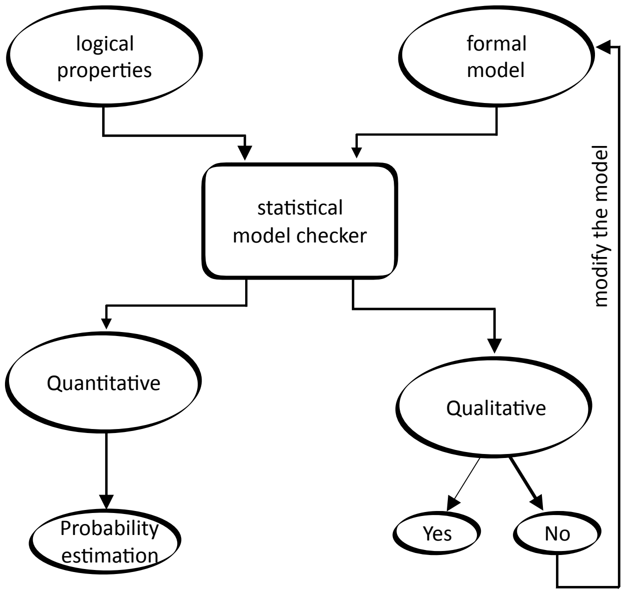


Figure 2.2: Statistical model checking procedure.

states. This is known as the state space explosion problem. However, with the latest advances in the area of model checking, such as using optimal data structures and smart algorithms, modern model checkers such as UPPAAL [21], Spin [22] or NuSVM [23] can be applied on system models with state spaces up to 10^{476} states [24].

Since in this thesis, we aim at applying model checking on industrial systems, the exhaustive model checking is likely not to scale. Therefore, we resort to a special type of model checking called *statistical model checking* (SMC), which computes the probability that a model satisfies a given property up to some probability, based on a finite number of model simulations. A high-level overview of the SMC is given in Figure 2.2. SMC uses a series of simulation-based techniques to answer two types of questions: i) *Qualitative*: is the probability of a given property being satisfied by a random system execution greater or equal than some threshold? and ii) *Quantitative:* what is the probability that a random system execution satisfies a given property? The qualitative properties are also referred to as *hypothesis testing*, while the quantitative are called *probability estimation*. In both cases, the answer provided by the procedure will be correct up to a certain level of confidence. Since the statistical model checking is less memory intensive than the model checking, it can be used to statistically verify models with infinite state spaces. Even though the technique is less precise than the exact model checking, still it solves the verification problem in a rigorous and efficient way.

In our work, we use UPPAAL Statistical Model Checker (SMC) [17] for formal analysis of industrial models used as case studies. The input of the UPPAAL SMC tool is a network of stochastic timed automata and a set of properties formalized in temporal logic. In the following sub-sections, we briefly overview the timed automata and stochastic priced timed automata frameworks, and the temporal logic used for specification of the system properties and the UPPAAL SMC model-checker.

2.3.2 Timed Automata and Stochastic Priced Timed Automata (SPTA)

Timed automata [25] is an extension of finite-state automata with a set or real-valued variables called *clocks*, suitable for modeling the behavior of real-time systems. The clocks are non-negative variables that grow at a fixed rate with the passage of time, and can be reset to zero. The formal definition of a timed automaton (TA) is given as the following tuple:

$$TA = \langle L, l_0, X, \Sigma, E, I \rangle \tag{2.1}$$

where: L is a finite set of locations, $l_0 \in L$ is the initial location, X is a finite set of clocks, Σ is a finite set of actions, including synchronization and internal actions, $E \subseteq L \times B(X) \times \Sigma \times 2^X \times L$ is a finite set of edges of type $e = (l, g, a, r, l')$, where l and l' are the source and the sink locations of the edge, respectively, g is a predicate on $\mathbb{R}^X$ called guard, $a \in \Sigma$ is the action label, and r is the set of clocks that are reset when the edge is traversed. $I : L \to B(X)$ is a function that assigns invariants to locations, which bound the time allowed in a particular location. An edge is going to be traversed if its guard g evaluates to true. $B(X)$ represents the set of formulas called *clock constrains* of the following form $x \bowtie c$ or $x - z \bowtie c$, where $x, z \in X$, $c \in \mathbb{N}$ and $\bowtie \, \in \{<, \leq, =, \geq, >\}$. A clock constraint is downwards closed if $\bowtie \, \in \{<, \leq, =\}$.

The semantics of TA is defined as a *timed transition system* $(S, \to)$, where S is set of states, and the $\to$ is the transition relation that defines how the system evolves from one state to another. A state in the system is a pair (l, v), where l is the location and the v is the valuation of the clocks. A timed automaton can proceed, that is, move to a new state, by performing either a *discrete* or a *delay* transition. By executing a *discrete* transition the automaton transitions from one location into another without any time delay, whereas by executing a *delay* transition the automaton stays in the same location while time passes. A *path* (or trace) σ of a TA is an infinite sequence $\sigma = s_0a_0s_1a_1s_2a_2...$ of states alternated by transitions, be they discrete or delay, such that $s_i \xrightarrow{a_i} s_{i+1}$.

A system can be modeled as a set of communicating components. Let $A_1, A_2, ...A_n$ be a set of timed automata each corresponding to an individual component in the system. A *network* of timed automata (NTA) is simply a parallel composition $A_1 \| A_2 \| \cdots \| A_n$ of a finite number of timed automata.

Stochastic Priced Timed Automata

Priced Timed Automata [26] extend timed automata with a set of real-valued variables that evolve at different rates. The formal definition of a priced timed automaton (PTA) is given by the following equation:

$$PTA = \langle L, l_0, X, \Sigma, E, R, I \rangle \tag{2.2}$$

where: L is a finite set of locations, $l_0 \in L$ is the initial location, X is a finite set of real-valued variables, $\Sigma = \Sigma_i \uplus \Sigma_o$ is a finite set of actions partitioned into inputs (Σ_i) and outputs (Σ_o), $E \subseteq L \times \mathcal{L}(X) \times \Sigma \times 2^X \times L$ is a finite set of edges where $\mathcal{L}(X)$ denotes the set of lower bound guards over X, $R : L \to \mathcal{L}^X$ assigns a rate vector to each location, and $I : L \to \mathcal{U}(X)$ assigns an invariant to each location where $\mathcal{U}(X)$ denotes the set of upper bound guards over X.

The stochastic priced timed automata (SPTA) [17] extend the PTA with a delay density function (μ), which is a set of all density delay functions $\mu_s \in L \times R^X$, which can be either uniform or exponential distribution, and an output probability function (γ), which is the set of all output probability functions γ_s over the Σ_o output edges of the automaton. Assuming the formal definition of PTA given above, the formal definition of a SPTA is given by the following tuple:

$$SPTA = \langle PTA, \mu, \gamma \rangle \tag{2.3}$$

The stochastic semantics of a priced timed automaton $SPTA$ with a corresponding set of states S is defined based on the probability distributions for both delays and outputs for each state $s = (l, v) \in S$ of the automaton [26].

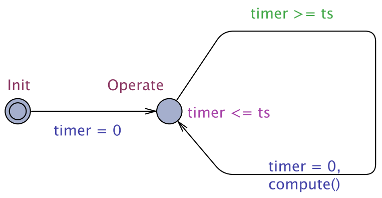
The *delay density function* (μ_s) over delays in $\mathbb{R}_{\geq 0}$ (positive reals), is either a uniform or an exponential distribution depending on the invariant in l. With E_l we denote the disjunction of guards g such that $e = (l, g, o, -, -) \in E$ for some output o. Then, $d(l, v)$ denotes the infimum delay before the output is enabled $d(l, v) = \inf\{d \in \mathbb{R}_{\geq 0} : v{+}d \models E_l\}$, whereas $D(l, v) = \sup\{d \in \mathbb{R}_{\geq 0} : v + d \models I(l)\}$ is the supremum delay. If the supremum delay $D(l, v) < \infty$, then the delay density function μ_s in a given state s is a uniform distribution over the interval $[d(l, v), D(l, v)]$. Otherwise, when the upper bound on the delays out of s does not exist, μ_s is an exponential distribution with a rate $P(l)$, where $P : L \rightarrow \mathbb{R}_{\geq 0}$ is an additional distribution rate specified for the automaton. The *output probability function* γ_s for every state $s = (l, v) \in S$ is the uniform distribution over the set $\{o : (l, g, o, -, -) \in E \wedge v \models g\}$.

The stochastic semantics of networks of SPTA (NSPTA) is based on the principle of independence between the components [26]. Each component, based on the delay density function and the output probability function repeatedly decides on which output to generate and at which point in time. In such race between components, the output will be determined by the component that has chosen to produce output after the minimum delay.

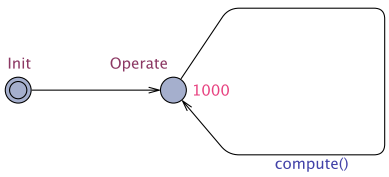
In our work, for encoding the patterns, we use SPTA with real-valued clocks that evolve with implicit rate 1. These automata are in fact timed automata with stochastic semantics, called stochastic timed automata (STA). A network of STA (NSTA) is a parallel composition of STA, defined in a similar way like NSPTA. The notion of SPTA is introduced due to the fact, that, for analysis we use monitor automata (composed in parallel with the actual system model) that implement the *stop-watch* mechanism, which renders the model an NSPTA.

2.3.3 UPPAAL Statistical Model Checker (SMC)

UPPAAL [21] in an integrated development environment for modeling, simulation and verification of real-time systems. It has been developed as a joint research effort by the Uppsala University and Aalborg University. The tool has been first released in 1995 and since has been constantly updated with new features. UPPAAL Statistical Model Checker (SMC) [27] is an extension of UPPAAL model checker for statistical model checking. The input language of the UPPAAL SMC is a network of STA. In the following section, we present an illustrative example of a stochastic timed automata as modeled in UPPAAL SMC.



(a) Timed Automata



(b) Stochastic Timed Automata

Figure 2.3: Illustrative scenario of UPPAAL stochastic timed automata

UPPAAL SMC Stochastic Timed Automata

In this section we present an illustrative example of an ordinary timed automaton and a stochastic timed automaton as supported by UPPAAL SMC tool.

The input language of the UPPAAL model checker extends the original timed automata framework with a number of features, including: constants, global and local data variables (integer variables with bounded domain), arithmetic operators, arrays, synchronization channels, urgent and committed locations, as well as definition of procedures using a subset of the C language [28]. UPPAAL SMC uses STA as defined above as its input language. A network of stochastic timed automata (NSTA) in UPPAAL is a parallel composition of a finite set of stochastic timed automata over X and Σ, synchronizing over channels and using shared variables.

Figures 2.3a and 2.3b show an example of stochastic timed automata as supported by UPPAAL SMC. The automaton in Figure 2.3a shows an example of an ordinary UPPAAL TA that models the behavior of a component in the system that periodically executes some computational routine (`compute()`) that maps inputs into outputs. It is composed of two locations: `Init` and `Operate`, out of which `Init` is marked to be the initial one, denoted by two concentric circles. On the edge from `Init` to the `Operate` location the automaton performs an *update* action, in this particular case being a reset of the clock variable `timer`. The `Operate` location is decorated with an invariant `timer` $\leq$ `ts`, denoting that the automaton is allowed to stay in that location as long as the value of the clock variable is smaller or equal to the value of the sample time (`ts`). The `Operate` location represents the operational mode of the automaton and has a single looping transition decorated with a *guard* expression `timer` $\geq$ `ts`. The automaton takes the looping edge on `Operate` location if the guard `timer` $\geq$ `ts` is satisfied, that is, as soon as `timer ==`

`ts`. On the same edge two update actions are performed, namely executing the computational routine that produces output from the execution (`compute()`), and reset of the clock variable. The computational routine is encoded as a `C` function. PTA uses the same modeling concepts as TA except that it allows real-valued values that can evolve non-linearly. For example, the TA in Figure 2.3a can be transformed into PTA if we consider energy consumption encoded as differential equation $e' = n$, where $n \in \mathbb{N}$ added to the `Operate` location.

Figure 2.3b shows an example of a timed automaton with stochastic semantics. The automaton is composed of the same two locations (`Init` - the initial one, and `Operate`) as the previous one in Figure 2.3a. The role of the automaton is to emulate the behavior of a component that executes continuously, that is, at very small time intervals. To model the continuous behavior of the component, instead of an invariant, the `Operate` location is decorated with a *rate of exponential*. The distribution parameter λ is the user-defined parameter in the delay function that calculates the probability of the automaton leaving the `Operate` location at each simulation step, given as: $Pr(\text{leaving after } t) = 1 - e^{-\lambda t}$. The greater the value of λ, the smaller is the probability that the automaton stays in the location.

2.3.4 Specifying Properties in Temporal Logic

In this section we give an overview of the different temporal logics used in this thesis for specifying properties of time-transition systems.

Computational Tree Logic (CTL) is a branching time logic used for formal specification of finite-state systems [29]. The interpretation of CTL is defined over a model M that consists of a non-empty set of states S, a labeling function $Label : S \rightarrow 2^{AP}$ that assigns a set of atomic propositions (AP) to each state in the model and a successor function $R : S \rightarrow S$ which assigns a set of successor states to each state $s \in S$.

The syntax of a CTL formula consists of quantifiers over paths and path-specific temporal operators. In CTL, there are two path quantifiers: a universal one "A" meaning *"for all paths"*, and an existential one "E" denoting *"there exists a path"*. A valid CTL formula is of the type $\varphi\, U\, \psi$, where "U" (*"until"*) represents the basic path-specific temporal operator, that can be combined with either of the path quantifiers. There are two additional derived path-specific temporal operators, given as follows: the *"Future"* temporal operator (denoted as F or $\Diamond$), denoting that a formula eventually becomes `true`, with its semantics defined as: $F\varphi \Leftrightarrow true\ U\ \varphi$; and the *"Globally"* path-specific temporal operator (denoted as G or $\Box$) meaning that a given formula is always `true`, with the following semantics: ($G\varphi \Leftrightarrow \neg\, F\, \neg\varphi$). There exists also a weaker version of the U operator called *"weak-until"* (denoted as W) defined as: $\varphi\, W\, \psi \Leftrightarrow (\varphi\, U\, \psi) \;\vee\; G\varphi$, which is used to capture formulas where the right hand side term (ψ) might never be satisfied.

Timed CTL (TCTL) [30] is a an extension of the CTL with clock constraints. In TCTL each of the path-specific operators has a timed version that uses constrains over clocks. In this thesis, we use the following syntax: $Operator_{\bowtie T}$, where $Operator \;\in\; \{U, F, G, W\}$ and $\bowtie \in\; \{<, \leq, =, \geq, >\}$ and T is a numeric bound on the real-valued variable. For instance the formula $EF_{\leq T}\varphi$ requires that there exists an execution path along which φ eventually becomes `true` within T time units.

For specifying probabilistic time-constrained temporal properties, we use the probabilistic extension of the weighted metrics temporal logics (PWMTL) as supported by UPPAAL SMC [26]. A valid PWMTL property in UPPAAL SMC is as follows:

$$\psi \;::=\; \mathbb{P}(F_{C \leq c}\varphi) \bowtie p \mid \mathbb{P}(G_{C \leq c}\varphi) \bowtie p \tag{2.4}$$

where C is the observer clock of the automaton under analysis, φ is a state-property with respect to the automaton, $\bowtie \in\; \{<, \leq, =, \geq, >\}$ and $p \in [0, 1]$.

2.4 Satisfiability Modulo Theories (SMT) and Z3

The problem of determining whether a formula expressing constraints (equality, inequality, etc.) has a solution is called *constraint satisfiability* problem. The most well-known constraint satisfiability problem is the propositional satisfaction *SAT*, where the problem is to decide if a formula over Boolean variables, formed using logical connectives can be made `true` by choosing `false/true` values of the constituent variables.

To express our constraints, in this thesis we use first-order logic formulas. A first-order logic formula is a logical formula formed using logical connectives, variables, quantifiers and function and predicate symbols. A solution for first-order logic formulas is a *model*, which in fact is an interpretation of the variable, function and predicate symbols that makes the given formula `true`. Additionally, the formulas that we use contain arithmetic operators such as $\{<, \leq, =, +, -, *\}$. For checking satisfiability of such formulas we use Satisfiability Modulo Theories (SMT) [12], which is basically an extension of the classical SAT problem over first-order logic formulas where the interpretation of some symbols is constrained by a background theory.

It is a well-known fact that the decidability for SAT is NP-complete and that the first-order logic is undecidable in general (under all possible interpretations), thus it is not feasible to develop a procedure that can solve an arbitrary SMT problem. To be able to use SMT solving in practice, most of the decision procedures used today focus on realistic examples and provide means for efficiently solving problems that occur in practice. The basic assumption for such procedures is that the satisfaction of formulas produced by verification and analysis tools is due to a small fraction of the formula, while the rest is deemed irrelevant. In recent years, thanks to advances in core algorithms, optimizations of data structures and heuristics, there is a tremendous progress in problems that can be solved using SAT/SMT procedures. In addition, a significant role in the advancement is played by the increasingly mature state-of-the-art tools.

In our work, we use Z3 [13], which is a state-of-the-art SMT solver and theorem prover developed and maintained by Microsoft RiSE group. The advantage of Z3 is that it has a stable group of developers who maintain the tool, as well as a broad academic community that is actively using it. The input of the tool is a set of *assertions* that can be either declarations or formulas. Originally, the assertions are specified using the SMT-LIB language [31]. Additionally, Z3 provides a number of application programmable interfaces (APIs) for specifying assertions using common programming languages such as `C#, Python, Java,` etc. Declarations in Z3 can be either *constants* or *functions*. In fact, in Z3 everything is expressed as functions, with constants being functions with arity 0. The types in Z3 are called *sorts* with the following provided by the tool: `Int, Real, Bool` and `Function`. The set of supported types can be extended with user-defined types. Z3 supports two types of quantifiers: universal quantifier (`ForAll`) and existential quantifier (`Exists`). For optimization of the decision procedure, the tool uses a number of tactics.

Z3 uses an internal stack where it stores the set of formulas whose satisfiability is to be checked. The command `assert` adds a new formula to the stack. The SMT decision procedure is invoked by executing the command `check-sat`, which checks where there is a solution for the conjunction of all the assertions on the stack. If the set of assertions is satisfiable, the Z3 tool returns the result `SAT`, which can be accompanied by the model that contains the witness assignment of the variables. The model is generated using the command `get-model`. In the opposite case, that is, when the set of assertions on the stack is not satisfied, the tool returns `UNSAT`, together with a *minimal* set of inconsistent assertions.

2.5 Specification Patterns

One way of enabling practitioners who are not experts in formal techniques, to create formal system specifications, is to provide them with methods and tools for a structured and reusable style of specifying requirements, where the structures have precise semantics (a defined relationship between their syntax and the model of computation) such that the formal specification can be automatically extracted.

An interesting approach called the *specification patterns system* (SPS) [32] has been proposed to facilitate the formal specification of system properties for practitioners who are not experts in formal methods. The proposed approach is based on the assumption that systems' specifications are framed within reoccurring solutions, from which a set of patterns can be extracted and saved for future reuse. Each pattern is characterized by a behavior that it captures, and an extent of the program execution in which the behavior must hold. The patterns are expressed as a combination of literal and non-literal terminals. The non-literal terminals can be either boolean expressions that describe system properties, or integer values that capture timing aspects. The rest of the pattern is made of literal terminals, which cannot be changed.

The original SPS catalog proposed by Dwyer et al. [32] is compiled by analyzing more than 500 examples of property specifications for various systems. The catalog contains 13 qualitative patterns, which for easier navigation are divided into two categories: *order* and *occurrence*, expressed in various types of temporal logics. The occurrence category contains patterns that describe the occurrence of a given state/event in the system, while the patterns from the ordering category are used to capture the relative ordering of the occurrence of multiple events/states during system execution. The catalog also introduced six different scopes, given as following:

- *Globally*, the entire program execution;
- *Before Q*, before the first occurrence of the state/event Q);
- *After Q*, after the first occurrence of the state/event Q;
- *Between Q and R*, any part of the program execution between states/events Q and R;
- *After Q until R*, similar as Between Q and R, except that the execution continues even if the second state/event never occurs.

One of the limitations of the SPS catalog provided by Dwyer et al. [32] is that it does not contain patterns for specification of real-time properties. For that purpose, Konrad and Cheng introduced a new category of patterns, called *real-time*, suitable for specification of real-time systems. Consequently, the extended catalog of specification patterns is called *real-time specification pattern system* (RTSPS). In the same work, Konrad and Cheng additionally proposed the controlled natural language (CNL) representation on top of the formal notations to increase readability and accessibility of specifications to different stakeholders. For illustration, let us consider the following requirement:

Globally, it is always the case that if (ECU was started) holds, then (CMS sent valid signal to totalFuelLevel) holds after at most 2 seconds.

The given example presents a requirement from industrial operational system, expressed via SPS using the *Bounded Response* pattern with *Global* scope. The non-literal terminals are given in brackets.

The next major update of the initial SPS catalog has been proposed by Grunske [33], as a set of patterns for commonly used probabilistic properties of the system. The most comprehensive SPS catalog is compiled by Autili et al. [34] by aligning different existing catalogs and performing systematic gap analysis with the purpose of discovering missing patterns. All these approaches are backed by tool support, as a necessary aid for applying them on realistic systems [35, 36, 37, 38, 39].

2.6 Sanity Checking

The term sanity checking has been introduced by Kupferman [40] to denote the process of automatically establishing the quality of formal system specifications represented as a set of temporal formulas with respect to formally defined criteria. In the literature, there are a number of sanity checking approaches that use different sets of formal criteria to assess the quality of the system specification, such as: *vacuity* (checking whether one or more requirements can be implied by other requirements in the specification) [40, 41], *coverage* (how much of the models' behavior has been captured by its specification) [40] *consistency* (lack of contradicting formulas within a specification) [42, 43, 44] and *completeness* (distinguish desired system behavior from any other) [42, 45].

The sanity checking techniques are suitable for quantifying the relationship between the model and its specification or for assessing internal quality of the specification without requiring a system model. An example of the former application of sanity checking, is to quantify which portion of all the possible behaviors of a given model are covered by its specification, whereas the second case of sanity checking can be used to assess for example the internal consistency of the specification. Intuitively, the first type of sanity checking requires a system model, thus the name *model-based*, while the second type of sanity checking does not require a structural or functional model of the system, hence the name *model-free* sanity checking [14]. The benefits of the model-free sanity checking is the possibility to detect errors in the specifications in the early phases of development, thus preventing their propagation into the subsequent artifacts.

Most of the existing sanity checking approaches define sanity checking criteria in such a way that it can be automatically checked using model checking. Despite the benefit of exhaustive sanity checking, such approaches can suffer from number of limitations such as the state-space explosion, or a very long analysis time (for complex specifications). Therefore, for early assessments of the specifications' quality, complementary techniques such as SAT/SMT-based analysis could be beneficial.

Chapter 3

Research Method

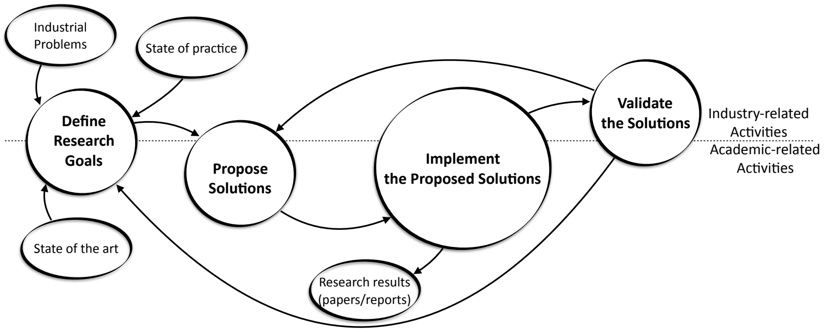

Industrial Problems
State of practice
Define Research Goals
Propose Solutions
Implement the Proposed Solutions
Validate the Solutions
Industry-related Activities
Academic-related Activities
State of the art
Research results (papers/reports)

Figure 3.1: Our research process.

Research methods represent concrete ways of solving a given research problem. According to the Meriam-Webster dictionary, research methodology is a *“body of methods, rules and postulates employed by a discipline”*. In this section we give an overview of the research process that we use to develop and address the research goals in this thesis.

The research process that we use in our research for this thesis is given in Figure 3.1. The research process represents an adaptation of the four steps research methodology [46] to our particular research context and includes the following steps: *problem formulation*, *propose solution*, *implement solution* and *validation*. The first step in the research methodology process is the research problem formulation. In order to define our research problem, first we perform identification of the real-world software engineering issues. For instance, in order to identify the problems that software engineers in the automotive domain are facing during the system specification phase, first, we conducted a series of interviews. Then we analyzed the quality of the existing artifacts (system specifications) and the features of the state-of-the-practice tools used for creating the artifacts. Once we have identified the research problem, we transfer it to the research setting by defining one or more research goals. To define our research goals, apply the *critical analysis of relevant literature and practice* method [47]. During this process, both the research problem and the research goals are refined and narrowed down.

Next, we propose a solution that addresses the identified research goals. For instance, we propose the SMT-based approach for the formal requirements consistency analysis described in Paper C [15] to address the needs for early debugging of system specifications in industrial settings. By critically assessing the existing approaches for consistency analysis, we discovered that there does not exist an approach for lightweight and fast consistency analysis in settings when no system model is available. The initially proposed solution is then refined in several iterations in a process that revolves around discussions, analysis and improvement until a stable form is reached which can be then implemented such that it can be applied in real-world scenarios. To assess the correctness of the proposed solution and to investigate whether it addresses the research goals in an adequate manner, we apply the following techniques: formal proof of correctness, case studies or proof-of-concept implementation [46]. The direct results of solving the solution are concertized/transferred into a set of research papers, presentations and tools.

During the last step, we perform the validation of our research results. The main goal of the validation phase is to check whether our research results are applicable to the real-world software engineering problems. This step is performed in close cooperation with industry, by applying the “proof by demonstration” [46] research method, where both researchers and engineers evaluate the research results. During the validation phase, the following aspects of the results are assessed: i) what is the scope of the proposed solution, that is, check whether the proposed solution fully or partially addresses the industrial problem; ii) scalability, to determine whether the proposed solution can be applied on the actual industrial systems, and iii) usability, that is, to what extent does the proposed solution support the transfer of the research results into industrial practice. For instance, the SMT-based consistency analysis methodology [15] (Paper C) is validated against an operational system (Fuel Level Display from Scania), while the approach for formal analysis of Simulink models using statistical model checking [48] (Paper D) is validated against a prototype implementation of an industrial Simulink model of the Brake-by-Wire system.

Chapter 4

Research Problem

In this section, we define the research problem and goals of the thesis. In Section 4.1 we describe the research problem, after which in Section 4.2 we define the overall research goal based on the actual state of practice and state of the art. To narrow the over-arching goal, we define in the same section research goals that help us to structure our research and relate the results to the problem.

4.1 Problem Definition

The predominant way of specifying requirements in the automotive domain is still as free-text, using mostly English language due to internationalization. The requirements are organized in system specification documents (system specifications), which are created and managed either by using general purpose text editors or specialized platforms such as IBM Rational Doors [49]. This practice of specifying requirements in natural language may lead to ambiguity in some of the requirements (due to the inherent ambiguity of the natural language). It also provides limited traceability and hinders the analyzability of requirements that are prone to inconsistencies for large specifications.

Currently, in industrial settings, manual peer-review is the most widespread technique for checking the correctness of the systems' specifications [5]. However, due to the increased size and complexity of the requirements specification documents, the cost for manual peer-review increases, and the quality assurance of the specifications becomes a challenge. A potential way to deal with the complexity of system specifications and to be able to reason about their quality, is to employ computer-aided analysis and verification enabled by formal techniques. Despite that the application of such techniques on some industrial use cases has been shown to be feasible [42, 50], their actual adoption in industry is limited by the difficulty of producing formal specifications and formal system models.

Even assuming that employing formal methods during industrial system development could be beneficial, it is unrealistic to expect that the industrial practitioners are able to effortlessly write formal specifications. Moreover, system specifications have to be accessible to a wide range of stakeholders, including managers, customer support and maintenance service people who must be able to read and interpret the requirements.

One way of enabling industrial practitioners to create formal system specifications is to provide them with methods and tools for a structured and reusable style of specifying requirements, where the structures have precise semantics (a defined relationship between their syntax and the model of computation), such that the formal specifications can be automatically extracted. As mentioned in Preliminaries (Chapter 2), an approach for pattern-based formal system specification [32], its real-time extension [7], and supporting tools [35, 36, 37, 38, 39] have been proposed to facilitate the formal specification of system properties for practitioners who are not experts in formal methods. What is missing from the existing endeavors is a study of the applicability of such approaches, and of the expressiveness of patterns in capturing industrial systems' requirements. The questions that await answers are: How can industrial system's requirements be formalized via specification patterns and how many of them can be captured? Are there types of requirements that do not fit the existing set of patterns? How does an engineer know what pattern to select? How to validate that the formalized behavior captures the engineer's intention?

Once the system specifications have been formally expressed, the next step is to ensure its consistency. This can be done via sanity checking. The majority of the existing work on automated sanity checking uses model checking as the formal technique to carry out the task [14, 40, 41, 42, 43, 44, 50]. The analysis might suffer from the well-known state-explosion problem, so its scalability is limited. Industry has an imperative need for early checking of the correctness of the system specifications for preventing potential specification errors from propagating to subsequent artifacts, including system models and the code. Consequently, a lightweight sanity-checking technique that can be applied in the early phases of system development might be beneficial.

Most industries, and especially the automotive one enjoys the benefits of the MBD paradigm, as models provide a good way of abstracting the engineering problem and documenting the design. Consequently, most of the solutions are implemented according to the MBD paradigm using different tools. In the automotive domain, Simulink is the “de-facto” standard for developing system models. For deployment, the engineers use specialized commercial tools to generate code directly from the Simulink models. Assuming that the code generation itself is correct, establishing the correctness of the behavioral Simulink models is of utmost importance as it has a direct impact on the correctness of the code deployed in vehicles.

4.2 Research Goals Definition

Based on the above discussed problems, we formulate the overall research goal of the thesis as follows:

Overall Research Goal. *Facilitate the quality increase of industrial embedded systems through automated formal techniques for: i) requirements specification and ii) analysis of design-time Simulink models.*

The overall goal sets to produce industrially-applicable formal methods and enable automation, by proposing an adequate tool support for helping the engineers who are not experts in formal methods, to formally specify and analyze different artifacts. Nevertheless, it is obvious that the overall goal is highly abstract and broad. In order to narrow down our goal and to be able to measure the contributions, we divide it into three sub-goals in the following.

We define the first subgoal as:

Subgoal 1. *Propose an engineer-friendly way for the formal specification of industrial embedded systems’ requirements.*

The result of addressing Subgoal 1 should be a concrete methodology, be it new or an adaptation of an existing one, which is expressive enough for formalizing the requirements of industrial embedded systems, out of which we chose automotive systems as representative. To increase the user friendliness of the approach, the methodology is to be accompanied by an adequate tool support.

Once the formal system specification is generated, the next question to answer is: how to determine if such a specification is qualitative, for instance, free of inconsistencies? For this purpose, we have formulated the second subgoal as:

Subgoal 2. *Enable early consistency checking of formalized requirements of industrial systems.*

The outcome of addressing Subgoal 2 should be an automated approach for consistency checking of the formalized system specifications in the early phases of the system development, when no structural or behavioral system model is available.

Having precisely specified and consistent system requirements is not enough to increase the quality of embedded systems. They need to be checked for fulfillment on development artifacts usually employed in the design of industrial embedded systems. Such artifacts are Simulink models used by engineers both for early simulations of the system's functions, but also for code generation. Therefore, our last subgoal targets the generation of formal system models corresponding to Simulink models, on which formal analysis techniques can be applied. Subgoal 3 is given below:

Subgoal 3. *Enable formal analysis of industrial systems models described in Simulink.*

The outcome of addressing Subgoal 3 should be an automated procedure for generating formal system model based on Simulink industrial models that can be formally analyzed by using state-of-the art tools and methods.

Chapter 5

Thesis Contributions

In this chapter, we give an overview of the research results and contributions that address the research goals defined in Section 4.2. The main contributions of the thesis are on three fronts: i) engineer-friendly formal requirements specification method and tool; ii) an SMT-based method for the automated consistency analysis of formalized system requirements specifications expressed as TCTL formulas, and iii) a method and tool for formal analysis of industrial system models described in Simulink using statistical model checking. Throughout the thesis, we focus our research on automotive systems, due to our cooperation with Swedish vehicle manufacturing companies.

5.1 Pattern-based Formal System Specification

Our first contribution is an engineering-friendly way for formal system requirements specification. It can be divided into two parts: in the first part we assess the suitability of a candidate approach for creating formal system specifications for practitioners who are not experts in formal methods, namely the RTSPS introduced in Section 2.5, while in the second part, we propose a tool called SeSAMM Specifier such that the methodology can be potentially adopted in the industrial development process.

5.1.1 Reassessing the Pattern-based Approach for Formal System Specification in the Automotive Domain

We select the pattern-based approach and the RTSPS catalog as a candidate approach for creating formal system specifications, based on the results of a previous study [9]. In the given paper, the authors show that specification patterns and the RTSPS catalog are expressive enough to formalize requirements of systems produced by a single supplier in the automotive domain. In addition, their results show that a subset of patterns is enough to express the majority of the requirements. The limitation of their study is the fact that the considered requirements belong to systems from a single vendor, thus generalizing the results to the whole domain remains an open question.

In order to gain further understanding of the benefits, limitations and challenges encountered when formalizing requirements in a realistic setup, we perform a case study in collaboration with Scania, Sweden. In our work, we do not attempt to remove the above-mentioned limitation, but just to investigate further whether we can enlarge the "borders" of the previous results of Post et al. [9]. The goal of the case study is to take a subset of non-structured functional requirements of the E/E systems written in natural language, and formalize them using the RTSPS catalog of the specification patterns. The case study is performed in several steps, as follows: i) system specifications gathering, ii) formalization using specification patterns, and iii) analysis of the results.

For collecting the requirements for the study, we contacted a number of engineers from the company and asked them to provide requirements documents exemplifying their usual work. In response, we received four documents with a number of unstructured functional requirements written in natural language. Once we obtained the documents, we extracted the requirements in a separate data sheet, in order to make sure that only the requirements were assessed, and not context information or other meta-data related to them. As a result, we compiled a list of one hundred requirements. By doing this, we had an indirect way of measuring the quality of the system specifications, meaning that if a requirement cannot be patterned without knowing other information appearing in the document, it will most likely not be prone to an automated formalization.

In the second phase, we formalized each of the extracted requirements by employing RTSPS. The formalization was performed by expressing each of the requirements via one of the available patterns, such that the formalized representation could be automatically extracted. Since the goal of the exercise was to assess the expressiveness and adequacy of the patterns, there was no need to proceed further and obtain the expressions in temporal logics. For the

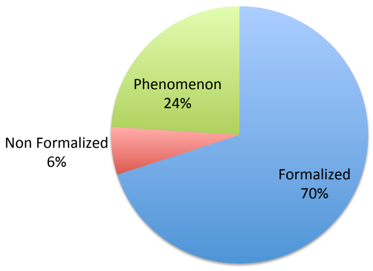


Figure 5.1: Formalization results.

purpose of our research, we claim that a requirement is formalizable if there exists a pattern that captures its semantics.

The results of our case study show that the specification patterns are expressive enough to formalize the requirements included in the case study. As presented in Figure 5.1, 70% of all the requirements included in the case study could be expressed using RTSPS patterns. The remaining portion of requirements is divided into two categories, as follows: 6% of all the requirements are categorized as not formalizable, meaning that the described behavior cannot be expressed using patterns. According to our experience, high complexity, high level of ambiguity and lack of information are the main reasons impeding the patterning of such requirements. The third class of requirements are the so-called *phenomenon*, which refer to requirements that do not express system behavior, but give information about data or the system configuration. These requirements cannot be mapped into patterns, but can be expressed by means of non-literal terminals.

In total, in our case study we use 7 patterns to capture the behavior of the formalized requirements (see Figure 5.2). This finding is aligned with the claim of Post et al. [9], that a small subset of patterns is enough to express most of the requirements of an embedded automotive system. Despite the positive results, our case study reveals some challenges during the formalization of the requirements. First, the formalization of system specifications using patterns has a learning curve. During the formalization, we have noticed that the speed of formalization of requirements increases with experience. However, finding ways to soften the learning curve is desirable. Second, and maybe a more important finding is that it is hard to validate whether the requirements

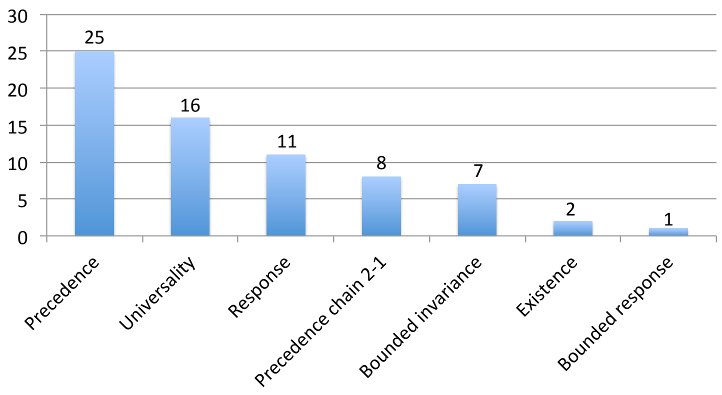


Figure 5.2: Pattern frequency.

expressed using specification patterns capture the intention of the engineers accurately. The presented contributions are published as Paper A [8] and address Subgoal 1.

5.1.2 Integrating the Pattern-based System Specification in an Industrial Toolchain

Driven by the fact that the specification patterns are expressive enough to formalize most of the requirements in the automotive domain, as our second contribution we propose a tool called SeSAMM Specifier, which enables engineers to specify their systems using the specification patterns. The tool has been designed to satisfy demands such as flexibility and extensiveness.

On one hand, flexibility means that the tool can be integrated into existing toolchain(s) used in industrial settings instead of being developed as a standalone tool. By integration, the tool can benefit from existing features of other tools already existing in industry. On the other hand, extensiveness means that the tool must not be bound to a predefined catalog of patterns, instead it can be extended and adapted further depending on the context. In order to explain how the SeSAMM Specifier tool supports these features, we give an overview of its architecture as presented in Figure 5.3. The architecture of the tool consists of three data sources for storing information, and two user interfaces (UIs) that are used for managing it. The three data sources are given as follows: *Domain Ontology* stores information about the various concepts of

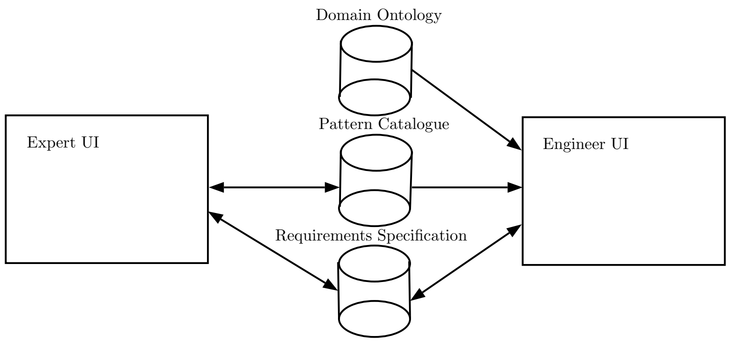


Figure 5.3: SESAMM Specifier Architecture

the system architecture, such as function and signal names, software variables, etc.; *Pattern Catalog* contains all the patterns that can be used to formally specify requirements, whereas the *Requirements Specification* data source contains all the requirements expressed via patterns.

The information in the data sources is manged by the two interfaces of the tool. The arrows in the architectural design represent the data flow and the permissions in the system. For example, the information stored in the *Domain Ontology* can be accesses only through the Engineer UI that has permission only to read the stored data, but not to change it. In contrast, the information about the requirements expressed via patterns stored in the *Requirements Specification* data source can be accessed and modified by both user interfaces of the tool.

The tool is accessible to two types of users through its interfaces: the *Expert UI* enables experts in formal methods to manage the catalog of the specification patterns of the tool, while the *Engineer UI* is intended to help the engineers to create formal system specifications. The use and the functionalities of the tools will be explained via the two work flows as presented in Figure 5.4. The detailed explanation of both interfaces is given in our work [10] or in Chapter 9.

The work flow for the experts denotes the process of creating new, or modifying existing patterns in the pattern catalog stored in the *Pattern Catalog* data source. In the first step (Step 1 in Figure 5.4), the expert user selects to create a new pattern or modify an existing one. Once all the steps of the work flow are completed (Steps 1, 2 and 3 in Figure 5.4), the pattern is saved into the *Pattern Catalog* data source.

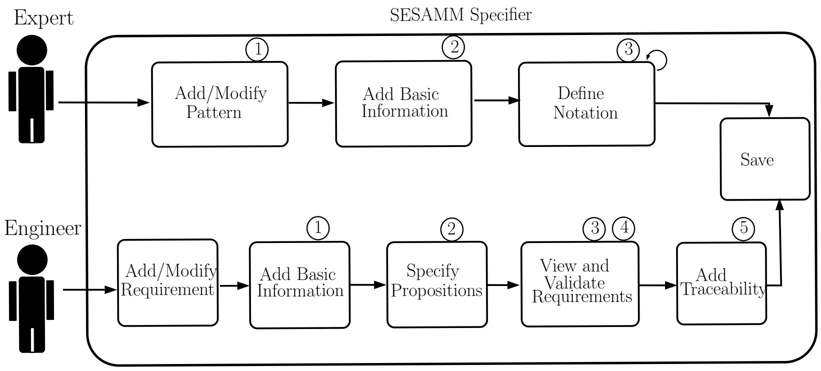


Figure 5.4: Expert and Engineer work flows of SeSAMM Specifier

The other work flow in the system describes how the tool and the Engineering UI can be used by the engineers to create formal system requirements specifications expressed using patterns. The work flow in initialized by the engineer by selecting the option to either create a new requirement or to modify an existing one (Step 1 in Figure 5.4). Additionally, the interface incorporates a mechanism for validation of the formalized behavior expressed via patterns (Steps 3 and 4 in Figure 5.4). For that purpose, the tool incorporates three different visual notations for visualizing the formalized requirements behavior. The presented contributions are published as Paper B [10] and address Subgoal 1.

5.2 Automated Consistency Analysis of Formalized System Specifications

Sanity checking of formalized requirements prior to full formal analysis can be used to detect errors that might decrease the subsequent verification effort, while also increasing the quality of the requirements of a system that can later be reused or evolved.

Therefore, our next contribution is an automated SMT-based consistency analysis methodology for the formal system specifications expressed as a set of TCTL formulas obtained by applying the specification patterns. The proposed methodology belongs to the class of model-free sanity checking techniques, meaning that it does not require any type of structural or functional model of the system, thus it can be applied at early phases of system development. The methodology includes several steps, as follows: first, we recall the formal definition of a systems' requirements specification inconsistency, which we are in fact interested in detecting. Then, we transform the TCTL patterns into first-order logic (FOL) formulas by instantiating the semantics of the path and temporal operators. Next, the FOL formulas are transformed into Z3 assertions, which are later optimized for analyzability via abstraction rules. Lastly, for validation, we apply our methodology on a set of requirements from an operational industrial system, that is the Fuel Level Display function, implemented in Scania vehicles.

The definition of the logical inconsistency of formalized system requirements specifications that we rely on is given below:

Definition 1 (Inconsistent specification)**.** *Let* $\Phi = \{\varphi_1, \varphi_2, ..., \varphi_n\}$ *denote the system requirements specification, where each of the formulas* $(\varphi_1, \varphi_2, ..., \varphi_n)$ *encodes a requirement. We say that the set* Φ *is inconsistent if the following implication is satisfied:* $\varphi_1 \wedge \varphi_2 \wedge ... \wedge \varphi_n \implies False$.

From the given definition, it follows that a system specification is *inconsistent* if there does not exist valuation of all terms of the conjunct, such that the latter is true. To disprove the inconsistency, it is enough to provide a witness set of valuations of the variables, which satisfies the conjunction of all the formulas. Checking whether there exists an interpretation that satisfies the Boolean conjunction given above represents a classical Boolean satisfiability (SAT) problem. Since in our work, the FOL formulas contain arithmetic operations such as $\{+, -, <, \leq, \geq, >\}$ we use SMT.

The structured derivation proofs for transforming TCTL into FOL formulas are given in Paper C [15] (also in Section 10.5 of Chapter 10). Once the set of FOL formulas is generated, next we encode them as a set of Z3 assertions, which can be checked for consistency using the Z3 tool. The encoding process and the abstraction rules that we use to generate analyzable sets of Z3 assertions are as follows:

R1: Directly map the FOL constructs into Z3 syntax elements. For instance, mapping the quantifiers ($\forall$ into `ForAll`, $\exists$ into `Exists`, etc.), modeling port values as functions of time, etc.

R2: Reduce complexity by abstraction: (a) eliminate path (σ) universal quantifiers, and (b) collect location (i) and time in location (d) into a tuple position (pos).

R3: Abstract the universally quantified $pos = (i, d)$ to the universally quantified $pos.d$.

The process of applying the rules R1, R2 and R3 on the set of patterns (P_i) used for formalizing the system specifications can be illustrated as follows:

$$P_i \xrightarrow{R1,R2} P_i' \xrightarrow{R3} P_{i\,Z3},\ i \in [1, n]$$

By applying rule R1 we generate a set of Z3 assertions where each assertion corresponds to one requirement in the specification. If we recall that the TCTL formulas are interpreted over a branching model, the resulting FOL formulas are quantified over three variables as follows: execution paths, locations and clock valuations. Also, we know that the number of quantifiers and quantified variables has a negative impact on the decidability of the SMT procedure. To reduce the complexity of the assertions, we apply rules R2 and R3. As a result, the path and location quantified variables are eliminated from the assertions, meaning that the newly obtained set of formulas are quantified over time only.

Despite the application of the complexity reduction technique, the set of Z3 assertions could not be analyzed for consistency, as the procedure did not terminate with a result. The cause for this are additional nested quantifiers arising from use of the weak-until operator in some of the patterns, for modeling the sporadic occurrence of some event. Since the occurrence of all sporadic events in the system is bounded from above, we apply a mitigation strategy according to which all sporadic events are transformed into periodic events with the period equal to the upper bound of the allowed interval. Even though such a model of the requirements is pessimistic, our abstraction is conservative (the satisfaction of the original formula follows from the satisfaction of the abstracted one).

For validation, we apply the proposed approach on a set of requirements of Scania's Fuel Level Display (FLD) system. Using our approach, we manage to prove the consistency of 24 FLD requirements within seconds. Since the FLD is an operational system, such result is expected.

The presented contributions are published as Paper C [15] and address Subgoal 2.

5.3 Statistical Analysis of Simulink Models

As a last contribution of the thesis, we propose an approach for the formal analysis of industrial embedded system models described in Simulink using UPPAAL SMC. To enable the statistical model checking of Simulink models, we propose the following: i) first, we categorize Simulink blocks according to their execution and introduce patterns for transforming atomic Simulink blocks into networks of STA, and a flattening algorithm for composite Simulink blocks; ii) second, we provide a tool called SIMPPAAL that completely automates the procedure and iii) third, we validate our approach by applying the SIMPPAAL tool on the Brake-by-Wire Simulink model from Volvo GTT, Sweden. In the following, we present each of the aspects of the contribution in more details. The contributions listed below are published as Paper D [48], and address Subgoal 3.

5.3.1 Transformation of Simulink Blocks into Stochastic Timed Automata

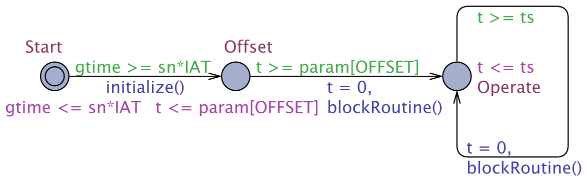
We provide a pattern-based approach for transforming sets of atomic Simulink blocks into networks of stochastic timed automata (STA). To achieve this, we classify the atomic Simulink blocks into two categories: continuous and discrete. A block is considered to be continuous if it updates its output continuously (at infinitely small intervals). In contrast, the discrete Simulink blocks compute their outputs at specific time intervals during the simulation. The distance between two periods at which a discrete block produces an output is called *sample time*. To be able to transform Simulink atomic blocks into corresponding STA in sound manner, we first provide a formal definition of the syntax and semantics of an atomic Simulink block.

Definition 2 (Formal Definition of an atomic Simulink Block)**.** *An atomic Simulink block can be formally defined as the following tuple:*

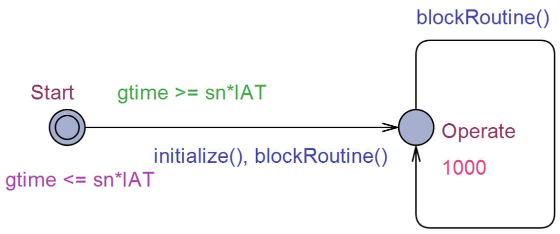
$$B = \langle sn, V_{in}, V_{out}, V_D, \Delta, Init, blockRoutine \rangle \tag{5.1}$$

where: sn is the execution order number of the block inside the respective Simulink model, V_{in}, V_{out} and V_D denote the sets of input, output and state variables of the block, respectively; Δ is the time stamp composed of the sample time (t_s) and the offset parameters (ts, offset $\in \mathbb{R}_{\geq 0}$), at which the block updates its outputs, $Init()$ is the initialization function and $blockRoutine()$ is a function that maps the inputs and state variables into outputs.

The atomic Simulink blocks are transformed into networks of STA using a pattern-based approach. We propose one pattern for each type of Simulink block, discrete and continuous, as given in Figures 5.5a and 5.5b. The STA pattern for discrete blocks (Figure 5.5a) consists of three locations: `Start`,



(a) Discrete block



(b) Continuous block

Figure 5.5: Transformation patterns for atomic Simulink blocks

`Offset` and `Operate`. The pattern captures the behavior of a discrete block during simulation time as follows: the automaton initially is in the `Start` location. It stays there until it is released, that is, waits for its turn for execution. The time at which the block is released is calculated based on the execution number of the block in the Simulink model (`sn`) and the inter-arrival time of the signals in the model (`IAT`). The combination of these two parameters is given as a constraint over the global clock variable (`gtime`). After the block is triggered for execution, it can delay the computation of the first output for a specific time interval denoted as offset, modeled via the `Offset` location in the STA template. After the offset elapses, the automaton proceeds into operation mode by going into the `Operate` location of the STA. On the edge between the `Offset` and `Operate` the automaton produces the first output and resets the local clock (`t`). The `Operate` location has only one outgoing edge executed at every sample time (`ts`) to update the output.

The STA template for a continuous Simulink block is similar to the discrete one with the following differences. Since the continuous Simulink blocks are not allowed to delay the first execution, the `Offset` location has been removed from the template. Secondly, in order to approximate the continuous update of the output, instead of an invariant, the `Operate` location is decorated with a *rate of exponential* parameter, which is used in the density function that computes the probability of the automaton to stay in that location at each simulation step. The higher value for the parameter, the less the probability of the automaton to stay in the given location. The actual function of each Simulink block is encoded in C, as blockRoutine(). To ensure the correctness of each block routine, we apply the program verifier Dafny [51], which uses the assume-guarantee paradigm to check the correctness of the code.

The composite Simulink blocks are used for creating a hierarchical structure of the model. Unlike the atomic ones, the composite blocks do not have a predefined behavior as they are realized through a set of atomic Simulink blocks. To eliminate the hierarchy, we propose a flattening algorithm that replaces the composite Simulink blocks with a set of atomic Simulink blocks, which when executed in the correct order imposed by their respective execution order number, exhibit exactly the same behavior as the composite block. While the set of blocks is always given inside the model, the challenge is to ensure their correct execution, such that the original behavior modeled using the composite Simulink block is preserved in the formal model. To assign the correct execution order of the atomic blocks inside a given composite Simulink block we apply the proposed flattening algorithm. For detailed description of the flattening procedure, we refer out reader to our work [16] or Chapter 11 of this thesis.

A Simulink model is then a composition of Simulink blocks as follows:

$$S = B_1 \otimes B_2 \otimes B_3 \cdots \otimes B_n \tag{5.2}$$

where: $sn^s = \bigcup_{i=1}^{n} sn^i$ is an ordered list of execution, in which $\forall(i,j).(i < j \Rightarrow s_i < s_j)$, $V_{in}^S = \bigcup_{i=1}^{n} V_{in}^i$ is the set of input variables, $V_{out}^S = \bigcup_{i=1}^{n} V_{out}^i$ is the set of output variables, $V_D^S = \bigcup_{i=1}^{n} V_D^i$ is the set of internal state variables, $\Delta_S = \bigcup_{i=1}^{n} \Delta^i$ is the set of time points at which the respective data and output variables are updated, and $(Init; blockRoutine)_S \triangleq$ $(Init_1; blockRoutine_1)|_{=\Delta_1}; (Init_1; blockRoutine_2)|_{=\Delta_2};$ $\ldots\ ; (Init_n; blockRoutine_n)|_{=\Delta_n}$ is an ordered list of pairs of (Init, blockRoutine), which are executed atomically at given times Δ_i.

5.3.2 SIMPPAAL Tool

To automate the complete procedure of transforming Simulink models into networks of STA suitable for analysis using UPPAAL SMC, we propose SIMPPAAL tool whose function is as follows. The tool takes as input a Simulink model and the list of execution order numbers of each Simulink block inside the model, be it atomic or composite, and it automatically generates the network of STA that UPPAAL SMC can be fed with, for formal analysis. Additionally, for each of the transformed block, the tool automatically generates a Dafny encoding of the block routine that can be used for proving the correctness of the computational routine.

The tool has been designed following the modular architecture principle, in order to reduce the implementation complexity and to have dedicated parts of the tool responsible for a specific set of tasks. Each of the components in the tool is called *module*. The first prototype implementation of SIMPPAAL as included in this thesis, is based on three modules given as follows: a module for flattening the model by assigning an execution order number for each atomic Simulink block relative to the root Simulink model; a module for parsing and manipulating Simulink model files, and a module for transforming atomic Simulink blocks into a network of STA. For details about the implementation of each module and the complete work flow of the tool we refer our reader to earlier work [48] or Chapter 11 of this thesis.

5.3.3 Validation of SIMPPAAL on a Brake-by-Wire System

To validate our approach and to assess the effectiveness of the SIMPPAAL tool, we applied it on a Brake-by-Wire Simulink model from Volvo Group Truck Technology. Brake-by-Wire is a prototype implementation of a braking function equipped with an Anti-Lock Braking System (ABS) functionality, without any mechanical connection between the brake pedal, and the brake actuators. The sensor reads the position of the break pedal which is used by the algorithm to determine how much breaking torque to apply on each of the wheels of the vehicle.

The Simulink model of the Brake-by-Wire function consists of 320 atomic Simulink blocks placed inside a number of Simulink model files and libraries. The transformation procedure, including the flattening plus the transformation of the atomic Simulink blocks takes about 20 seconds to complete. The generated network of STA consists of 149 STA and 12 constants represented as data variables in UPPAAL. Of all the transformed blocks, 133 are continuous and

Table 5.1 SMC analysis results for BBW.

Req.	Query	Result	Runs
$R1_{BBW}$	$Pr[<= 200](<> Monitor.End)$	Pr $\in [0.998, 1]$ with confidence 0.999	3797
$R2_{BBW}$	$Pr[<= 200](<> Monitor1.End\ and\ (Monitor1.x - Monitor2.x <= 20\ and\ Monitor1.x - Monitor2.x >= -20))$	Pr $\in [0.990014, 1]$ with confidence 0.995	597
$R3_{BBW}$	$Pr[<= 200](<> pedal\_map\_161\_signal <= 100)$	Pr $\in [0.995002, 1]$ with confidence 0.9975	1334
$R4_{BBW}$	$Pr[<= 20](<> Monitor.End\ and\ Monitor.s > 20\ and\ Monitor.torque == 0)$	Pr $\in [0.902606, 1]$ with confidence 0.95	36

16 are discrete.

The obtained formal model is then subjected to formal analysis with respect to both timing and functional properties. The properties that we have analyzed, the probability for their satisfaction and the time for verification are given in Table 5.1.

Chapter 6

Related Work

In this section, we present an overview of research endeavors related to the three main research problems, that are considered in our work, including: formal requirements specification via specification patterns, automated requirements' consistency analysis, and formal analysis of behavioral models of industrial embedded systems specified with Simulink.

Specification Patterns. The work on specification patterns has two major directions. The first direction is towards enriching the specification pattern catalog by identifying new patterns, whereas the second one is more focused towards bringing the specification patterns closer to the practitioners. While there is a large body of work in academia that focuses on improving the specification pattern catalog [7, 32, 33, 34], there are not many studies that test the applicability of the patterns in real-world scenarios. In our current work, the main focus is on the usability side of the patterns as a way for formalizing system specifications. In our work, we focus on two aspects of the usability of the specification patterns: i) perform a case study to test the applicability of the patterns and ii) provide an adequate tool support for addressing the needs of the practitioners who are not experts in formal methods. Our contributions are two-fold: first, the results of our case study strengthen the claims of the earlier formalization attempts in industrial settings [9], and second, we propose a tool called SeSAMM Specifier, which not only enables various practitioners in formal specification of their systems, but also provides means for validation of the formally specified behavior through a number different visual formalisms.

There is a number of academic tools that have been developed to facilitate the formal requirements specification using the specification patterns. The PROPerty ELucidation system (PROPEL) [35] and Property Specification (Prospect) [37] support requirements specification using disciplined natural language and finite-state automata (FSA), relying solely on state-based notations to formally represent requirements behavior. The Property ASSistant (PASS) [38] tool has been implemented to facilitate the specification of event-based systems. Other tools such as CHARMY [36] support the design and validation of architectural specifications captured in UML. The PSPWizard tool [34] is very similar to PROPEL and Prospect, yet it exhibits an advantage by providing a more comprehensive catalog of specification patterns.

If compared to the tools listed above, our SeSAMM Specifier tool differs in several ways. First, all previously mentioned tools rely on a specific pattern catalog. In contrast, SeSAMM Specifier, does not include a predefined catalog, rather it is designed and developed to be a general tool built on top of the pattern-based approach [10]. Consequently, the SeSAMM Specifier has more expressive power than the rest of the tools. Secondly, the SeSAMM Specifier provides a mechanism for giving visual feedback to the users. Although not unique with respect to this feature, our tool provides more options than most tools, and we are currently working on extending the set of available graphical notations for visualizing behavior. What is truly unique about the SeSAMM Specifier is the fact that it has been developed with practitioners in the loop. This has resulted in a tool that engineers can associate themselves with, which can have a positive effect on the adoption of the tool in their everyday work.

Formal Analysis of Requirements Specification. In our work we have proposed an approach for automated consistency analysis using Z3 SMT solver. In the following, we list some of the related approaches for automated consistency analysis of requirements specified as temporal formulas.

Barnat et al. [14] propose a model-free sanity checking procedure for consistency analysis of system requirements specification in Linear Temporal Logic (LTL) [52] by means of model checking. The approach has later been extended [50] to support generation of a minimal inconsistent set of requirements. Despite the exhaustiveness, the approach suffers from the inherent complexity of transforming the LTL formulas into automata, especially for complex systems, potentially making it unusable in industrial settings. A similar approach for consistency checking of requirements specified in LTL is proposed by Ellen et al. [53]. The authors introduce a definition for the so-called existential consistency, that is, the existence of at least one system run that satisfies the complete set of requirements. Similar to what we propose, the analysis procedure has been integrated into an industrially relevant tool, aiming at industrial application. The work by Post et al. [44] defines the notion of rt-(in)consistency of real-time requirements. The definition covers cases where the requirements in the systems requirements specification can be inconsistent due to timing constrains. The checking for rt-inconsistency is reduced to model checking, where a deadlock situation implies inconsistency of requirements.

Despite the exhaustiveness of the consistency checking approaches mentioned above, all of them suffer from two major limitations: the time required for generating the requirements model as well as the time for analysis that grows exponentially with the number of requirements that should be analyzed. In comparison, our approach copes well with the time for generating the model for analysis. This is due to the fact that the input model for analysis is a set of Z3 assertions, which can be generated in negligible time as compared to building automata models as required by some of the approaches [14, 44]. However, Z3 is not as expressive, and only accepts abstractions of our actual specification model in order to be able to return a result. Therefore, we cannot detect complete temporal inconsistencies. Another difference is the exhaustiveness of consistency analysis. Model checking is exhaustive, whereas we sacrifice this for avoiding the potential state space explosion. The model-checking-based consistency analysis [14, 44] can guarantee the absence of any inconsistencies in the system, while our approach (similarly to [53]) is suitable for checking whether the system specification is realizable as such, that is, if there exists at least one system run that satisfies the conjunction of all the requirements in the specification. Mahmud et al. [54] propose a more high-level consistency analysis approach applied on Boolean level, without taking the temporal aspects into consideration approach. In cases where inconsistencies are detected, all of the approaches (including ours) are able to generate the minimal inconsistent set, that is, the minimal set of inconsistent requirements. The above listed characteristics make our approach suitable to be used in the early phases of system requirements specification, where a more lightweight and considerably faster procedure might be more suited.

The quality of the system requirements specifications can be assess even when they are expressed informally, that is, in natural language. One such approach is proposed by Fabbrini et al. [55, 56], which assesses the quality of the requirements specifications using natural language processing techniques with respect to a quality model. The approach is complemented with an adequate tool support for automating the analysis [57]. Recent study [58] shows that such techniques can be applied large sets of industrial requirements to help the engineers to prioritize the requirements to be manually analyzed for defects.

Formal Verification of Simulink Models. The existing approaches for verification of complex Simulink models can be divided into three broad categories: i) generation and abstraction of simulation traces, ii) abstraction of blocks into contracts/theories and their formal analysis, and iii) model-to-model (M2M) transformation followed by model checking.

The approaches from the first category abstract Simulink simulation traces into state machines representing the system's behavior, which is then subjected to model checking [59]. For instance, PlasmaLab follows this approach and uses SMC for model checking [60]. The limitation of the approach, however, is the lack of exhaustiveness when it comes to generation of the simulation traces, as additional simulation traces may not be easy to obtain. Moreover, since it is based on system traces, it is not adequate for verification of extra-functional properties, at least without further changes on the initial model. On the positive side, the approach is generic for any kind of Simulink diagram and does not require adding more computation if new blocks are considered.

The approaches from the second category use a two step process for verification. First, the system designer "lifts" the specification of each block using some type of logical language. Second, the whole specification is composed and fed into some kind of analysis engine. In [61], Ferrante et al. use contract-based theory in order to model the block specification, and rely on a combination of SAT solvers and the NuSMV model checker for analysis. In some other approaches [62], PVS is used for writing the specification, and its theorem-proving capabilities are employed for the Simulink model verification. A limitation of this strategy is that it is not generate the formal model automatically, like SIMPPAAL does.

The approaches from the third category are intended for high level of automation. They usually rely on automated M2M transformation from Simulink into an automata language that can be verified with model checking, and have received much more attention in the literature. The approach proposed by Barnat et al. [63] transforms the Simulink models into the language of an LTL explicit model checker called DiViNE. The approach is applicable for discrete blocks only, however they show it suitable for the aeronautics industry. Similarly, the approach by Meenakshi et al. [64] proposes transforming of discrete blocks into NuSMV. In contrast, Agrawal et al. [65] addresses the transformation of Simulink into networks of automata, without providing concrete means for formal verification. The work by Miller [66] provides a translation from Simulink to Lustre and enables formal verification with a constellation of model checkers and provers. The transformation of StateFlow design elements has been addressed in research endeavors by Manamcheri [67] and

Jiang et al. [68] in which they propose transformation frameworks from StateFlow/Simulink into timed and hybrid automata, respectively, yet not considering other types of Simulink blocks.

In general, the solutions available for automated M2M transformation of Simulink are quite restrictive with respect to the number of block types supported, as typically only discrete blocks or only StateFlow diagrams are addressed. Also, they have been applied only on academic or middle-size Simulink diagrams, such as the engine control system appearing in the Simulink distribution, which raises concerns about the scalability of the approaches. The only exception is the approach by Zuliani et al. [69], which uses Bayesian statistical model checking for analyzing the specification, thus it can scale better. Despite that, the approach has not been applied on Simulink models of industrial systems, and it seems to have practical limitations such as not accepting multi-file Simulink models.

Our proposed framework relies on M2M transformation and statistical model checking, but it goes beyond the current state of the art by reducing the modeling effort as M2M transformation is based on templates and fully automated. Additionally, we are supporting a larger number of Simulink blocks (although some of them are still under development), whereas for validation we use real-size industrial examples. We also aim at generating a formal model as close to the Simulink model as possible, so we encode the functions of blocks not as differential equations, when the case, but as C routines that are faithful to the Simulink modeling. To the best of our knowledge, it is also the only approach that introduces formal verification at the code level of the transformation, using Dafny [51].

Chapter 7

Conclusions and Future Work

In this thesis we have established the foundations of a framework for increasing the quality of embedded systems through automated specification and analysis of the requirements and behavioral models created at different development phases. To fulfill our goal, we have presented a number of contributions on several fronts. First, we have shown that the specification patterns are expressive enough for formalizing industrial requirements, particularly in the automotive domain. Despite being expressive enough, our results show that the patterns still have some limitations such as steep learning curve, problem of validation of formalized behavior and lack of tool support tailored for industrial needs. Some of the open problems in this area have been addressed by our second major contribution, which is a prototype implementation of the SeSAMM Specifier tool [10]. The tool has been designed and implemented with close cooperation with the engineers in the loop, in order to increase the chances that the tool addresses their needs and become integrated into the development process. The tool provides features such as managing the specification pattern catalog, such that the expressiveness is no longer a limitation, an interface for non-expert practitioners to formally specify their systems, and a validation mechanism based on visualization of the formalized behavior, to check whether the specification captures the actual intentions of the engineers.

As our next contribution, we have proposed an SMT-based consistency analysis technique suitable for consistency checking of industrial systems' requirements early in the development process, when no behavioral or structural model of the system exists. To enable this, we have provided a formal definition of consistency and an analysis procedure for checking it. For analysis we rely on existing state-of-the-art SMT solver and theorem prover Z3 [13]. For validation, the proposed approach has been applied on a prototype implementation of a running truck software function, namely Fuel Level Display from Scania. The results from the validation show that our technique is suitable for detecting inconsistencies in the requirements that describe mutually exclusive events/states. Additionally, the execution time for the SMT-based procedure is short as it terminates in a matter of seconds, compared to model-checking based consistency analysis techniques, which for systems of similar size are reported to require several hours.

Our last contribution is an approach for the formal analysis of behavioral models specified using Simulink. To enable this, first we classify the most frequently used Simulink blocks obtained by analyzing various systems models from Scania and Volvo GTT, and second, we provide a template-based transformation for each of the identified classes into timed automata with stochastic semantics suitable for checking using UPPAAL SMC tool [17]. To validate our approach we apply the transformation on one prototype industrial system, that is, the BBW of Volvo GTT.

There are several directions for future research endeavors to fill in the voids in the current state of the framework. The most immediate future work is on improving the SIMPPAAL tool for generating the formal model. The improvement encompasses mostly technical improvements and adding new features to the tool. First, we want to refactor the code-base of the tool so that we can open source it completely and offer it to the community. Secondly, we plan to optimize the existing features of the tool in order to be able to apply it on more systems, possibly with higher complexity and from other vendors. Lastly, we intend to explore the possibility to apply other tools for formally analyzing Simulink models. One such example is to try to use the STORM probabilistic model [70] checker instead of UPPAAL SMC. Such transition would require developing a new formal model of the system and incorporating this into the SIMPPAAL platform. This will perhaps enhance the class of systems that we can tackle, and improve the scalability of the analysis.

Concerning the formal system specification using specification patterns, we plan to perform more empirical studies around formal requirements specification via the specification patterns in order to identify new gaps and to more accurately measure the benefits of using patterns in industry. Additionally, we aim at extending our automated requirements analysis methodology by: i) improving the scalability of the current consistency analysis technique and ii) including other criteria such as coverage, completeness, vacuity, etc. Finally, our last goal is to unify the current and future contributions into a complete framework for improving the quality of embedded systems and to validate it in industrial settings.

Bibliography

[1] Manfred Broy. Challenges in Automotive Software Engineering. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 33–42, New York, NY, USA, 2006. ACM.

[2] Manfred Broy, Ingolf H Kruger, Alexander Pretschner, and Christian Salzmann. Engineering automotive software. *Proceedings of the IEEE*, 95(2):356–373, 2007.

[3] Klaus Grimm. Software Technology in an Automotive Company: Major Challenges. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 498–503, Washington, DC, USA, 2003. IEEE Computer Society.

[4] Alexander Pretschner, Manfred Broy, Ingolf H. Kruger, and Thomas Stauner. Software Engineering for Automotive Systems: A Roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 55–71, Washington, DC, USA, 2007. IEEE Computer Society.

[5] Gursimran Singh Walia and Jeffrey C. Carver. A Systematic Literature Review to Identify and Classify Software Requirement Errors. *Inf. Softw. Technol.*, 51(7):1087–1109, July 2009.

[6] ISO/DIS 26262-1 - Road vehicles Functional safety Part 1 Glossary. Technical report, Geneva, Switzerland, July 2009.

[7] Sascha Konrad and Betty H. C. Cheng. Real-time Specification Patterns. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 372–381, New York, NY, USA, 2005. ACM.

[8] Predrag Filipovikj, Mattias Nyberg, and Guillermo Rodriguez-Navas. Reassessing the pattern-based approach for formalizing requirements in the automotive domain. In *Proceedings of the 22nd IEEE International Requirements Engineering Conference (RE)*, volume 00, pages 444–450, Los Alamitos, CA, USA, 2014. IEEE Computer Society.

[9] Amalinda Post, Igor Menzel, Jochen Hoenicke, and Andreas Podelski. Automotive Behavioral Requirements Expressed in a Specification Pattern System: A Case Study at BOSCH. *Requir. Eng.*, pages 19–33, 2012.

[10] Predrag Filipovikj, Trevor Jagerfield, Mattias Nyberg, Guillermo Rodríguez-Navas, and Cristina Seceleanu. Integrating Pattern-Based Formal Requirements Specification in an Industrial Tool-Chain. In *40th IEEE Annual Computer Software and Applications Conference, COMPSAC Workshops 2016, Atlanta, GA, USA, June 10-14, 2016*, pages 167–173. IEEE Computer Society, 2016.

[11] Xinhai Zhang, Magnus Persson, Mattias Nyberg, Behrooz Mokhtari, Anton Einarson, Henrik Linder, Jonas Westman, De-Jiu Chen, and Martin Törngren. Experience on applying software architecture recovery to automotive embedded systems. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014*, pages 379–382, 2014.

[12] Leonardo De Moura and Nikolaj Bjørner. Satisfiability Modulo Theories: Introduction and Applications. *Commun. ACM*, 54(9):69–77, September 2011.

[13] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] Jiří Barnat, Petr Bauch, and Luboš Brim. Checking Sanity of Software Requirements. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, pages 48–62, Berlin, Heidelberg, 2012. Springer-Verlag.

[15] Predrag Filipovikj, Guillermo Rodriguez-Navas, Mattias Nyberg, and Cristina Seceleanu. SMT-based Consistency Analysis of Industrial Systems Requirements. In *The proceedings of the 32nd ACM Symposium on Applied Computing (SAC). Marrakech, Morocco.* ACM, April 2017.

[16] Predrag Filipovikj, Nesredin Mahmud, Raluca Marinescu, Cristina Seceleanu, Oscar Ljungkrantz, and Henrik Lönn. Simulink to UPPAAL Statistical Model Checker: Analyzing Automotive Industrial Systems. In John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, volume 9995 of *Lecture Notes in Computer Science*, pages 748–756, 2016.

[17] A. David, D. Du, K.G. Larsen, A. Legay, M. Mikučionis, D.B. Poulsen, and S. Sedwards. Statistical Model Checking for Stochastic Hybrid Systems. *arXiv preprint arXiv:1208.3856*, 2012.

[18] Frederick P. Brooks, Jr. *The Mythical Man-month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[19] Bran Selic. Model-Driven Development: Its Essence and Opportunities. In *Ninth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2006), 24-26 April 2006, Gyeongju, Korea*, pages 313–319. IEEE Computer Society, 2006.

[20] J. B. Dabney and T. L Harman. *Mastering Simulink*. Pearson/Prentice Hall, 2004.

[21] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1:134–152, 1997.

[22] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.

[23] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, pages 359–364, London, UK, UK, 2002. Springer-Verlag.

[24] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series).* The MIT Press, 2008.

[25] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Journal of Theoretical Computer Science*, 126(2):183–235, April 1994.

[26] Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikučionis, Danny Bøgsted Poulsen, Jonas van Vliet, and Zheng Wang. *Statistical Model Checking for Networks of Priced Timed Automata*, pages 80–96. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[27] Alexandre David, Kim G Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. Uppaal SMC tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415, 2015.

[28] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on Uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Bertinora, Italy, September 13-18, 2004, Revised Lectures*, pages 200–236, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[29] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.

[30] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-Checking in Dense Real-Time. *Information and Computation*, 104(1):2 – 34, 1993.

[31] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available at `www.SMT-LIB.org`.

[32] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property Specification Patterns for Finite-state Verification. In *Proceedings of the Second Workshop on Formal Methods in Software Practice*, FMSP '98, pages 7–15, New York, NY, USA, 1998. ACM.

[33] Lars Grunske. Specification Patterns for Probabilistic Quality Properties. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 31–40, New York, NY, USA, 2008. ACM.

[34] Marco Autili, Lars Grunske, Markus Lumpe, Patrizio Pelliccione, and Antony Tang. Aligning Qualitative, Real-Time, and Probabilistic Property Specification Patterns Using a Structured English Grammar. *IEEE Transactions on Software Engineering*, 41(7):620–638, 2015.

[35] Rachel L. Cobleigh, George S. Avrunin, and Lori A. Clarke. User Guidance for Creating Precise and Accessible Property Specifications. In *SIGSOFT '06/FSE-14*, pages 208–218. ACM, 2006.

[36] Paola Inverardi, Henry Muccini, and Patrizio Pelliccione. CHARMY: An Extensible Tool for Architectural Analysis. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 111–114, New York, NY, USA, 2005. ACM.

[37] Oscar Mondragón, Ann Q. Gates, and Steven Roach. Prospec: Support for Elicitation and Formal Specification of Software Properties. *Electronic Notes in Theoretical Computer Science*, pages 67 – 88, 2003.

[38] Daniela Remenska, Tim A. C. Willemse, Jeff Templon, Kees Verstoep, and Henri Bal. Property Specification Made Easy: Harnessing the Power of Model Checking in UML Designs. In Erika Ábrahám and Catuscia Palamidessi, editors, *Formal Techniques for Distributed Objects, Components, and Systems: 34th IFIP WG 6.1 International Conference, FORTE 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014. Proceedings*, pages 17–32, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[39] Lori A. Clarke, Leon J. Osterweil, George S. Avrunin, and Rachel L. Smith. PROPEL: An Approach Supporting Property Elucidation. volume 00, page 11, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[40] Orna Kupferman. Sanity Checks in Formal Verification. In *Proceedings of the 17th International Conference on Concurrency Theory*, CONCUR'06, pages 37–51, Berlin, Heidelberg, 2006. Springer-Verlag.

[41] A. Post, J. Hoenicke, and A. Podelski. Vacuous Real-time Requirements. In *Proceedings of the 2011 IEEE 19th International Requirements En-*

gineering Conference, RE ’11, pages 153–162, Washington, DC, USA, 2011. IEEE Computer Society.

[42] Mats P. E. Heimdahl and Nancy G. Leveson. Completeness and Consistency in Hierarchical State-Based Requirements. *IEEE Trans. Softw. Eng.*, 22(6):363–377, June 1996.

[43] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Automated Consistency Checking of Requirements Specifications. *ACM Transactions Software Engineering Methodology*, 5(3):231–261, July 1996.

[44] Amalinda Post, Jochen Hoenicke, and Andreas Podelski. Rt-inconsistency: A New Property for Real-time Requirements. In *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering: Part of the Joint European Conferences on Theory and Practice of Software*, FASE’11/ETAPS’11, pages 34–49, Berlin, Heidelberg, 2011. Springer-Verlag.

[45] Matthew S. Jaffe, Nancy G. Leveson, Mats P. E. Heimdahl, and Bonnie E. Melhart. Software Requirements Analysis for Real-Time Process-Control Systems. *IEEE Trans. Softw. Eng.*, 17(3):241–258, March 1991.

[46] Hilary J. Holz, Anne Applin, Bruria Haberman, Donald Joyce, Helen Purchase, and Catherine Reed. Research Methods in Computing: What Are They, and How Should We Teach Them? *SIGCSE Bull.*, 38(4):96–114, June 2006.

[47] Marvin V. Zelkowitz and Dolores Wallace. Experimental Validation In Software Engineering. *Information and Software Technology*, 39:735–743, 1997.

[48] Predrag Filipovikj, Nesredin Mahmud, Raluca Marinescu, Guillermo Rodriguez-Navas, Cristina Seceleanu, Oscar Ljungkrantz, and Henrik Lönn. Analyzing Industrial Simulink Models by Statistical Model Checking. Technical report, March 2017.

[49] IBM Rational Doors. `http://www-03.ibm.com/software/products/en/ratidoor`, 2017. [Online; accessed 27-March-2017].

[50] Jiří Barnat, Petr Bauch, Nikola Beneš, Luboš Brim, Jan Beran, and Tomáš Kratochvíla. Analyzing Sanity of Requirements for Avionics Systems. *Form. Asp. Comput.*, 28(1):45–63, March 2016.

[51] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

[52] Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[53] Christian Ellen, Sven Sieverding, and Hardi Hungar. *Detecting Consistencies and Inconsistencies of Pattern-Based Functional Requirements*, pages 155–169. Springer International Publishing, Cham, 2014.

[54] Nesredin Mahmud, Cristina Seceleanu, and Oscar Ljungkrantz. ReSA Tool: Structured Requirements Specification and SAT-based Consistency-checking. In Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki, editors, *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems, FedCSIS 2016, Gdańsk, Poland, September 11-14, 2016.*, pages 1737–1746, 2016.

[55] F. Fabbrini, M. Fusani, S. Gnesi, and G. Lami. An Automatic Quality Evaluation for Natural Language Requirements. In *in Proceedings of the Seventh International Workshop on RE: Foundation for Software Quality (REFSQ2001*, pages 4–5, 2001.

[56] F. Fabbrini, M. Fusani, S. Gnesi, and G. Lami. The Linguistic Approach to the Natural Language Requirements Quality: Benefit of the Use of an Automatic Tool. In *Proceedings of the 26th Annual NASA Goddard Software Engineering Workshop*, SEW '01, pages 97–, Washington, DC, USA, 2001. IEEE Computer Society.

[57] Stefania Gnesi, Fabrizio Fabbrini, Mario Fusani, and Gianluca Trentanni. An automatic tool for the analysis of natural language requirements. *CRL Publishing: Leicester*, 20:53–62, 2005.

[58] Benedetta Rosadini, Alessio Ferrari, Gloria Gori, Alessandro Fantechi, Stefania Gnesi, Iacopo Trotta, and Stefano Bacherini. *Using NLP to Detect Requirements Defects: An Industrial Experience in the Railway Domain*, pages 344–360. Springer International Publishing, Cham, 2017.

[59] Benoît Boyer, Kevin Corre, Axel Legay, and Sean Sedwards. PLASMA-lab: A Flexible, Distributable Statistical Model Checking Library.

In Kaustubh Joshi, Markus Siegle, Mariëlle Stoelinga, and Pedro R. D'Argenio, editors, *Quantitative Evaluation of Systems: 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*, pages 160–164, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[60] Axel Legay and Louis-Marie Traonouez. Statistical Model Checking of Simulink Models with Plasma Lab. In Cyrille Artho and Peter Csaba Ölveczky, editors, *Formal Techniques for Safety-Critical Systems - Fourth International Workshop, FTSCS 2015, Paris, France, November 6-7, 2015. Revised Selected Papers*, volume 596 of *Communications in Computer and Information Science*, pages 259–264. Springer, 2015.

[61] Orlando Ferrante, Luca Benvenuti, Leonardo Mangeruca, Christos Sofronis, and Alberto Ferrari. Parallel NuSMV: A NuSMV extension for the verification of complex embedded systems. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7613 LNCS, pages 409–416, 2012.

[62] Ashlie B. Hocking, M. Anthony Aiello, John C. Knight, and Nikos Aréchiga. Proving Critical Properties of Simulink Models. In *Proceedings of IEEE International Symposium on High Assurance Systems Engineering*, volume 2016-March, pages 189–196, 2016.

[63] Jiri Barnat, Jan Beran, Lubos Brim, Tomas Kratochvíla, and Petr Ročkai. *Tool Chain to Support Automated Formal Verification of Avionics Simulink Designs*, pages 78–92. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[64] B. Meenakshi, Abhishek Bhatnagar, and Sudeepa Roy. Tool for Translating Simulink Models into Input Language of a Model Checker. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006. Proceedings*, pages 606–620, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[65] Aditya Agrawal, Gyula Simon, and Gabor Karsai. Semantic Translation of Simulink/Stateflow Models to Hybrid Automata Using Graph Transformations. *Electron. Notes Theor. Comput. Sci.*, 109:43–56, December 2004.

[66] Steven P. Miller. Bridging the Gap Between Model-Based Development and Model Checking. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, pages 443–453, Berlin, Heidelberg, 2009. Springer-Verlag.

[67] Karthik Manamcheri, Sayan Mitra, Stanley Bak, and Marco Caccamo. A Step Towards Verification and Synthesis from Simulink/Stateflow Models. In *Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control*, HSCC '11, pages 317–318, New York, NY, USA, 2011. ACM.

[68] Yu Jiang, Yixiao Yang, Han Liu, Hui Kong, Ming Gu, Jiaguang Sun, and Lui Sha. *From Stateflow Simulation to Verified Implementation: A Verification Approach and A Real-Time Train Controller Design.* Institute of Electrical and Electronics Engineers Inc., United States, 4 2016.

[69] Paolo Zuliani, André Platzer, and Edmund M. Clarke. Bayesian Statistical Model Checking with Application to Simulink/Stateflow Verification. pages 243–252, 2010.

[70] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. A storm is Coming: A Modern Probabilistic Model Checker. *arXiv preprint arXiv:1702.04311*, 2017.