Mälardalen University Licentiate Thesis
No.177

An Observer-Based Technique with Trace Links for Requirements Validation in Embedded Real-Time Systems

Jiale Zhou

October 2014

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

ISSN 1651-9256
ISBN 978-91-7485-160-1
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

Abstract

With the growing complexity of embedded real-time systems, requirements validation becomes an ever-more critical activity for developing such systems. Studies have revealed that most of the anomalies discovered in the development of complex systems belong to requirement and specification phases. Model-based techniques, enabling formal semantics and requirements traceability, are emerging as promising solutions to cost-effective requirements validation. In these techniques, the functional behaviors derived from lower-level requirements are specified in terms of analyzable models at a certain level of abstraction. Further, upper-level requirements are formalized into verifiable queries and/or formulas. Meanwhile, trace links between requirements at various levels of abstraction as well as between requirements and subsequent artifacts (such as verifiable queries and/or formulas, and analyzable models) are built, through which the queries and/or formulas can be fed into the corresponding models for further analysis. However, such model-based techniques suffer from some limitations, such as how to support semi- or fully-automatic trace links creation between diverse development artifacts, how to ease the demand of heavy mathematics background knowledge to specify queries and/or formulas, and how to analyze models without encountering the state explosion problem.

In this thesis, the technical contributions are four-fold: 1) we have introduced an improved Vector Space Model (VSM)-based requirements traceability creation/recovery approach using a novel context analysis and, 2) we have proposed a lightweight model-based approach to requirements validation by using the Timed Abstract State Machine (TASM) language with newly defined *Observer* and *Event* constructs and, 3) we have combined our model-based approach with a restricted use case modeling approach for feature-oriented requirements validation and, 4) we have improved the *Observer* construct of the extended TASM (eTASM) via proposing a new observer specification logic to

facilitate the observer specification, as well as defining the corresponding observer execution process. Finally, we have demonstrated the applicability of our contributions in real world usage through various applications.

Acknowledgments

I am greatly indebted to lots of people. Without their help and expert guidance, the work presented in this licentiate thesis would not have been possible. Here, I would like to express my gratitude, appreciation, and many thanks to them.

First of all, I would like to extend my sincere gratitude to my supervisors Prof. Kristina Lundqvist, Dr. Yue Lu and Prof. Mikael Sjödin for their support, encouragement and intensive guidance throughout my research work. It has been nothing but a great pleasure to work with you. I am looking forward to the work we have ahead of us.

High tribute shall be paid to Prof. Lars Asplund for his encouraging me to become a Ph.D. student; Prof. Kristina Forsberg for her industrial experience brought to my work and friendly invitation to her family activities; and Prof. Jakob Axelsson for his valuable suggestions on my research work. My special thanks should go to: my office-mates over the years, Andreas Johnson, Adnan Causevic, Hüseyin Aysan, Abhilash Thekkilakattil and Kaj Hänninen, for all the help and great office hours; Göran Bertheau and Kristian Wiklund, for their interest in my work and helpful suggestions; my Chinese colleagues, Yin Hang, Kan Yu, and Meng Liu, for their sharing work life and research experience.

My genuine thanks must be given to Damir Isovic, Hans Hansson, Thomas Nolte, Sasikumar Punnekkat, Ivica Crnkovic, Radu Dobrin, Frank Lüders, Jan Carlson, Henrik Lönn, Daniel Karlsson, Bo Liwång, et al., for all the guidance, help, inspiration and interesting discussions. I would also like to thank the administrative staff, Malin Rosqvist, Carola Ryttersson, Gunnar Widforss, Susanne Fronnå, et. al., for making many things easier.

I would like to jointly thank all the people at the IDT department, Mälardalen University. I have truly not seen a more friendly, encouraging, inspiring and open-minded environment to work in.

Here, I should have to mention a list of my Chinese friends outside of work: Shiliang Tong, Wenkai Wang, Ling Lu, Bo He, Zihao Liu, Wenjun

Wang, Yankai Shao, Bohan Guo, Lu Zhou, Tian Qiu, et al., for making my life vivid and much easier.

Finally, I would like to express my deepest gratitude to my beloved family. My deepest gratitude goes to my parents for loving considerations and great confidence in me all through these years. Many thanks go to my wife Mrs. Yuhui Zhao for being always supportive in all these rough and tough days and bringing endless love and happiness to my life.

Jiale Zhou
Västerås, October 2014

List of Publications

Papers Included in the Licentiate Thesis[1]

Paper A *A Context-based Information Retrieval Technique for Recovering Use-Case-to-Source-Code Trace Links in Embedded Software Systems*. Jiale Zhou, Yue Lu, Kristina Lundqvist. Proceedings of the 39th Euromicro Conference on Software Engineering and Advanced Applications (SEAA'13), Santander, Spain, September 2013.

Paper B *A TASM-based Requirements Validation Approach for Safety-critical Embedded Systems*. Jiale Zhou, Yue Lu and Kristina Lundqvist. Proceedings of the 19th Ada-Europe International Conference on Reliable Software Technologies (Ada-Europe'14), Paris, France, June 2014.

Paper C *Towards Feature-Oriented Requirements Validation for Automotive Systems*. Jiale Zhou, Yue Lu, Kristina Lundqvist, Henrik Lönn, Daniel Karlsson, Bo Liwång. Proceedings of the 22nd IEEE International Requirements Engineering Conference (RE'14), Karlskrona, Sweden, August 2014.

Paper D *The Observer-based Technique for Requirements Validation in Embedded Real-time Systems*. Jiale Zhou, Yue Lu, Kristina Lundqvist. Proceedings of the 1st International Workshop on Requirements Engineering and Testing (RET'14), Karlskrona, Sweden, August 2014.

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

Related Publication not Included in the Licentiate Thesis

- *Formal Execution Semantics for Asynchronous Constructs of AADL.* Jiale Zhou, Andreas Johnsen and Kristina Lundqvist. Proceedings of the 5th International Workshop on Model Based Architecting and Construction of Embedded Systems (ACES-MB'12), Innsbruck, Austria, October, 2012.

Contents

I

Thesis

Chapter 1

Introduction

1.1 Motivation

Embedded real-time systems (ERTS) have become an intrinsic part of human life, which include highly critical systems in multiple domains, such as automotive, avionics, and industrial automation. With the growing complexity of ERTS during the last decades, requirements can no longer be specified and analyzed merely at the outset of the systems development life cycle (SDLC). On the contrary, requirements tend to be formulated in a more complex setting in which there is a continuum of requirement and specification[1] levels as more and more details are added throughout the development life cycle. This setting presents an evolutionary development model of requirements and specifications, and implies a tight interleave between design activities and requirements engineering activities during the SDLC [1] [2], since the specifications produced at upper-level will serve the role of requirements with respect to the lower-level. Therefore, the term *requirements* and *requirement specifications* are used interchangeably in this thesis, if no explicit explanation.

There are various models describing the SDLC in diverse sizes and areas [3], among which the V-model is the most widely recognized in the field of ERTS. Figure 1.1 shows the traditional V-model SDLC featured with a three-level requirements development model considered in this thesis. At the beginning of the SDLC, it is an essential task to elicit a set of requirements for the

[1] A requirement specification is a collection of statements that describes the proposed system's behaviors aiming to meet the corresponding requirements.

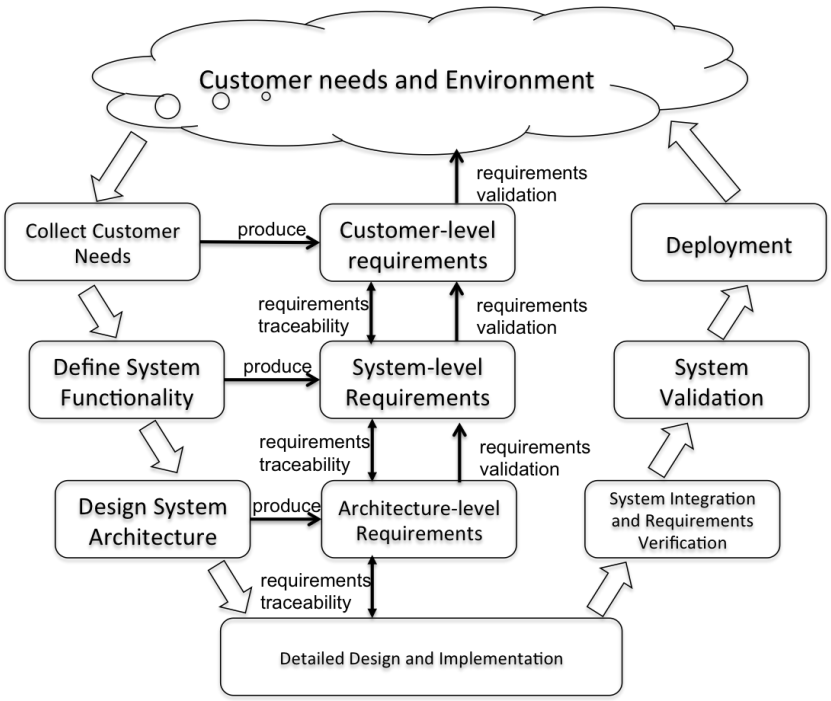

Figure 1.1: The V-model SDLC and the three-level requirements development model.

proposed system based on the opinions of various stakeholders, i.e., customer-level requirements. In doing so, a means of clarifying their needs and domain-specific problems and a basis for discussion with the stakeholders will be given. One example is “the customer wants a Brake-by-Wire system to substitute the traditional mechanical brake system”. Rather than proceeding straight to implementation phase, it is necessary to first determine what functionalities and properties the proposed system must have, regardless of more detailed design. Therefore, the initial customer-level requirements will be further transformed, in terms of being decomposed or refined, into a set of system-level requirements, to help to establish a common understanding of the proposed solution to the domain-specific problem. One typical example is “the system shall provide a base brake functionality where the driver presses the brake pedal so that

the braking system starts decelerating the vehicle". Based off of the system-level requirements, it is now possible to consider alternative design architectures. The design architecture defines what components the proposed system consists of and how such components interact with each other. Subsequently, the architecture-level requirements stipulate the requirements for each system component w.r.t. their functional behaviors, interaction constraints, and any other required properties featured with performance quality, reliability, safety, etc. One example is "the brake torque calculator shall compute the driver requested torque and send the value to the vehicle brake controller, when a brake pedal displacement is detected". Note that, in some cases, the components at the architecture-level are still too complex to be implemented directly, and therefore will be further refined into more specific requirement specifications corresponding to software components and hardware components which are associated with system implementation.

Many efforts have discussed the importance of a set of well-defined requirements [4] [5] and the severe consequences when they were ill-defined [6]. Nevertheless, it is still a challenge to produce a set of well-defined requirements in such a complex setting. Studies have revealed that most of the anomalies discovered in late development phases can be traced back to hidden flaws in the requirement phase [7] [8]. For instance, there exist contradictory functional behaviors in the requirement specifications; the architecture-level requirements do not satisfy all of the system-level requirements; or expected properties specified in the customer-level are discovered to be infeasible in the late phases of development. For this reason, requirements validation is playing an ever-more significant role in the development life cycle of ERTS, which confirms the correctness of requirements at different levels of abstraction, in the sense of consistency and completeness [1]. In detail, consistency refers to situations in which there exist no internal contradictions at each level of requirements, while completeness refers to situations in which the continuum of requirement levels must possess two fundamental characteristics in terms of neither objects nor entities are left undefined at each level of requirements and the requirements at lower-level can address all of the requirements at the corresponding upper-level.

Model-based techniques, enabling formal semantics and requirements traceability, are emerging as promising solutions to cost-effective requirements validation of ERTS [9] [10]. Such techniques are considered to possess the advantages of accelerating project development, requiring less human efforts, and making it possible to detect and fix the potential defects and errors in a fairly early stage of the SDLC. In model-based techniques, the functional behaviors

derived from lower-level requirements are specified as analyzable models at a certain level of abstraction. Further, upper-level requirements are formalized into verifiable queries and/or formulas. Meanwhile, trace links between requirements at various levels of abstraction as well as between requirements and subsequent artifacts (such as verifiable queries and/or formulas, and analyzable models) are manually or (semi-)automatically created/recovered, through which the queries and/or formulas can be fed into the corresponding models. In this way, the requirements are reasoned about to resolve behavioral contradictions and to discover unexpected behaviors, and it is also verified that they are neither so strict to forbid desired behaviors, nor so weak to allow undesired behaviors. Especially, model checking [11] and theorem proving [12] are most widely explored as formal model-based techniques for the requirements validation purpose. Model checking is a formal technique for automatically and exhaustively verifying correctness properties against a finite-state system. On the contrary, theorem proving is an interactive formal technique where a designer employs a theorem-proving tool through partially guided, rigorous proof steps, to show that the implementation implies the property of interest. However, such formal model-based techniques enabling trace link creation/recovery also suffer from some limitations: 1) how to support semi- or fully-automatic trace links creation between diverse development artifacts [13] and, 2) how to analyze the system model without having the state explosion problem of model checking occurred and, 3) how to ease the demand of heavy mathematics background knowledge to perform theorem proving.

1.2 The Overview of Contributions

In this thesis, we have tackled the aforementioned limitations and contributed to a model-based technique for the purpose of requirements validation. In particular, the technical contributions are four-fold:

- We have introduced an improved VSM-based requirements traceability recovery approach using a novel context analysis. As aforementioned, in the three-level requirements development model featured with customer-level, system-level and architecture-level, our method can better utilize the context information extracted from the upper-level requirements to discover the subsequent artifacts at the lower levels.

- We have proposed an observer-based approach to requirements validation by using the Timed Abstract State Machine (TASM) language [14]

with newly defined *Observer* and *Event* constructs, namely the extended TASM (eTASM) language . To be specific, our approach models both functional and non-functional requirements of the system under consideration at different levels, aiming to validate the completeness and consistency of requirements by utilizing our in-progress toolset and a model checker.

- We have presented a model-based approach to feature-oriented requirements validation by utilizing the eTASM. Especially, the approach starts with the behavioral specification of features and the associated requirements by following a restricted use case modeling approach, and then formalizes such specifications by using the eTASM language for the purpose of validation.

- We have improved the eTASM language via proposing a new observer specification logic, namely Events Monitoring Logic, to facilitate the observer specification, as well as defining the corresponding observer execution process.

1.3 Thesis Outline

This thesis is divided into two parts: **Part I** includes four chapters. Chapter 1 provides an introduction of the thesis where the motivation of our work and an overview of the thesis contributions are presented. In Chapter 2, we describe the background knowledge of the research work underlying the thesis, and related work. In Chapter 3, a research overview is presented, including the research questions guiding our work, the challenges associated with each question, our contributions to each question, and the research methodology adopted in this thesis. In Chapter 4, we summarize the thesis work with concluding remarks, and ending with a discussion of the future work.

Part II incorporates the research papers included in this thesis, which are organized as:

- **Chapter 5 Paper A**: A Context-based Information Retrieval Technique for Recovering Use-Case-to-Source-Code Trace Links in Embedded Software Systems

- **Chapter 6 Paper B**: A TASM-based Requirements Validation Approach for Safety-critical Embedded Systems.

- **Chapter 7 Paper C**: Towards Feature-Oriented Requirements Validation for Automotive Systems.
- **Chapter 8 Paper D**: The Observer-based Technique for Requirements Validation in Embedded Real-time Systems

Chapter 2

Background and Related work

This chapter presents some background knowledge that underlies this thesis and the related work in the field of requirements validation.

2.1 Background

Requirements validation is an indispensable activity that is performed to increase the confidence that the system under consideration would meet its requirements, which is imposed in the early phases of the SDLC. Different from system validation which confirms that the built system does what it is supposed to do, requirements validation is the process of confirming the *consistency* and *completeness* of requirements throughout various levels of abstraction. Further, once consistency and completeness of requirements are validated, then the *correctness* of such requirements can be achieved [1]. Consistency refers to situations in which requirements contains no internal contradictions. Completeness implies requirements must exhibit two fundamental characteristics: 1) The information does not contain any undefined objects or entities in the continuum of requirements, and 2) lower-level requirements are able to satisfy upper-level requirements.

Regarding the requirements development model considered in our thesis, each of the three requirement levels has their own validation purpose as follows:

- The *customer-level requirements* must be guaranteed that they collect all of the stated business objectives, clarify the needs of various stakeholders, and are easy-to-use for communication between stakeholders and developers.
- The *system-level requirements* provide a high-level solution to implementation of the system under consideration, which must be validated to make sure that they address the stated business objectives, meet the needs of stakeholders, and contain no internal contradictions.
- The validation procedure for the *architecture-level requirements* not only ensure that they are consistent and complete with the upper-level requirements (i.e., system-level requirements), but also makes sure the design solution is feasible, unambiguous and verifiable.

There are numerous requirements validation techniques available in the literature, which can be roughly categorized into three categories in terms of *requirements review* [15] [16] [17], *requirements prototyping* [18] and *model-based requirements validation* [10]. *Requirements review* is conducted by following a well-planed process, during which the requirements are manually but thoroughly searched, sorted and analyzed by domain experts, customers, developers and/or other stakeholders. *Requirements prototyping* is a process that transforms written requirements into a workable version of the system under consideration yet with limited functionalities. Prototypes aim at helping stakeholders communicate with each other and identifying errors and omissions in the requirements. With the increasing complexity of ERTS, *model-based techniques* enabling formal semantics and requirements traceability, such as model-checking and theorem proving, stand out as promising solutions to cost-effective requirements validation of ERTS.

This thesis aims to contribute to requirements traceability and model-based requirements validation techniques for ERTS, of which background knowledge is presented in the following sections.

2.1.1 Embedded Real-Time Systems

An embedded real-time system typically have a primary purpose of providing at least partial control of the system or environment in which it is embedded [19]. Automotive systems such as electronic brake systems are typical examples of ERTS. Most of such systems can be modeled by using four types of components in the sense of *environment*, *sensors*, *controllers*, and *actuators*.

Environment models the behavior of the external entities interacting with the ERTS by defining a set of variables. *Sensors* are used to monitor the behaviors of *environment* by measuring the variables of interest. *Controllers* are the components used to perform computation in order to implement the control function. *Actuators* model the components designed to affect the *environment* by manipulating the corresponding variables.

Two inherent characteristics can explicitly distinguish ERTS from general computer systems [20]: 1) From the real-time perspective, time plays a critical role in defining the correctness of an embedded real-time system since a correct answer provided too late or too early can be as erroneous as providing an incorrect answer, and 2) from the embedded perspective, ERTS represent a special class of real-time systems where the software system is not stand-alone, but is part of a larger system and must work with other components to achieve the system's goals.

The first characteristic stipulates that requirements validation for an embedded real-time system should be defined in terms of two key aspects – functional behaviors and non-functional properties. Meanwhile, the second characteristic implies a more complex setting of performing requirements validation. When several ERTS are put together to form a larger system, these sub-systems usually run simultaneously and independently, possibly on different hardware. Communication between different sub-systems is usually conducted in indirect ways. To be specific, the sub-systems and their environment form a closed loop where the output from the actuator of a sub-system changes the variables of the environment, which is then read through the sensors and becomes input to other sub-systems at a later time. Under this circumstance, unexpected behaviors can arise from the activation of two or more ERTS whose outputs from different actuators create contradictory physical forces on the same physical environment.

2.1.2 Information Retrieval (IR)-Based Traceability Creation/Recovery

Tracing requirements throughout the life cycle supports engineers in ensuring requirements coverage, saving efforts for validation when changes occur, etc. The challenges connected to traceability have been empirically investigated and reported over the years, among which trace link creation/recovery remains an interesting one to tackle [21].

The information retrieval (IR)-based traceability technique aims at utilizing IR techniques to create/recover trace links between diverse development

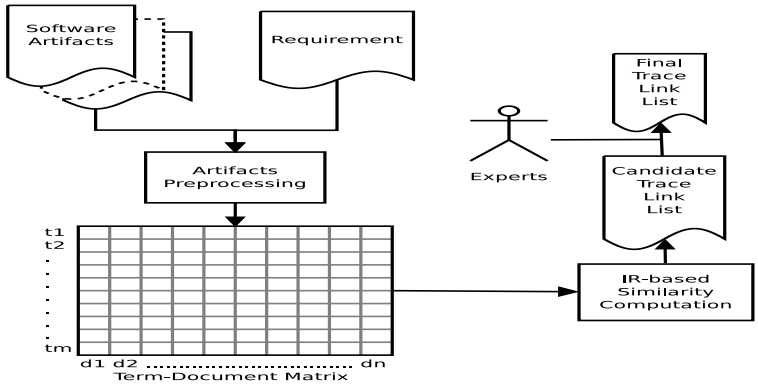

Figure 2.1: An IR-based traceability recovery process.

artifacts. Typically, an IR-based traceability creation process follows the steps depicted in Figure 2.1. The artifacts have to be preprocessed before they are used to compute similarity scores. The preprocessing of the artifacts includes a text normalization by removing most non-textual tokens (e.g., operators, punctuations) and splitting compound identifiers into separate words by using the underscore or camel case splitting heuristic. Furthermore, common terms, referred to as "stop words" (e.g., articles, prepositions, common usage verbs, and programming language keywords), which contribute less to the understanding about artifacts, are also discarded by using a stop word filter. Words with the length less than a defined threshold are also pruned out. In addition, stemmer is commonly used to perform a morphological analysis, which reduces the inflected words to their root, e.g., returning verb conjugations and removing plural nouns. After preprocessing, an artifact (e.g., a requirement) can be represented as a plain document containing a list of terms (in this thesis, we use *documents* and *artifacts* interchangeably). The extracted terms are generally stored in a $m \times N$ matrix (called term-by-document matrix), where m is the number of all the terms that occur in all the documents, and N is the number of documents in the corpus. A generic entry $w_{i,j}$ of the matrix denotes a measure of the relevance of the i_{th} term in the j_{th} document. Based on the term-by-document matrix representation, diverse IR methods can be used to calculate textual similarities between paired artifacts. The standard Vector Space Model (VSM) [22] technique has been successfully applied to calculate the similarity score between artifacts in the requirements traceability

creation/recovery process [23]. In VSM, given the entire collection of unique terms $T = \{t_1, \ldots, t_m\}$ in a corpus with N documents, the document d_n is represented as a vector $d_n = \{w_{1,d_n}, \ldots, w_{m,d_n}\}$ consisting of m unique terms from the corpus with an assigned weight w_{i,d_n} through a certain weighting scheme. Therefore, the similarity score, denoted as $sim(q, d)$, between the query document q and the target document d is calculated by using the cosine of the angle between their vectors:

$$sim(q,d) = \frac{\sum_{i=1}^{m} w_{i,q} \cdot w_{i,d}}{\sqrt{\sum_{i=1}^{m} w_{i,q}^2 \cdot \sum_{i=1}^{m} w_{i,d}^2}} \tag{2.1}$$

The weighting scheme $w_{i,q}$ and $w_{i,d}$ denoting the *term frequency-inverse document frequency* (i.e., *tf-idf*) are calculated as follows:

$$w_{i,q} = tf_i(q) \cdot idf_i, \quad w_{i,d} = tf_i(d) \cdot idf_i \tag{2.2}$$

where $tf_i(q)$ and $tf_i(d)$ are measured by the number of times the term t_i occurs in the query document q and the target document d respectively, and idf_i is computed as $log(\frac{N}{df_i})$, where df_i is the number of documents containing the term t_i.

Pairs with a similarity score lower than a certain threshold (usually defined based on engineers' experience) are filtered out, and the reserved pairs form the candidate trace link list. The ranked list of candidate trace links are then vetted by software engineers to decide if such links are true positive or not.

2.1.3 Model-Based Requirements Validation

The principal idea of the model-based requirements validation is that the system requirements at different levels of abstraction are specified in corresponding (semi-)formal specification languages in order to perform formal verification, such as model-checking and theorem proving. In this section, we have an overview of model checking and theorem proving. An introduction of the TASM specification language together with a lightweight verification technique used in this thesis will be given as well.

Model Checking

Model checking [11] is a formal technique for automatically and exhaustively verifying correctness properties against a finite-state system. Given a finite-state system M and a correctness property ρ, the state transition graph of M is

algorithmically traversed to verify if ρ holds in the current available state. If the property holds in all of the system states, the property is marked as satisfied. Otherwise, model checking can produce a counter-example i.e., a partial execution trace leading to a system state where the property is not satisfied by the model. The model checking approach is conceptually simple and is applicable in a wide variety of languages and application areas. However, this approach highly relies on the exhaustive exploration of the state space, which has suffered from the notorious state-space explosion problem. Newer approaches relying on a symbolic representation of the state space can significantly improve the performance of model checking, but only for systems with simple, repetitive elements such as hardware applications.

Theorem Proving

Theorem proving [12] is an interactive formal technique, compared to model checking. In this approach, both upper-level requirements and corresponding lower-level specifications are represented as logic descriptions e.g., using process algebras or higher-order predicate logics. Then, a designer employs a theorem-proving tool through partially guided, rigorous proof steps, to show that the specifications can imply the requirements and contain no internal contradictions. Unfortunately, the logic descriptions commonly used in the theorem proving approach are typically not understandable by the non-software stakeholders involved in most requirements engineering activities, and thus are hardly suitable as high-level requirement languages.

An Overview of TASM

In this thesis, we use the Timed Abstract State Machine language [14] as the formal specification language which we will extend for the purpose of requirements validation. TASM is a textual formal language for the specification of ERTS, which extends the Abstract State Machine (ASM) [24] with the capability of modeling timing properties and resource consumption of the system under consideration. TASM inherits the easy-to-use feature from ASM, which is a literate specification language understandable and usable without extensive mathematical training [25]. A TASM model consists of two parts – an environment and a set of main machines. The environment defines the set and the type of variables, and the set of named resources which machines can consume. The main machine is made up of a set of monitored variables which can affect the machine execution, a set of controlled variables which can be modified by

machines, and a set of machine rules. The set of rules specify the machine execution logic in the form of "if *condition* then *action*", where *condition* is an expression depending on the monitored variables, and *action* is a set of updates of the controlled variables. We can also use the rule "else then *action*" which is enabled merely when no other rules are enabled. A rule can specify the annotation of the time duration and resource consumption of its execution. The duration of a rule execution can be the keyword *next* that essentially states the fact that time should elapse until one of the other rules is enabled. Figure 2.2 shows a toy example of the TASM language, which models the behavior of a switch turning on/off a light [14].

TASM describes the basic execution semantics as the computing steps with time and resource annotations: In one step, it reads the monitored variables, selects a rule of which *condition* is satisfied, consumes the specified resources, and after waiting for the duration of the execution, it applies the update set instantaneously. If more than one rules are enabled at the same time, it nondeterministically selects one to execute. As a specification language, TASM supports the concepts of parallelism which stipulates TASM machines are executed in parallel, and hierarchical composition which is achieved by means of auxiliary machines which can be used in other machines. There are two kinds of auxiliary machines - *function* machines which can take environment variables as parameters and return execution result, and *sub* machines which can encapsulate machine rules for reuse purpose [14]. Communication between machines, including main machines and auxiliary machines, can be achieved by defining corresponding environment variables. All the aforementioned features of TASM make it as a good-fit for some key activities of requirements engineering for ERTS, e.g., model-based requirements validation for ERTS.

Runtime Monitoring

Runtime monitoring comprises having an observer monitor the execution of a system and check its conformity with a property of interest. Comparing model checking and theorem proving, runtime monitoring implements a lightweight technique for the purpose of system verification [26], which inspires us to extend the TASM language with observer constructs to perform requirements validation in a lightweight way. The basic principle of runtime monitoring is that: when a system is running, it will generate a number of events $a_1, a_2, \ldots, a_n$ ($n \geq 0$) reflecting the functional behaviors and non-functional properties; the events can be abstracted as a linear trace $\omega = a_1 a_2 \ldots a_n$; an observer representing a given correctness property of interest is usually specified as a logic

```
ENVIRONMENT:
  VARIABLES:
    light_status light := OFF;
    switch_status switch := DOWN;
  USER-DEFINED TYPES:
    light_status := {ON, OFF};
    switch_status := {UP, DOWN};
  RESOURCES:
    power:=[0,10]                    % user-defined named resource
MAIN MACHINE:
  MONITORED VARIABLES:
    switch;
  CONTROLLED VARIABLES:
    light;
  RULES:
    R1: Turn On{                     % the name of the rule
      t:= 1;                         % the time duration of the rule
      power:=[2,5];                  % the resource consumption of the rule
      if light = OFF and switch = UP then   % the rule body
        light := ON;
    }
    R2: Turn Off {
      t:= [1,2];
      power:=[3,5];
      if light = ON and switch = DOWN then
        light := OFF;
    }
```

Figure 2.2: A toy example of the TASM language

expression E; then, the monitoring process can be regarded as solving the membership problem for the given logic expression E and trace ω, which is to determine whether ω is in the language defined by E. The observer can either start monitoring as soon as any event is available, or store the available events at first and then start working when a stored trace is available. In the former case, we speak of on-line monitoring, while in the latter case we speak of off-line monitoring.

2.2 Related Work

In this section, we describe the state-of-the-art in the related fields, in the sense of traceability creation/recovery (related with Paper A), model-based requirements validation (related with Paper B and Paper C) and runtime monitoring (related with Paper D).

2.2.1 Traceability Creation/Recovery

Many recent studies have explored the feasibility of different IR methods for semi-automatically or fully-automatically creating/recovering trace links between development artifacts. Deerwester et al. [27] discuss the effectiveness of Latent Semantic Indexing (LSI) for recovering trace links between different kinds of artifacts, and Marcus et al. [28] further the work, showing a promising result over VSM. Abadi et al. [29] present a novel IR technique based on Jensen & Shannon (JS) model. They also compared JS, VSM and LSI for traceability recovery purpose, and concluded that VSM and JS are the best-fits. A similar comparison is also conducted by Oliveto et al. [30], which showed that for building trace links between requirements and source code, the results of JS, VSM and LSI are almost equivalent. Based on the aforementioned results, we have chosen the VSM-based method to implement our context analysis approach.

Variants of basic IR methods have been proposed to improve the accuracy of IR-based traceability recovery approaches. Fautsch et al. [31] present four extensions to the classical *tf-idf* VSM model. The basic idea is to retrieve domain specific information. Kong et al. [32] present a VSM enhancement using term location. By utilizing the relationship between words in different textual documents, a better accuracy was achieved. Lucia et al. [33] present the approach that uses smoothing filter to improve the input in the IR-based traceability recovery process. Specifically, the words that contribute less information but repeatedly occur in the documents, are removed. Cleland-Huang et al. [34] introduce three accuracy enhancement strategies, which are hierarchical modeling, logical clustering of artifacts, and semi-automatic pruning of the probabilistic network. Our approach is a variant of IR methods as well, but it is different from the strategies proposed in the prior pieces of work. We propose to deal with the context information differently from the real intent of requirements. Therefore, our approach is possible to be combined with those prior pieces of work, which can be a future research direction.

Some other pieces of work also show us another promising perspective. Asuncion et al. [35] apply Topic Modeling technique, featured by Latent Dirichlet Allocation (LDA) [36] to capture trace links prospectively. Lucia et al. [37] discuss the feasibility of using user feedback analysis to improve the accuracy of the results of traceability recovery tools. Mahmoud et al. [38] propose a semantic relatedness approach that exploits external knowledge sources, e.g., Wikipedia, to identify a set of relevant terms that are used to expand the query, in an attempt to improve traceability results.

2.2.2 Model-Based Requirements Validation

In the field of model-based requirements validation, there are several interesting pieces of work deserved to be mentioned. Event-B [39] similar with eTASM is also a formal state-based modeling language that represents a system as a combination of states and state transitions. Mashkoor et al. [40] propose a set of transformation heuristics to validate the Event-B specification by using animation. Iliasov [10] shows how to use Event-B for systems development, where the system constraints are formalized as a set of visualized proof obligations which can be synthesized as use cases. Such proof obligations are then reasoned about their satisfaction in the corresponding Event-B model. This work mainly focus on the validation of use cases. Cardei et al. [41] present a methodology that first converts SysML requirements models into the proposed requirements ontology model, and then performs the rule-based reasoning to detect omissions and inconsistency. Different from our work, we validate requirements from the behavioral perspective. Cimatti et al. [9] introduce a series of techniques that have been developed for the formalization and validation of requirements for safety-critical systems. Specifically, the methodology consists of three main steps in terms of informal analysis, formalization, and formal validation. Our approach has similar but more detailed steps, and we use eTASM as the specification language which orients to embedded real-time systems. Scandurra et al. [42] propose a framework to automatically transform use cases into ASM models, which are used to validate the requirements through scenario-based simulation. However, in this work, non-functional properties are not considered. MARTE [43] is a UML profile for modeling and analysis of real-time embedded systems, covering both functional and non-functional properties of the system. Nevertheless, to our best knowledge, there has not been any work about using MARTE for the purposes of requirements validation.

A variety of model-based approaches have been proposed to perform feature-oriented requirements validation. Kimbler et al. [44] introduce a user-oriented approach to feature interaction analysis of telephony services. It aims first at creating use case models to describe different possible ways of using the system services, and then building service usage models which simulate the dynamic relations between services. Moreover, Amyot et al. [45] propose an approach to detecting feature interactions of telecommunication systems as well, by using Use Case Maps (UCMs) for designing features, and LOTOS for the formal specification of features. However, their work mainly aimed at modeling the features of the telecommunication system which are different from the

one of automotive systems. The approach proposed in this thesis aims to detect the feature interaction problem in the automotive domain. In automotive industrial, it becomes increasingly popular to organize requirements by using feature models. Sampath et al. [46] present a formal specification and analysis method for automotive features in the early stages of software development process. This method starts with an empty specification, and then incrementally adds clauses to the specification until all the feature requirements are satisfied. The evolutionary approach inspires us to explore the possibility of using eTASM in an evolutionary way in the future. Arora et al. [47] propose a method and algorithms for identifying and resolving feature interactions in the early stages of the software development life-cycle. The work uses *State Machines* to model the behavior of independent features, context diagrams to integrate independent features, and Live Sequence Charts to capture the interactions of features. Compared to this work emphasizing the analysis of requirements specification, our approach introduces restricted use case models [48] as intermediate artifacts between natural language specifications (NLS) and formal specifications, in order to reduce ambiguities caused by NLS and to increase the automation from NLS to other formalisms.

2.2.3 Runtime Monitoring

In the runtime monitoring field, many logics are used for the purpose of validation and verification. Giannakopoulou et al. [49] present an approach to checking a running program against Linear Temporal Logic (LTL) specifications. In particular, the LTL formulae representing the properties of interest are translated into finite-state automata, which are used as observers monitoring the program behaviors. Barringer et al. [50] present a compact and powerful logic, namely Eagle, which is based on recursive parameterized rule definitions over the standard propositional logic operators together with three primitive temporal operators in the sense of a past-state operator, a next-state operator, and a concatenation-state operator. Basin et al. [51] extend the metric first-order temporal logic (MFOTL) with aggregation operators in order to specify observers that represent the compliance policies on aggregated data. Compliance policies represent normative regulations, which specify permissive and obligatory actions for system users. The authors provide a monitoring algorithm for the enriched observer specification language as well. However, to our best knowledge, the extended regular expressions (ESE) language is much more widely accepted in practice for its simplicity. Roşu et al. [52] present a rewriting algorithm for testing membership of a word in a regular language described

by an extended regular expression. The algorithm is based on an event consumption idea: a just arrived event is consumed by the regular expression, i.e., the extended regular expression modifies itself into another expression when dropping the event. In our opinion, this algorithm can be easier to implement and grasp by engineers, compared to the aforementioned pieces of work. Furthermore, we have proposed our observer monitoring logic based on the event consumption idea.

Some other pieces of work discuss runtime monitoring from another perspective, which are worth to notice. Bauer et al. [53] discuss a three-value semantics (false, true, inconclusive) for LTL and TLTL observers on finite traces, where an observer outputs *false* when a finite prefix is impossible to be the prefix of any accepting trace and, *true* when a finite prefix can be accepted by any infinite extension of the trace and, *inconclusive* in other cases. Additionally, Falcone et al. [54] give an related and interesting discussion about the monitorability of properties in the safety-progress classification. Leucker et al. [26] present a brief account of the field of runtime monitoring. They give a definition of runtime monitoring and make a comparison to well-known verification techniques in terms of model checking and testing.

Chapter 3

Research Overview

3.1 Research Questions, Challenges and Contributions

In this thesis, we aim at contributing to a general research question as follow:

- **General research question:** How can, in accordance with industrial demands, requirements of complex and dependable embedded real-time systems be validated, in a structured and practical way?

During our research, we have had four more specific research questions that guided our work to contribute to the general question. In this section, we present such specific research questions, describe research challenges associated with each question, and summarize our contributions to each question. Table 3.1 presents the relations between our research questions and papers.

Table 3.1: Relations between the papers and research questions.

Paper	Research Question	Current State
Paper A	RQ1	Published at SEAA'13
Paper B	RQ2	Published at Ada-Europe'14
Paper C	RQ3	Published at RE'14
Paper D	RQ4	Published at RET'14

3.1.1 Research Question 1 (RQ1)

How can we improve the quality (i.e., achieve higher level of precision and recall) of semi-automatic or fully-automatic creation of trace links between requirements at various levels of abstraction throughout systems development life cycle, in order to facilitate requirements validation?

- **The challenges associated with RQ1 (Ch1):** Despite the wide recognition of its necessary support to requirements validation [55], effective traceability is still rarely established in contemporary industrial settings, especially for legacy systems [56]. This phenomenon may be attributed to the difficulty in automating the generation of trace links. Manual establishment and maintenance of trace links tend to be costly to implement and are therefore perceived as financially nonviable by many companies [55], [57]. To address this problem, many efforts [27], [32], [33], [34], [36], [38], [58] have been devoted to semi-automatic or fully-automatic trace link creation. However, the precision and recall of generated trace links are still at a low level of accuracy, such that the trace link creation throughout the entire systems development process, remains a challenging issue [21].

  - **Our contributions to RQ1 (OC1):** We have improved the Vector Space Model (VSM)-based requirements traceability recovery approach by using a novel context analysis. Specifically, the analysis method can better utilize context information extracted from requirements (e.g., use cases) to discover relevant subsequent artifacts (e.g., source code files). Our approach has been evaluated by using three different embedded applications in the domains of industrial automation, automotive and mobile. The evaluation has shown that our new approach can achieve better accuracy than VSM, in terms of higher values of three main information retrieval (IR) metrics, i.e., precision, recall, and mean average precision, when it handles embedded software applications.

3.1.2 Research Question 2 (RQ2)

How can we validate requirements at various levels of abstraction in a lightweight way, compared to *formal methods*, in early stages of model-based development life cycle of embedded real-time systems?

- **The challenges associated with RQ2 (Ch2):** In order to increase the confidence in the correctness of requirements, model-based *formal methods* techniques, such as model checking and theorem proving, have been to a large extent investigated into the field of requirements validation [9] [10]. In these techniques, the system structure and behaviors derived from lower-level requirements are often specified in terms of analyzable models at a certain level of abstraction. Further, upper-level requirements are formalized into verifiable queries or formulas and then fed into the models to perform model checking and/or theorem proving. In this way, the lower-level requirements are reasoned about to resolve contradictions, and it is also verified that they are neither so strict to forbid desired behaviors, nor so weak to allow undesired behaviors. However, such formal methods techniques also suffer from some limitations, such as how to ease the demand of heavy mathematics background knowledge to perform theorem proving, and how to model the target without having the state explosion problem of model checking occurring. Therefore, a lightweight approach to requirements validation is of paramount importance to achieve success.

  - **Our contributions to RQ2 (OC2):** We have extended a formal specification language Timed Abstract State Machine (TASM) with two newly defined constructs *Event* and *Observer*, and have proposed an observer-based approach to requirements validation by using the eTASM. Specifically, our approach can: 1) model both functional and non-functional requirements of the system under consideration at different levels of abstraction and, 2) perform requirements validation by utilizing our developing toolset and a model checker. Furthermore, we have demonstrated the applicability of our approach in real world usage through an industrial application of a Brake-by-Wire system.

3.1.3 Research Question 3 (RQ3)

How can we validate feature-oriented requirements, in order to detect the feature interaction problem that refers to the situation in which two or more features exhibit unexpected behaviors in the requirements specification?

- **The challenges associated with RQ3 (Ch3):** Feature models, as a domain-specific requirements model, can capture commonality and variability of a software product line through a set of features. With the

increasing size of feature models, the inherent variability of the feature sets will easily lead to the feature interaction problem (referring to the situation that two or more features exhibit unexpected behaviors). Many developed techniques [45], [47], [59] start with translating the natural language specification (NLS) of a feature into a formal language specification (FLS) of the feature behaviors, and then requirements validation is performed based on the generated formalisms. The main challenges facing these techniques lie in: 1) ambiguities in the NLS cause imprecise definitions and even wrong understanding of the feature behaviors and, 2) the direct translation from the NLS to an FLS tends to be very costly and, 3) the NLS hinders to a large extent the possibility of performing automatic feature-oriented requirements validation.

  - **Our contributions to RQ3 (OC3):** We have proposed a feature-oriented requirements validation approach. Such approach starts with the behavioral specification of features and the associated requirements by following a restricted use case modeling approach, and then formalizes such specifications by using the eTASM language for analysis. Moreover, we have demonstrated the applicability of our approach through a Vehicle Locking-Unlocking system.

3.1.4 Research Question 4 (RQ4)

How can we improve the observer-based approach proposed in our previous work, Paper B [60] and Paper C [61], for the purpose of validating the requirements of such embedded real-time systems that are featured with complicated behaviors?

- **The challenges associated with RQ4 (Ch4):** In our previous work [60] and [61], we assume that the observer representing the property of interest can be specified by following the logic of regular expressions, and the entire event trace generated by the eTASM models should be available before observers start to monitor and analyze the trace. The monitoring algorithm is implemented in an off-line way as searching for a specific word matching the regular expression in a prepared document. However, the main drawbacks of these assumptions are two-fold: 1) the expressiveness power of regular expressions falls short of expressing unordered fixed-count events where the occurrence multiplicities of these events are pre-defined but the corresponding order is random and, 2) the

monitoring algorithm used in [60] can not be applied at runtime because of the assumption that the entire event trace is pre-achieved. Therefore, improving the logic used to specify observers is of paramount importance for our observer-based validation technique to achieve success in practice.

- **Our contributions to RQ4 (OC4):** 1) We have given the formal definitions of the observer constructs and observer-related concepts, as well as have defined the corresponding observer execution process and, 2) we have proposed the Events Monitoring Logic (EvML) to facilitate the observer specification and, 3) we have presented an execution process of running observers to validate the requirements describing functional behaviors and non-functional properties of embedded real-time systems. Finally, we have illustrated the applicability of our technique by using a Vehicle Locking-Unlocking system.

3.2 Research Methodology

In order to adequately contribute to our general research question listed above, a research methodology suitable for such a given setting is planned and followed. The methodology used in our research is based on the research steps proposed by Shaw [62], which is summarized in the following steps, and shown in Figure 3.1:

1. Formulating our initial research idea, based on the general research question.

2. Conducting literature reviews to study the state of the art in the requirements validation field.

3. Understanding the current research settings and deriving a specific research question from the research idea to guide our work.

4. Analyzing the state of the art in the requirements validation field based on the guiding research question.

5. Answering the research question by presenting the proposed solutions and achieved research results.

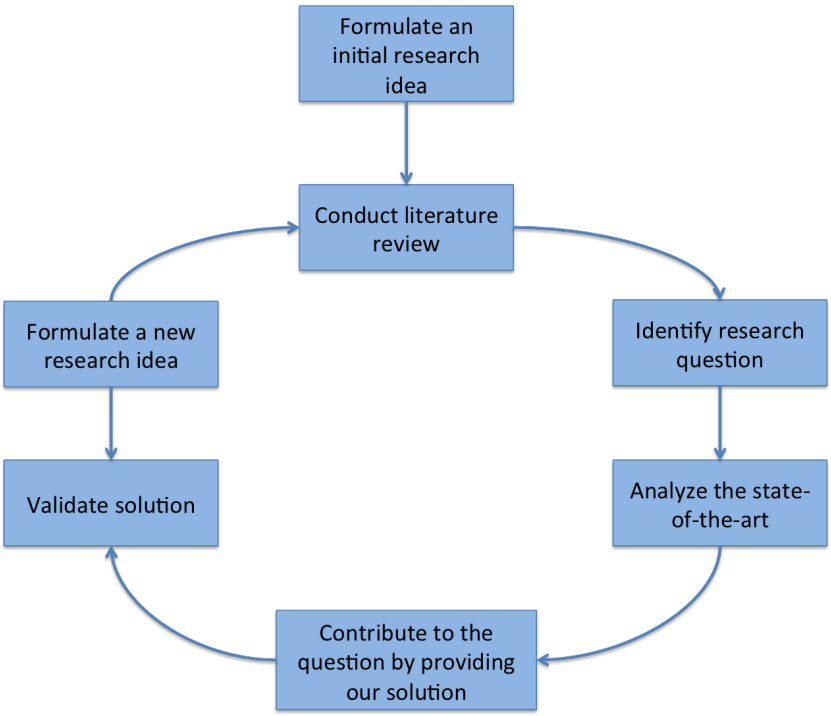

Figure 3.1: The main research steps.

6. Validating whether the research results can be applied in the real-world applications.

7. Formulating a new research idea through the experience gained from our previous research, in order to further answer our general research question.

In this work, the general research question helps us formulate the initial research idea at step 1. Then, the planned steps (2 - 7) are performed iteratively to conduct our research until the desired results for our general research question is achieved.

Chapter 4

Conclusions and Future Work

4.1 Conclusions

Requirements validation is an inevitable activity in the systems development life cycle (SDLC), ensuring the successful development of embedded real-time systems (ERTS). In this thesis, we have contributed to requirements validation from two aspects in terms of trace link creation/recovery and model-based requirements validation.

In order to improve the quality of trace links, we have proposed a new Vector Space Model (VSM)-based approach for requirements traceability creation/recovery in **Paper A**, which uses a novel context analysis. Specifically, our approach extracts context information (such as use case titles and preconditions) from the requirement documents in the beginning. The context information and requirements are respectively processed to generate a list of trace links. Finally, the two lists are combined together to form a final list of trace links, through a weighted knowledge model. Comparing the standard VSM technique, the experiment results have shown that our approach can obtain better quality candidate trace link lists, in terms of higher scores of three main information retrieval metrics [13], i.e., *recall*, *precision*, and *mean average precision (MAP)*.

Since model checking and theorem proving involve exhaustive search of system states and highly human interaction, respectively, we aim at develop-

ing a lightweight approach to requirements validation. Briefly speaking, 1) we have extended the TASM language with two newly defined *Observer* and *Event* constructs in **Paper B** and, 2) we have presented an observer-based approach to requirements validation by utilizing the eTASM language to model requirements at various levels in both **Paper B** and **Paper C** and, 3) we have proposed an observer specification logic, namely EvML, as well as a newly introduced rewriting-based monitoring algorithm for EvML in **Paper D**. Our illustration applications by using a Brake-by-Wire system and a Vehicle Locking-Unlocking system, have shown that our approach can achieve the goal of requirements validation.

4.2 Future Work

In our viewpoint, this thesis work has brought possibilities to conduct further research in certain research questions that are not thoroughly addressed and could be interesting to investigate in the future. Some of these possibilities could be:

Generally speaking, a comprehensive future work task involves: 1) since the algorithms and approaches proposed in this thesis are merely formally defined or described, we are about to implement or support them in our TASM TOOLSET and, 2) we would like to include a wider industrial validation of our developed techniques in the future.

It would be of great interest to further improve the evaluation part of our trace link creation/recovery technique by providing some statistical evidence with statistical hypothesis test, which can be conducted by performing, e.g., Wilcoxon signed-rank test with Monte Carlo permutation. Moreover, we also consider determining the optimal value of the parameters in the weighting schema for combining two ranked trace link lists in the analysis. We have realized that it would also result in some interesting discoveries, by applying some user feedback technique to our context analysis. The investigation of using other automated information retrieval methods, instead of VSM, together with our context analysis is highly appreciated on our good side as well.

We would be interested in combining our proposed requirements modeling approach with a set of assistant techniques, such as rule/pattern-based algorithm in order to semi- or fully-automatically transform natural languages into eTASM models.

Furthermore, we believe that the observer-based requirements validation approach would be of more help by combining it with certain test-case gener-

ation techniques, i.e., the observers could be specified automatically.

Bibliography

[1] D. Zowghi and V. Gervasi, "The Three Cs of Requirements: Consistency, Completeness, and Correctness," in *Proceedings of REFSQ'02*, 2002.

[2] J. Hammond, R. Rawlings, and A. Hall, "Will It Work?," in *Proceedings of RE'01*, pp. 102–109, 2001.

[3] N. M. A. Munassar and A. Govardhan, "A Comparison Between Five Models of Software Engineering," *IJCSI International Journal of Computer Science Issues*, vol. 7, no. 5, pp. 94–101, 2010.

[4] B. Berenbach, F. Schneider, and H. Naughton, "The Use of A Requirements Modeling Language for Industrial Applications," in *Proceedings of RE'12*, pp. 285–290, 2012.

[5] E. Bjarnason, P. Runeson, M. Borg, M. Unterkalmsteiner, E. Engström, B. Regnell, G. Sabaliauskaite, A. Loconsole, T. Gorschek, and R. Feldt, "Challenges and Practices in Aligning Requirements with Verification and Validation: A Case Study of Six Companies," *Empirical Software Engineering*, pp. 1–47, 2013.

[6] A. T. Bahill and S. J. Henderson, "Requirements Development, Verification and Validation Exhibited in Famous Failures," *Syst. Eng*, 2005.

[7] A. Ellis, "Achieving Safety in Complex Control Systems," in *Proceedings of SCSC'95*, pp. 1–14, Springer London, 1995.

[8] N. G. Leveson, *Safeware: System Safety and Computers*. NY, USA: ACM, 1995.

[9] A. Cimatti, M. Roveri, A. Susi, and S. Tonetta, "From Informal Requirements to Property-Driven Formal Validation," in *Proceedings of FMICS'09*, pp. 166–181, Berlin, Heidelberg: Springer-Verlag, 2009.

[10] A. Iliasov, "Augmenting Formal Development with Use Case Reasoning," in *Proceedings of Ada-Europe'12*, pp. 133–146, Springer Berlin Heidelberg, 2012.

[11] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, 2008.

[12] G. Spanoudakis and A. Zisman, "Software Traceability: A Roadmap," in *Handbook of Software Engineering and Knowledge Engineering*, pp. 395–428, World Scientific Publishing, 2004.

[13] J. Cleland-Huang, O. Gotel, and A. Zisman, *Software and Systems Traceability*. Springer, 2012.

[14] M. Ouimet, *A Formal Framework for Specification-Based Embedded Real-Time System Engineering*. PhD thesis, Department of Aeronautics and Astronautics, MIT, 2008.

[15] T. Gilb and D. Graham, *Software Inspection*. Addison Wesley, 1993.

[16] S. H. Ow and K. Lumpur, "Design and Code Inspections to Reduce Errors in Program Development," in *Proceedings of COMPSAC'97*, pp. 542–547, 1997.

[17] M. E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal*, vol. 38, no. 2.3, pp. 258–287, 1999.

[18] I. Sommerville, *Software Engineering*. Ninth ed., 2011.

[19] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese, "Requirements Specification for Process-Control Systems," *IEEE Trans. Softw. Eng.*, vol. 20, pp. 684–707, Sept. 1994.

[20] J. Calvez, *Embedded Real-Time Systems*. Wiley, 1993.

[21] O. Gotel, J. Cleland-Huang, J. H. Hayes, A. Zisman, A. Egyed, P. Grünbacher, A. Dekhtyar, G. Antoniol, and J. Maletic, *The Grand Challenge of Traceability (v1.0)*, pp. 343–412. 2012.

[22] G. Salton, A. Wong, and C. S. Yang, "A Vector Space Model for Automatic Indexing," *Communications of the ACM*, vol. 18, pp. 613–620, Nov. 1975.

[23] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo, "Recovering Traceability Links between Code and Documentation," *IEEE Trans. Software Eng.*, vol. 28, no. 10, pp. 970–983, 2002.

[24] E. Börger and R. F. Stärk, *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.

[25] E. M. Clarke and J. M. Wing, "Formal Methods: State of The Art and Future Directions," *ACM Comput. Surv.*, vol. 28, pp. 626–643, Dec. 1996.

[26] M. Leucker and C. Schallhart, "A Brief Account of Runtime Verification," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009.

[27] S. Deerwester, S. Dumais, T. Landauer, G. Furnas, and L. Beck, "Improving information retrieval with latent semantic indexing," in *Annual Meeting of the American Society for Info. Science 25*, 1988.

[28] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proceedings of ICSE'03*, (Washington, DC, USA), pp. 125–135, IEEE Computer Society, 2003.

[29] A. Abadi, M. Nisenson, and Y. Simionovici, "A Traceability Technique for Specifications," in *Proceedings of ICPC'08*, pp. 103–112, 2008.

[30] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. D. Lucia, "On the Equivalence of Information Retrieval Methods for Automated Traceability Link Recovery," in *Proceedings of ICPC'10*, pp. 68–71, 2010.

[31] C. Fautsch and J. Savoy, "Adapting the tf-idf Vector-Space Model to Domain Specific Information Retrieval," in *Proceedings of SAC'10*, pp. 1708–1712, 2010.

[32] W.-K. Kong and J. H. Hayes, "Proximity-based Traceability: An Empirical Validation using Ranked Retrieval and Set-based Measures," in *Proceedings of EmpiRE'11*, pp. 45–52, 2011.

[33] A. D. Lucia, M. D. Penta, R. Oliveto, A. Panichella, and S. Panichella, "Improving IR-based Traceability Recovery Using Smoothing Filters," *International Conference on Program Comprehension*, vol. 0, pp. 21–30, 2011.

[34] J. Cleland-Huang, R. Settimi, C. Duan, and X. Zou, "Utilizing Supporting Evidence to Improve Dynamic Requirements Traceability," in *Proceedings of RE'05*, (Washington, DC, USA), pp. 135–144, IEEE Computer Society, 2005.

[35] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, "Software Traceability with Topic Modeling," in *Proceedings of ICSE'10*, pp. 95–104, 2010.

[36] D. Blei, A. Ng, and M. Jordan, "Latent Dirichlet Allocation," *Journal of Machine Learning Research*, no. 3, pp. 993–1022, 2003.

[37] A. De Lucia, R. Oliveto, and P. Sgueglia, "Incremental Approach and User Feedbacks: a Silver Bullet for Traceability Recovery," in *Proceedings of ICSM'06*, pp. 299–309, 2006.

[38] A. Mahmoud, N. Niu, and S. Xu, "A Semantic Relatedness Approach for Traceability Link Recovery," in *Proceedings of ICPC'12*, pp. 183–192, 2012.

[39] J.-R. Abrial, *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[40] A. Mashkoor, J.-P. Jacquot, and J. Souquières, "Transformation Heuristics for Formal Requirements Validation by Animation," in *Proceedings of SafeCert'09*, (York, United Kingdom), 2009.

[41] I. Cardei, M. Fonoage, and R. Shankar, "Model Based Requirements Specification and Validation for Component Architectures," in *Systems Conference, 2008 2nd Annual IEEE*, pp. 1–8, 2008.

[42] P. Scandurra, A. Arnoldi, T. Yue, and M. Dolci, "Functional Requirements Validation by Transforming Use Case Models into Abstract State Machines," in *Proceedings of SAC'12*, (NY, USA), pp. 1063–1068, ACM, 2012.

[43] OMG. `http://www.omgmarte.org/`, 2013.

[44] K. Kimbler and D. S. Birk, "Use Case Driven Analysis of Feature Interactions," in *Feature Interactions in Telecommunications Systems, IOS*, pp. 167–177, 1994.

[45] D. Amyot, L. Charfi, N. Gorse, T. Gray, L. Logrippo, J. Sincennes, B. Stepien, and T. Ware, "Feature Description and Feature Interaction Analysis with Use Case Maps and Lotos," in *Proceedings of FIW'00*, pp. 274–289, 2000.

[46] P. Sampath, S. Arora, and S. Ramesh, "Evolving Specifications Formally," in *Proceedings of RE'11*, pp. 5–14, Aug 2011.

[47] S. Arora, P. Sampath, and S. Ramesh, "Resolving Uncertainty in Automotive Feature Interactions," in *Proceedings of RE'12*, pp. 21–30, Sept 2012.

[48] T. Yue, L. C. Briand, and Y. Labiche, "A Use Case Modeling Approach to Facilitate the Transition Towards Analysis Models: Concepts and Empirical Evaluation," in *Proceedings of MODELS'09*, pp. 484–498, 2009.

[49] D. Giannakopoulou and K. Havelund, "Automata-Based Verification of Temporal Properties on Running Programs," in *Proceedings of ASE'01*, pp. 412–416, Nov 2001.

[50] H. Barringer, A. Goldberg, K. Havelund, and K. Sen, "Rule-Based Runtime Verification," in *Proceedings of VMCAI'04*, pp. 44–57, 2004.

[51] D. Basin, F. Klaedtke, S. Marinovic, and E. Zălinescu, "Monitoring of Temporal First-order Properties with Aggregations," in *Proceedings of RV'13*, pp. 40–58, 2013.

[52] M. Viswanathan, "Testing Extended Regular Language Membership Incrementally by Rewriting," in *Proceedings of RTA03*, pp. 499–514, Springer-Verlag, 2003.

[53] A. Bauer, M. Leucker, and C. Schallhart, "Monitoring of real-time properties," in *Proceedings of FSTTCS'06*, pp. 260–272, Springer, 2006.

[54] Y. Falcone, J.-C. Fernandez, and L. Mounier, "Runtime Verification of Safety-Progress Properties," in *Runtime Verification*, pp. 40–59, Berlin, Heidelberg: Springer-Verlag, 2009.

[55] B. Ramesh and M. Jarke, "Toward Reference Models for Requirements Traceability," *IEEE Trans. Softw. Eng.*, vol. 27, pp. 58–93, Jan. 2001.

[56] J. Kraft, Y. Lu, C. Norström, and A. Wall, "A Metaheuristic Approach for Best Effort Timing Analysis targeting Complex Legacy Real-Time Systems," in *Proceedings of RTAS'08*, 2008.

[57] O. C. Z. Gotel and A. C. W. Finkelstein, "An Analysis of the Requirements Traceability Problem," in *Proceedings of RE'94*, pp. 94–101, 1994.

[58] N. Ali, Y.-G. Gueheneuc, and G. Antoniol, "Trust-Based Requirements Traceability," in *Proceedings of ICPC'11*, pp. 111–120, 2011.

[59] M. Poppleton, "Towards Feature-Oriented Specification and Development with Event-B," in *Proceedings of REFSQ'07*, pp. 367–381, 2007.

[60] J. Zhou, Y. Lu, and K. Lundqvist, "A TASM-based Requirements Validation Approach for Safety-critical Embedded Systems," in *Proceedings of Ada-Europe'14*, June 2014.

[61] J. Zhou, Y. Lu, and K. Lundqvist, "Towards Feature-oriented Requirements Validation for Automotive Systems," in *Proceedings of RE'14*, August 2014.

[62] M. Shaw, "The Coming-of-Age of Software Architecture Research," in *Proceedings of ICSE'01*, (Washington, DC, USA), pp. 656–664a, IEEE Computer Society, 2001.

II

Included Papers

Chapter 5

Paper A: A Context-based Information Retrieval Technique for Recovering Use-Case-to-Source-Code Trace Links in Embedded Software Systems

Jiale Zhou, Yue Lu, Kristina Lundqvist.
Proceedings of the 39th Euromicro Conference on Software Engineering and Advanced Applications (SEAA'13), Santander, Spain, September 2013.

Abstract

Post-requirements traceability is the ability to relate requirements (e.g., use cases) forward to corresponding design documents, source code and test cases by establishing trace links. This ability is becoming ever more crucial within embedded systems development, as a critical activity of testing, verification, validation and certification. However, semi-automatically or fully-automatically generating accurate trace links remains an open research challenge, especially for legacy systems. Vector Space Model (VSM), a notably known Information Retrieval (IR) technique aims to remedy this situation. However, VSM's low-accuracy level in practice is a limitation. The contribution of this paper is an improved VSM-based post-requirements traceability recovery approach using a novel context analysis. Specifically, the analysis method can better utilize context information extracted from use cases to discover relevant source code files. Our approach is evaluated by using three different embedded applications in the domains of industrial automation, automotive and mobile. The evaluation shows that our new approach can achieve better accuracy than VSM, in terms of higher values of three main IR metrics, i.e., recall, precision, and mean average precision, when it handles embedded software applications.

5.1 Introduction

Requirements Management (RM) is a critical activity for system development. It should be carried out for all the phases of systems development life cycle (or the software development process in other words), rather than a single phase. RM assumes requirements elicitation, tracking and preservation of integrity, and handles a large amount of software development artifacts (i.e., the artifacts hereafter). The quality of RM is very important for system development, e.g., customers satisfaction, requirements coverage, efficient utilization of resources.

The heart of RM is Requirements Traceability (RT), which is defined as "the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through periods of on-going refinement and iteration in any of these phases)" [1]. RT provides critical support for system developers throughout the entire software development process. Tracing requirements can help to, but is not limited to, perform change impact analysis, risk analysis, criticality analysis, regression testing, and requirements satisfaction assessment. However, in traditional industrial practices, especially for legacy systems [2], trace links are manually established and maintained. Such activities tend to be costly to implement and are therefore perceived as financially non-viable by many companies [1, 3]. To address this problem, many efforts [4, 5, 6, 7, 8, 9, 10] have been devoted to semi-automatic or fully-automatic trace link creation. However, such creation throughout the entire systems development process, remains a challenging issue [11].

Among these efforts, the algebraic model Vector Space Model (VSM) [12] (referred to as the standard VSM hereafter), has been most investigated to build trace links between textual artifacts, such as requirements and source code [13, 14, 15]. After requirements and the target artifacts are preprocessed by e.g., removing stop words, stemming, the obtained term-document matrix is used by the standard VSM, which produces descending-ordered ranked lists of candidate trace links. Such candidate trace links contain the scores which express the similarity between requirements and subsequent artifacts, based on the occurrences of terms. Then, different strategies are applied to prune undesired links, and finally, the resulting candidate trace link lists are vetted by human analysts w.r.t. relevancy to a specific project. Nevertheless, statistical analysis [16] showed that analysts' tracing experience and amount of effort (applied to look for missing links, comfort level with tracing and so on) do not

affect the accuracy of the final trace link lists. Rather, the initial accuracy of the candidate trace link lists is the most important factor, impacting the accuracy of the final trace link lists. Our goal in this paper is to tackle the above problem by using a novel VSM-based context analysis, which involves lightweight human intervention in the early phase of the RT recovery process. In doing this, higher-accuracy candidate trace link lists are obtained when compared with the standard VSM, which also dramatically reduces the human efforts involved in the final phase of the RT recovery process.

Note that Use Case (UC) technique has been widely adopted as a Requirement Specification Language (RSL) in the embedded systems development, with the advantage of many benefits it provides [17]. In order to better illustrate our approach, we are particularly interested in establishing trace links between UCs and source code files in this work. In particular, the technical contributions of this paper are two-fold:

- We introduce the VSM-based context analysis, which consists of three steps. Specifically, the first step is to analyze the constructs of the RSL, in order to obtain the requirement intent and a set of context information. Further, the extracted context information is classified into two groups, i.e., requirements intent-positive and requirements intent-negative (cf. Section 5.3). In the second step, the requirement intent and the intent-positive context information are separately used by the standard VSM as input, which generates two trace link lists. Lastly, the two trace link lists are combined together through a weighted knowledge model, which generates the candidate trace link list.

- We show that our new approach improves the traceability accuracy of the standard VSM, by obtaining higher values of three main IR metrics, i.e., *recall*, *precision*, and *mean average precision* (MAP) scores. Typically, our case studies are three different embedded applications in the domains of industrial automation, automotive and mobile.

The remainder of the paper is organized as follows. Section 5.2 introduces the related work and background theory. Section 5.3 firstly gives an overview of our analysis, and then presents different parts of our proposed method together with the implementation of the algorithm in detail. Next, Section 5.4 that describes the evaluation setup, research questions for evaluation, evaluation metrics, improvements of analysis results as well as results validity, and finally, conclusions and future work are drawn in Section 5.5.

5.2 Background

This section firstly describes the related work in Section 5.2.1, and then illustrates the trace link recovery process based on Information Retrieval (IR) techniques in Section 5.2.2, which is followed by an introduction about context-based analysis in Section 5.2.3.

5.2.1 Related Work

Many recent studies have explored the feasibility of different IR methods for semi-automatically or fully-automatically recovering trace links between requirements and subsequent artifacts. Deerwesteret al. [4] discuss the effectiveness of Latent Semantic Indexing (LSI) for recovering trace links between different kinds of artifacts, and Marcuset al. [18] further the work, showing a promising result over VSM. Abadiet al. [14] present a novel IR technique based on Jensen & Shannon (JS) model. They also compared JS, VSM and LSI for traceability recovery purpose, and concluded that VSM and JS are the best-fits. A similar comparison is also conducted by Olivetoet al. [19], which showed that for building trace links between requirements and source code, the results of JS, VSM and LSI are almost equivalent.

Variants of basic IR methods have been proposed to improve the accuracy of IR-based traceability recovery approaches. Fautschet al. [15] present four extensions to the classical *tf-idf* VSM model. The basic idea is to retrieve domain specific information. Konget al. [8] present a VSM enhancement using term location. By utilizing the relationship between words in different textual documents, a better accuracy was achieved. Luciaet al. [9] present the approach that uses smoothing filter to improve the input in the IR-based traceability recovery process. Specifically, the words that contribute less information but repeatedly occur in the documents, are removed. Cleland-Huanget al. [6] introduce three accuracy enhancement strategies, which are hierarchical modeling, logical clustering of artifacts, and semi-automatic pruning of the probabilistic network. It should be pointed out that our approach is a variant of IR methods, but it is very different from the strategies proposed in the prior work. In their work [6], though context information is obtained from the artifact hierarchy to improve traceability results, it is not the type of "context" defined in our work. In our case, such context information is the title, pre-condition, and post-condition that are extracted from the requirements (i.e., UCs), which can contribute greatly to improve the traceability results.

Some other pieces of work also show us another promising perspec-

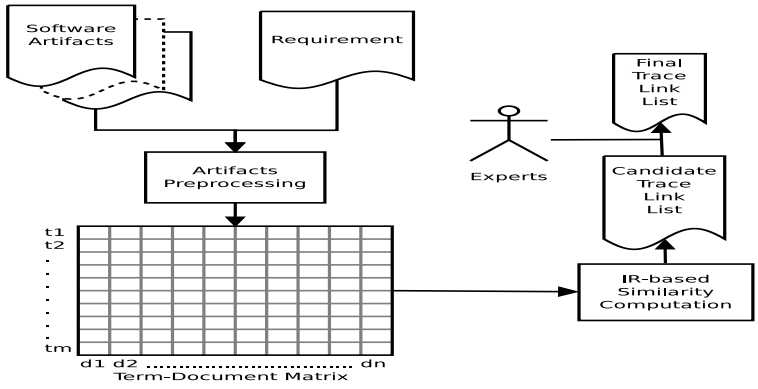

Figure 5.1: An IR-based traceability recovery process.

tive. Asuncionet al. [20] apply Topic Modeling technique, featured by Latent Dirichlet Allocation (LDA) [5] to capture trace links prospectively. Luciaet al. [21] discuss the feasibility of using user feedback analysis to improve the accuracy of the results of traceability recovery tools. Mahmoudet al. [10] propose a semantic relatedness approach that exploits external knowledge sources, e.g., Wikipedia, to identify a set of relevant terms that are used to expand the query, in an attempt to improve traceability results.

5.2.2 IR-based Traceability Recovery

The IR-based traceability recovery aims at utilizing IR techniques to compare a set of source artifacts as queries (e.g., requirements), against another set of target artifacts e.g., source code files, and calculate the textual similarities of all possible pairs of artifacts. The textual similarity between two artifacts is based on the occurrences of terms (words) within the artifacts contained in the repository. Pairs with a similarity score lower than a certain threshold (usually defined based on engineers' experience) are filtered out, and the reserved pairs form the candidate trace link list. The ranked list of candidate trace links are then analyzed by software engineers to decide if such links are true positive or not. Typically, an IR-based traceability recovery process follows the steps depicted in Figure 5.1.

The artifacts have to be preprocessed before they are used to compute similarity scores. The preprocessing of the artifacts includes a text normaliza-

tion by removing most non-textual tokens (e.g., operators, punctuations) and splitting compound identifiers into separate words by using the underscore or Camel Case splitting heuristic. Furthermore, common terms, referred to as "stop words" (e.g., articles, prepositions and programming language keywords), which contribute less to the understanding about artifacts, are also discarded by using a stop word filter. Words with the length less than a defined threshold are also pruned out. In addition, stemmer is commonly used to perform a morphological analysis, which reduces the inflected words to their root, e.g., returning verb conjugations and removing plural nouns.

After preprocessing, an artifact (e.g., a UC requirement, a source code file) can be represented as a plain document containing a list of terms (in this paper, we use *documents* and *artifacts* interchangeably). The extracted terms are generally stored in a $m \times N$ matrix (called term-by-document matrix), where m is the number of all the terms that occur in all the documents, and N is the number of documents in the corpus. A generic entry $w_{i,j}$ of the matrix denotes a measure of the relevance of the i_{th} term in the j_{th} document. Based on the term-by-document matrix representation, different IR methods can be used to calculate textual similarities between paired artifacts.

Particularly, in Vector Space Model (VSM) [13], given the entire collection of unique terms $T = \{t_1, \ldots, t_m\}$ in a corpus with N documents, the document d_n is represented as a vector $d_n = \{w_{1,d_n}, \ldots, w_{m,d_n}\}$ consisting of m unique terms from the corpus with an assigned weight w_{i,d_n} through a certain weighting scheme. Therefore, the similarity score, denoted as $sim(q, d)$, between the query document q and the target document d is calculated by using the cosine of the angle between their vectors:

$$sim(q,d) = \frac{\sum_{i=1}^{m} w_{i,q} \cdot w_{i,d}}{\sqrt{\sum_{i=1}^{m} w_{i,q}^2 \cdot \sum_{i=1}^{m} w_{i,d}^2}} \tag{5.1}$$

Next we introduce the *term frequency-inverse document frequency*, i.e., *tf-idf*, which is adopted as the weighting scheme by all the VSM-based approaches (including our method).

$$w_{i,q} = tf_i(q) \cdot idf_i, \quad w_{i,d} = tf_i(d) \cdot idf_i \tag{5.2}$$

where $tf_i(q)$ and $tf_i(d)$ are measured by the number of times the term t_i occurs in the query document q and the target document d respectively, and idf_i is computed as $log(\frac{N}{df_i})$, where df_i is the number of documents containing the term t_i.

The standard VSM described above has been applied to the requirements traceability recovery process [12]. In our work, the corpus is the entire set of requirements (i.e., UCs) and source code files. In applying the standard VSM, we select a UC (as the query q) and repeatedly calculate the similarity scores between the UC and all the source code files in the corpus. In this way, a descending-ordered ranked list of candidate trace links to the requirement will be generated by VSM. However, the low-level accuracy of the standard VSM in practice is a limitation [22]. From our viewpoint, the main reason is that the standard VSM takes the whole requirement document as its input, regardless of the relevance of the terms associated with the intent of the requirement. Therefore, it is inevitable that the majority of the terms used by VSM are irrelevant ones, hence misleading VSM to produce a very low-accuracy trace link list. Clearly, one possible improvement is to provide VSM with more relevant terms that can better represent the requirement intents, which are obtained through the context-based analysis introduced in the following section.

5.2.3 Context-based Analysis

Connolly [23] introduces an important fact about communication: Communication always takes place in a context. Suppose that we are interested in studying an intent of others, which we denote as I. The context consisting of whatever constructs surround I, serves to facilitate the communication between speakers and listeners. Furthermore, the intents of people can be reflected by the context of the conversation between them, based upon their understanding and interpretation. Accordingly, if we regard a pair of a requirement and its subsequent artifacts as "speaker" and "listeners", between which there is a successful communication, then context information surrounding speaker's intent is helpful to recover the pair, i.e., the trace link between the requirement and its subsequent artifacts.

In order to have a better understanding about the idea of using context-based analysis to improve the accuracy of the standard VSM, we give the following example of a little boy buying ice cream: A little boy wanted to eat an ice cream, so he wrote the words "ice cream" on his mother's shopping list. Later on, two actions took place and can be documented as: 1) his mother went to a shop and bought an ice cream; 2) he updated the status of his Twitter with the statement "Ice cream is my favorite dessert!". From the perspective of IR-based traceability recovery, we can consider his writing ice cream on the shopping list as the requirement. The two documented actions can be regarded as the subsequent artifacts, i.e., documents. In this case, both

documents are kind of related with the requirement, because all of the requirement and documents share the term “ice cream”. Thereby, it is very hard to conclude which document has the higher similarity score with the requirement by using the standard VSM. Nevertheless, if we perform context analysis on the requirement, we can find that the requirement intent is “eat an ice cream” and the requirement context is “shopping list”. Obviously, the first document and the requirement share the implicit information “shop”. Therefore, the first document is more relevant to the requirement about “eat an ice cream”, when the context analysis is applied. By using the above intuitive idea, we propose a novel VSM-based context analysis, which can obtain better traceability results, when compared with the standard VSM.

5.3 The Proposed VSM-Based Context Analysis Method

A functional requirement is a need that a particular product or process must be able to perform. It can also be regarded as one or a set of *intents*, of which the detailed interpretations are subsequent software development artifacts, e.g., design documents, source code, testing and maintenance documents. Accordingly, the traceability recovery process is about finding different relevant interpretations of such intents. In matters of interpretation, it is very important to understand the context. Since context not only plays a significant role in influencing the way that the intent is interpreted with a certain level of satisfaction, but also is a construct, helping project experts to improve the accuracy of traceability recovery process. In this work, such useful requirements context is extracted and used in the trace link recovery process.

In the following, we introduce the proposed VSM-based context analysis for the post-requirements traceability recovery process in detail. Firstly, an overview of the analysis is given in Section 5.3.1, which is followed by the description of the context analysis of Use Case (UC) requirements in Section 5.3.2. Next Section 5.3.3 describes the weighted knowledge model that we used to combine the two generated trace link lists. In order to better illustrate our approach, in this work, we are particularly focused on establishing trace links between UCs and source code files. Some other interesting types of RSL will be considered as part of our future work.

5.3.1 Overview of the VSM-based Context Analysis

The main idea of our proposed VSM-based context analysis is to utilize the context information to enhance the accuracy of the candidate trace link list that will be vetted by experts at the end of the RT recovery process. To be specific, its basic approach is summarized by the following three steps:

- The first step is to obtain the context information and the requirement intent, by analyzing the constructs of the RSL based upon experts' experience. The extracted context information will be further classified into two groups i.e., intent-positive and intent-negative, according to whether or not such context information can help to communicate and understand the requirement intent. This is the core part of our context analysis.

- After the context analysis, we employ the standard VSM to generate two ranked trace link lists between the requirement and artifacts, in terms of using the intent-positive context query and the intent query (as input), respectively. In doing this, we will get two candidate trace link lists at the end of this step.

- Finally, the two generated ranked lists are combined together to form a ranked candidate trace link list containing the recalculated similarity scores. This is done by using a *weighted knowledge model*, and the ranked candidate list will be vetted by experts to produce the final trace link list.

Figure 5.2 shows the detailed work flow of our approach. Note that our approach involves lightweight human intervention in the early phase of the RT recovery process, which is very different from the traditional way that heavyweight human intervention is often involved in the last stage of the RT recovery process. In doing this, we provide the standard VSM with more accurate information to generate better candidate trace link lists. As a result, the final trace link lists can be significantly improved as well as the pertaining human efforts demanded in the final phase of the RT recovery process can be dramatically reduced.

5.3.2 Context Analysis of Use Cases

Requirement context consists of whatever constructs surround requirement intent, which are relevant to the requirement interpretation on the subsequent

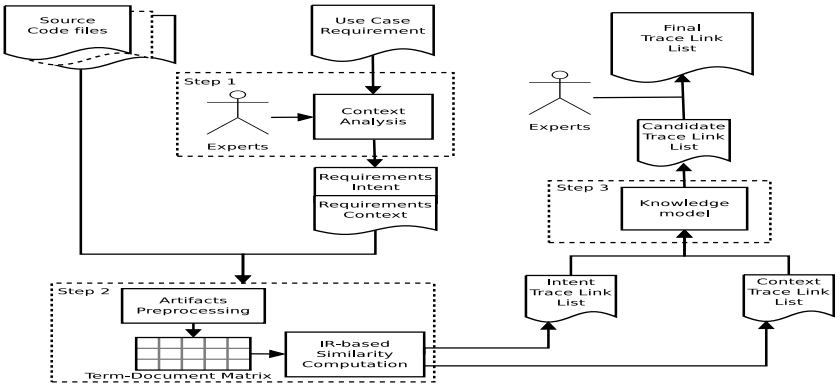

Figure 5.2: The work flow about our proposed VSM-based context analysis approach.

artifacts. Moreover, there are various constructs which are contained by different RSLs, introducing great diversity. In order to avoid the case that the constructs of requirements context become too intractable to be processed in practice, it is essential to restrict the range. Therefore, our context analysis defines the range of context and intent constructs for a given RSL, based around a set of proposed criteria and definitions using experts' experience. Additionally, since the results of our context analysis can be reused for the projects using the same RSL, we consider the human efforts involved in our approach can still be regarded as light-weighted.

The criterion, which we apply to judge if a construct belongs to context or intent, is given below:

Criterion 1. *A construct is considered as the requirement intent if it is used by the system developers to implement the functionalities of a system described by the requirement; it belongs to context otherwise.*

In this work, we choose UC technique as an example, to examine our approach. In general, although different projects or companies may use different structured UC requirements, the constructs of the UCs as in the widely accepted industrial practice [17], can be expressed by Definition 1.

Definition 1. *Use Case constructs of our interest include the title, preconditions, flow of events and post-conditions of the use case.*

It is interesting to stress that our current research only focuses on the functional requirements which describe the functionalities of a system. The above definition should thereby be adapted when non-functional requirements are considered.

According to Criterion 1, we give the following definitions of UC intent and UC context used in our work.

Definition 2. *Intent of a Use Case requirement refers to its flow of events.*

Definition 3. *Context of a Use Case requirement refers to its title, pre-condition, and post-condition.*

Further, our definitions are given based around the following train of thoughts:

- The title of a UC is traditionally named as an active verb phrase. Although its information is not rich enough for system developers to implement the functional requirement, it still can help to provide extra information for traceability recovery process. Therefore, the title construct is defined as context information.
- The initiation of a UC occurs whenever the pre-conditions are met, and the post-conditions, on the other hand, describe what data need to be stored in the UC. Therefore, they belong to context.
- The flow of events is the main part of a UC, which defines the relevant functional requirements with all the details. Hence it is considered as the intent of a UC.

However, from the traceability perspective, not all the context information is helpful for the interpretation of requirement intent, we thereby give the following definition of

Criterion 2. *If the construct aims at describing the functionalities of a system, i.e., the requirement intent, the construct is defined as intent-positive. If the construct aims at describing the constraints, extensions, meta information, etc., the construct is defined as intent-negative.*

Based on Criterion 2, the pre-conditions and post-conditions should be classified as intent-negative constructs, since they do not aim at describing the functions of a system. The title of UC is usually associated with some information about a certain functional requirement, it is thereby regarded as intent-positive construct.

5.3.3 The Weighted Knowledge Model

Getherset al.present the weighted knowledge model in [24], which is used in our work. Next we introduce its basic idea in our context: The two trace link lists generated by using requirements context query and requirements intent query (i.e., the context query and the intent query hereafter) are viewed as two knowledge sources, both of which can contribute greatly to address trace link recovery between the requirement and a set of target artifacts. Since the two trace link lists express their judgments from different perspectives (i.e., as either requirements context or requirements intent), their pertaining weights should be considered when they are combined together to obtain a more accurate candidate trace link list.

Formally, such a combination is obtained through two steps. At the first step, the two set of similarity scores of two trace link lists are normalized by using a standard normal distribution, as expressed by Equation 5.3:

$$sim_{l_i}(q,d)_n = \frac{sim_{l_i}(q,d) - mean(sim_{l_i}(q,D))}{stdev(sim_{l_i}(q,D))} \tag{5.3}$$

where q represents the query, D represents a set of related artifacts, $d \in D$, $sim_{l_i}(q,d)_n$ is the normalized similarity score of $sim_{l_i}(q,d)$ (where l_i is one of the trace link lists), and $sim_{l_i}(q,D)$ is a set of similarity scores in l_i. The functions $mean()$ and $stdev()$ return the mean and standard deviation of the similarity scores of two trace link lists respectively. Note that such normalization is required to guarantee that the two different sets of similarity scores are commensurable.

At the second step, the normalized scores are combined by using the following weighted knowledge model, as shown in Equation 5.4:

$$sim(cq,iq,d)_c = \lambda \times sim_{l_i}(cq,d)_n + (1-\lambda) \times sim_{l_j}(iq,d)_n \tag{5.4}$$

where cq and iq represent the context query and intent query, $\lambda \in [0,1]$ expresses the confidence in each query. The higher the value the higher confidence gives by the technique. In our evaluation, we find the value of λ to be 0.3 (as the weight of context query), which usually produces good combined similarity scores.

In a nutshell, our context analysis in practice provides the standard VSM with both requirements intent query and requirements context query. Typically, such a combination contains more enhanced semantics, which can accurately represent the intent of the requirement. As a result, the accuracy of the standard VSM toward recovering true trace links can be improved.

5.3.4 Algorithm Description

The precise description of the algorithm using pseudo-code is outlined in Algorithm 1, which takes four parameters and returns a ranked list of candidate trace links at the end of its execution.

Parameters:

D: list - the collection of source code files D

uc: string - the UC requirement uc

iq: string - the intent query iq of the use case uc

cq: string - the context query cq of the use case uc

Returns:

$LIST_{vsm-ca}$: list - the ranked candidate trace link list between the UC requirement uc and the source code files D, containing the combined similarity scores.

Algorithm 1 $VSM - CA(D, uc, iq, cq)$

```
1  m ← 0, D' ← ∅
2  D ← d_1, d_2, ..., d_{n-1}, d_n
3  cq ← context(uc)
4  iq ← intent(uc)
5  for all d_i ∈ D such that 1 ≤ i ≤ n do
6     sim_{cq,d_i} ← sim(cq, d_i)
7     sim_{iq,d_i} ← sim(iq, d_i)
8     if sim_{cq,d_i} ≠ 0 or sim_{iq,d_i} ≠ 0 then
9        m ← m + 1
10       D' ← D' ⋃ {d_i}
11    end if
12 end for
13 for all d'_i ∈ D' such that 1 ≤ i ≤ m do
14    sim_{cq,d'_i} ← sim_{l_1}(cq, d'_i)_n
15    sim_{iq,d'_i} ← sim_{l_2}(iq, d'_i)_n
16    sim(uc, d'_i) ← sim(cq, iq, d'_i)_c
17    if sim(uc, d'_i) ≥ threshold then
18       LIST_{vsm-ca} ← LIST_{vsm-ca} ⋃ {sim(uc, d'_i)}
19    end if
20 end for
21 return LIST_{vsm-ca}
```

5.4 Empirical Evaluation

This section describes the evaluation carried out to assess the improvement given by our VSM-based context analysis over the standard VSM, comprising

six parts. Section 5.4.1 introduces the evaluation setup, and Section 5.4.2 formulates our research questions for evaluation. Our evaluation metrics and the corresponding results are presented in Section 5.4.3 and Section 5.4.4 respectively. Finally, we summarize the evaluation by highlighting some interesting observations in Section 5.4.5, before we give our view of validity of results in regard to some possible threats in Section 5.4.6.

5.4.1 Definitions and Context

The goal of the evaluation is to provide the evidence that our VSM-based context analysis can obtain better results over the standard VSM, when it is used for trace link recovery in embedded applications.

The evaluation consists of three different embedded software applications developed at Mälardalen University in Sweden, which are: 1) one industrial robotic control system *iRobot* and, 2) one truck navigation system *iTruck* and, 3) one embedded mobile application *iSudoku*. Specifically, iRobot is a C program, which models a robotic control application containing complicated timing behavior, and it has been designed and evaluated in [25]. iTruck is also a C program, which is developed for modeling a truck navigation system by using SaveComp Component Model (SaveCCM) [26]. iSudoku is a Sudoku game application developed in Java for the Android platform. In addition, all the three applications have different number of UCs, source code files and true trace links, as shown in Table 5.1. One example of the UCs in iSudoku, with its title, pre-condition, flow of events and post-condition, is shown by Figure 5.3. Moreover, all the relevant files of the three case studies, e.g., UCs, source code files, are available upon request.

For implementation, we use the Lucene library [27], which is the well-known VSM with tf-idf weighting scheme, and has been considered as the default IR model by many pieces of work [14, 15, 20]. Our testbed is running Mac OS X, version 10.6.8, and the computer is equipped with the Intel Core Duo CPU i7 processor, 4GB RAM and a 256KB L2 Cache. The processor has four cores and one frequency level: 2.2 GHz.

5.4.2 Research Questions

In this study, we aim at addressing the following research question:

- Can our VSM-based context analysis approach obtain better quality trace link lists, compared with the standard VSM?

Table 5.1: Characteristics of the three different embedded software applications used in our evaluation.

System	KLOC	UCs	Source code files	True links
iRobot	2706	21	20	45
iTruck	952	24	14	37
iSudoku	9285	18	54	51

```
1 List Puzzles on Screen Use Case
2   Pre-conditions:
3     The game is initiated.
4   Flow of events:
5     The application loads all the puzzles stored in the local database.
6     The puzzles and their corresponding folders can be listed on the
7     screen one by one.
8   Post-conditions:
9     List variables is initiated.
```

Figure 5.3: An example shows one use case in iSudoku.

To answer this question, we plan to use three well-known IR metrics for results comparison (to be introduced in the following section).

5.4.3 Metrics

There are many different measures for evaluating the accuracy of IR methods. In this work, we use three well-known IR metrics, i.e., recall, precision and mean average precision (MAP) [22]. Specifically, *recall* shows the ratio of the number of relevant documents retrieved by the method over the total number of relevant documents, and 100% recall means that all relevant documents were retrieved. *Precision* is the fraction of the relevant documents retrieved over the total number of the retrieved documents, and 100% precision means that all the retrieved documents are relevant ones, though there could be some relevant links that were not discovered. Recall and precision can be expressed by Equation 5.5 and Equation 5.6 as follows:

$$recall = \frac{|D_{rel} \cap D_{ret}|}{|D_{rel}|} \tag{5.5}$$

where D_{rel} represents the collection of source code files that are relevant to a UC, and D_{ret} is the collection of the retrieved source code files using a certain

IR technique.

$$precision = \frac{|D_{rel} \cap D_{ret}|}{|D_{ret}|} \tag{5.6}$$

where D_{rel} represents the collection of source code files which are relevant to a UC, and D_{ret} is the collection of the retrieved source code files.

Another evaluation metric MAP, as one of the most frequently used IR measures, considers the rank of the retrieved trace links. The higher the MAP score is, the better quality of the retrieved ranked list of trace links is, in terms of requirements relevance. In particular, given a collection of UCs as queries Q, and a set of related source code files as documents D_a, the MAP score of the ranked list L of retrieved documents for the given query q, is defined as below:

$$MAP = \frac{1}{|Q|} \sum_{q=1}^{Q} \frac{1}{|D_a|} \sum_{d=1}^{D_a} SCORE_{rank}(d, L) \tag{5.7}$$

where $SCORE_{rank}(d, L)$ is the ranking score of the document d in the list L. The higher the rank of d is, the larger ranking score the document has.

5.4.4 Analysis of Results

In this section, we investigate whether the accuracy of our VSM-based context analysis approach is superior to that of the standard VSM approach.

Improvement of Recall and Precision Scores given by our VSM-based Context Analysis

Recall and *precision* frequently exist in a state of mutual tension. For instance, 100% recall can be achieved simply by returning all possible links. This may result in a very low level of precision, which is not so useful in practice [22]. When choosing any traceability recovery approach, the end-users should consider which one between *recall* or *precision* is preferred. For example, for safety-critical projects, *recall* will probably be more important than *precision*, since the end-users will not want to run the risk of missing any true links. On the other hand, in most cases, a non-safety critical project with a short time-to-market may prefer to favor *precision* [22], i.e., the end-users would expect that the traceability recovery approach can obtain more true links when they just have time to check a part of the candidate trace link list. It should be pointed

out that the level of recall at 90% is a common choice at which the precision scores are compared to show improvements [6, 24]. Figure 5.4 provides the precision/recall curves achieved by our approach and the standard VSM. As shown in the figure, we have one case (i.e., iRobot) where our approach outperforms for all levels of recall, and there are two other cases (i.e., iSudoku and iTruck) where the precision scores of our approach are much better than that acquired by the standard VSM approach when the level of recall is lower than 90%. This means that our approach would be more suitable for the embedded systems which prefer *precision* rather than *recall*.

Improvement of MAP Scores given by our VSM-based Context Analysis

Table 5.2 shows the improvement of MAP scores given by our method over the standard VSM, for all the three case studies. It is also interesting to stress that we only show the average of the MAP scores of all the UCs for a certain case study in the table, for the sake of space. Moreover, we also present such improvements in terms of percentages (i.e., $\frac{MAP_{vsm-ca}-MAP_{vsm}}{MAP_{vsm}}$, where vsm-ca is our VSM-based context analysis) in Column *Imprv. %* in Table 5.2. As shown in the table, the most significant improvement achieved by our approach is 28.0%, comparing the standard VSM.

Table 5.2: Our proposed VSM-based context analysis can retrieve higher MAP scores, comparing the standard VSM.

System	AVG of MAP VSM	AVG of MAP VSM-CA	Imprv. %
iRobot	0.717	0.733	2.23%
iTruck	0.660	0.762	15.4%
iSudoku	0.547	0.700	28.0%

5.4.5 Experiments Summary

Summarizing the above observations, our evaluation results have confirmed the following points:

- Our proposed approach can help to recover more accurate trace links than the standard VSM, in the sense of obtaining higher precision scores corresponding to certain levels of recall.

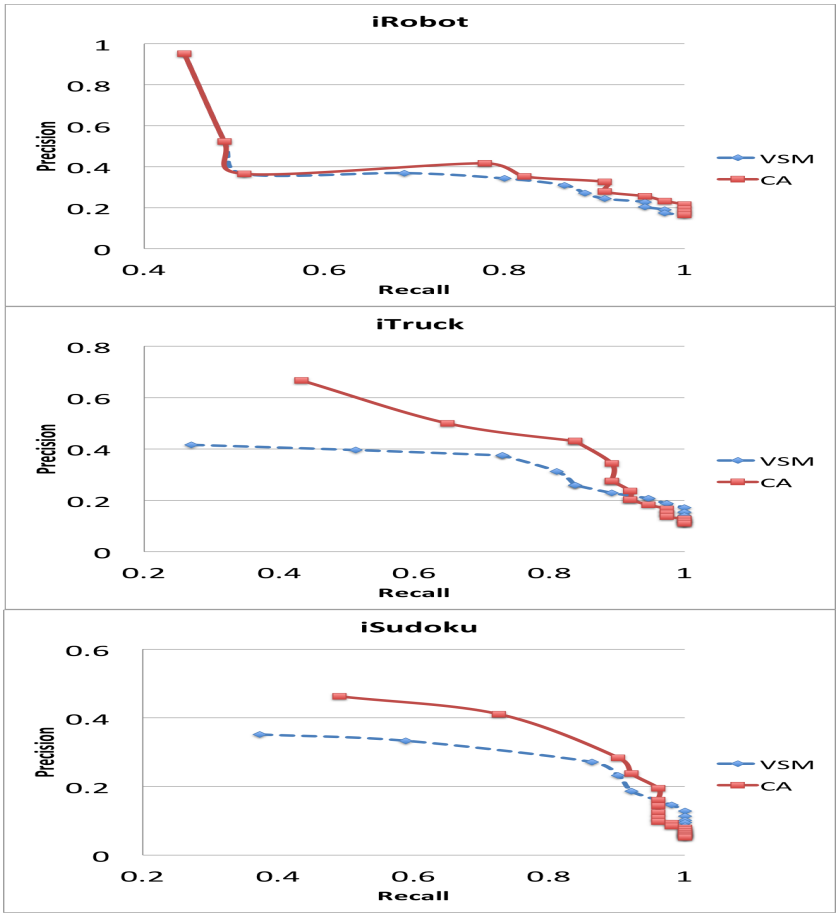

Figure 5.4: The precision/recall curves of our approach and the standard VSM approach, in the order of iRobot, iTruck and iSudoku from top to bottom. In addition, the curves in dashed lines are for the standard VSM.

- Our proposed approach can help to recover more accurate trace links than the standard VSM, in the sense of obtaining higher MAP scores.
- The computing time required by the trails of our approach, on average took only a few minutes to compute. This is an important step toward handling real life-scale requirements traceability problems.

5.4.6 Threats to Validity

In this section, we discuss the threats that can impact on the validity of our evaluation, from the following four perspectives [28]. The first category is *construct validity*, concerning the degree to which the study metrics accurately measure the concepts. The metrics used in our evaluation, i.e., *recall*, *precision* and *MAP*, have been widely adopted for assessing the traceability accuracy of IR methods. Therefore, we believe that they can sufficiently quantify the accuracy of two compared IR methods. The second category is *internal validity*, referring to the extent to which a treatment changes what is measured in the experiment. The internal validity of our experiment can only be affected by the chosen value of the parameter λ in the employed knowledge model. We choose the value of λ based on our empirical evidence. In the future, we will obtain the optimal value of λ by using some advanced techniques, such as optimization and machine learning. The third category is *external validity*, related to the extent to which we can generalize the study results. The reason is that different systems with various requirements and subsequent artifacts may lead to different results. In order to reduce the threats to the external validity, we have chosen three embedded software applications in different domains. Last but not least, the fourth category *conclusion validity* concerns if our evaluation observations can be supported by some valid statistical techniques as evidences. In this stage, we just visualize our results to illustrate our improvement. In the future, this will be done by using certain non-parametric or parametric statistical tests, such as Wilcoxon signed-rank test and ANOVA.

5.5 Conclusions and Future Work

In this paper, we have proposed a new Vector Space Model (VSM)-based approach for post-requirements traceability recovery, which uses a novel context analysis. Specifically, our approach utilizes context information featured by requirement context and requirement intent, to build trace links between use cases and relevant source code files, through a weighted knowledge model. We

have evaluated the approach by using three different embedded applications in industrial automation, automotive and mobile. The experiment results have shown that our approach can obtain better quality candidate trace link lists, in terms of higher scores of three main IR metrics, i.e., *recall*, *precision*, and *MAP*, comparing the standard VSM approach.

For future work, we will improve the evaluation part by providing some statistical evidence with statistical hypothesis test, which can be conducted by performing, e.g., Wilcoxon signed-rank test with Monte Carlo permutation. Moreover, we will consider to determine the optimal value of the parameters in the weighting schema for combining two ranked trace link lists in the analysis. We also plan to apply some user feedback technique to our context analysis, which would also result in some interesting discoveries. The investigation of using other automated information retrieval methods, instead of VSM, together with our context analysis, as well as the completion of some extensive evaluation by using more datasets are also highly appreciated on our good side.

Acknowledgements

This work was partially supported by the Swedish Research Council (VR), Stråal Säkerhets Myndigheten (SSM) and School of Innovation, Design and Engineering, Mälardalen University.

Bibliography

[1] O. C. Z. Gotel and A. C. W. Finkelstein, "An Analysis of the Requirements Traceability Problem," in *Proceedings of RE'94*, pp. 94–101, 1994.

[2] J. Kraft, Y. Lu, C. Norström, and A. Wall, "A Metaheuristic Approach for Best Effort Timing Analysis targeting Complex Legacy Real-Time Systems," in *Proceedings of RTAS'08*, 2008.

[3] B. Ramesh and M. Jarke, "Toward reference models for requirements traceability," *IEEE Trans. Softw. Eng.*, vol. 27, pp. 58–93, Jan. 2001.

[4] S. Deerwester, S. Dumais, T. Landauer, G. Furnas, and L. Beck, "Improving information retrieval with latent semantic indexing," in *Annual Meeting of the American Society for Info. Science 25*, 1988.

[5] D. Blei, A. Ng, and M. Jordan, "Latent Dirichlet Allocation," *Journal of Machine Learning Research*, no. 3, pp. 993–1022, 2003.

[6] J. Cleland-Huang, R. Settimi, C. Duan, and X. Zou, "Utilizing supporting evidence to improve dynamic requirements traceability," in *Proceedings of RE'05*, (Washington, DC, USA), pp. 135–144, IEEE Computer Society, 2005.

[7] N. Ali, Y.-G. Gueheneuc, and G. Antoniol, "Trust-Based Requirements Traceability," in *Proceedings of ICPC'11*, pp. 111–120, 2011.

[8] W.-K. Kong and J. H. Hayes, "Proximity-based Traceability: An Empirical Validation using Ranked Retrieval and Set-based Measures," in *Proceedings of EmpiRE'11*, pp. 45–52, 2011.

[9] A. D. Lucia, M. D. Penta, R. Oliveto, A. Panichella, and S. Panichella, "Improving ir-based traceability recovery using smoothing filters," *International Conference on Program Comprehension*, vol. 0, pp. 21–30, 2011.

[10] A. Mahmoud, N. Niu, and S. Xu, "A semantic relatedness approach for traceability link recovery.," in *Proceedings of ICPC' 12*, pp. 183–192, 2012.

[11] O. Gotel, J. Cleland-Huang, J. H. Hayes, A. Zisman, A. Egyed, P. Grnbacher, A. Dekhtyar, G. Antoniol, and J. Maletic, *The Grand Challenge of Traceability (v1.0)*, pp. 343–412. Springer-Verlag London Limited, 2012.

[12] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo, "Recovering Traceability Links between Code and Documentation," *IEEE Trans. Software Eng.*, vol. 28, no. 10, pp. 970–983, 2002.

[13] G. Salton, A. Wong, and C. S. Yang, "A vector space model for automatic indexing," *Communications of the ACM*, vol. 18, pp. 613–620, Nov. 1975.

[14] A. Abadi, M. Nisenson, and Y. Simionovici, "A Traceability Technique for Specifications," in *Proceedings of ICPC'08*, pp. 103–112, 2008.

[15] C. Fautsch and J. Savoy, "Adapting the tf-idf Vector-Space Model to Domain Specific Information Retrieval," in *Proceedings of SAC'10*, pp. 1708–1712, 2010.

[16] A. Dekhtyar, O. Dekhtyar, J. Holden, J. H. Hayes, D. Cuddeback, and W.-K. Kong, "On human analyst performance in assisted requirements tracing: Statistical analysis," in *Proceedings of RE' 11*, pp. 111–120, 2011.

[17] E. Nasr, J. McDermid, and G. Bernat, "Eliciting and specifying requirements with use cases for embedded systems," in *Proceedings of WORDS'02*, pp. 350–, 2002.

[18] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proceedings of ICSE'03*, (Washington, DC, USA), pp. 125–135, IEEE Computer Society, 2003.

[19] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. D. Lucia, "On the Equivalence of Information Retrieval Methods for Automated Traceability Link Recovery," in *Proceedings of ICPC'10*, pp. 68–71, 2010.

[20] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, "Software Traceability with Topic Modeling," in *Proceedings of ICSE' 10*, pp. 95–104, 2010.

[21] A. De Lucia, R. Oliveto, and P. Sgueglia, "Incremental approach and user feedbacks: a silver bullet for traceability recovery," in *Proceedings of ICSM'06*, pp. 299–309, 2006.

[22] C.-H. Jane, G. Orlena, and Z. Andrea, *Software and Systems Traceability*. Springer, 2012.

[23] J. H. Connolly, "Context in functional discourse grammar," *Alfa : Revista de Linguística*, vol. 51, pp. 11–33, 2009.

[24] M. Gethers, R. Oliveto, D. Poshyvanyk, and A. D. Lucia, "On integrating orthogonal information retrieval methods to improve traceability recovery," in *Proceedings of ICSM'11*, pp. 133–142, Sept. 2011.

[25] Y. Lu, T. Nolte, I. Bate, and L. Cucu-Grosjean, "A statistical response-time analysis of real-time embedded systems," in *Proceedings of RTSS'12*, pp. 351–362, December 2012.

[26] M. åkerholm, J. Carlson, J. Fredriksson, H. Hansson, J. Håkansson, A. Möller, P. Pettersson, and M. Tivoli, "The save approach to component-based development of vehicular systems," *Journal of Systems and Software*, vol. 80, pp. 655–667, May 2007.

[27] E. Hatcher, O. Gospodnetic, and M. McCandless, *Lucene in Action*. 2nd revised ed., 2010.

[28] B. Kitchenham, L. Pickard, and S. L. Pfleeger, "Case studies for method and tool evaluation," *IEEE Softw.*, vol. 12, pp. 52–62, July 1995.

Chapter 6

Paper B: A TASM-based Requirements Validation Approach for Safety-critical Embedded Systems

Jiale Zhou, Yue Lu and Kristina Lundqvist.
Proceedings of the 19th Ada-Europe International Conference on Reliable Software Technologies (Ada-Europe'14), Paris, France, June 2014.

Abstract

Requirements validation is an essential activity to carry out in the system development life cycle, and it confirms the completeness and consistency of requirements through various levels. Model-based *formal methods* can provide a cost-effective solution to requirements validation in a wide range of domains such as safety-critical applications. In this paper, we extend a formal language Timed Abstract State Machine (TASM) with two newly defined constructs *Event* and *Observer*, and propose a novel requirements validation approach based on the extended TASM. Specifically, our approach can: 1) model both functional and non-functional (e.g., timing and resource consumption) requirements of the system at different levels and, 2) perform requirements validation by utilizing our developed toolset and a model checker. Finally, we demonstrate the applicability of our approach in real world usage through an industrial case study of a Brake-by-Wire system.

6.1 Introduction

With the growing complexity of safety-critical systems, requirements are no longer merely specified at the outset of the systems development life cycle (SDLC). On the contrary, there is a continuum of requirements levels as more and more details are added throughout the SDLC, which can roughly be divided into two categories in terms of high-level and low-level requirements [1]. High-level requirements describe what features the proposed system has (i.e., features hereafter) and low-level requirements state how to develop such a system (i.e., requirements hereafter). Studies have revealed that most of the anomalies discovered in late development phases can be traced back to hidden flaws in the requirements [2] [3], such as contradictory or missing requirements, or requirements that are discovered to be impossible to satisfy features at the late phase of development. For this reason, requirements validation is playing a more and more significant role in the development process, which confirms the correctness of requirements, in the sense of consistency and completeness [4]. In details, consistency refers to situations where a specification contains no internal contradictions in the requirements, while completeness refers to situations where the requirements must possess two fundamental characteristics, in terms of neither objects nor entities are left undefined and the requirements can address all of the features.

In order to increase the confidence in the correctness of the requirements, model-based *formal methods* techniques have been to a large extend investigated into the field of requirements validation [5] [6]. In these techniques, the system design derived from requirements is often specified in terms of analyzable models at a certain level of abstraction. Further, features are formalized into verifiable queries or formulas and then fed into the models to perform model checking and/or theorem proving. In this way, the requirements are reasoned about to resolve contradictions, and it is also verified that they are neither so strict to forbid desired behaviors, nor so weak to allow undesired behaviors. However, such *formal methods* techniques also suffer from some limitations, such as how to ease the demand of heavy mathematics background knowledge to perform theorem proving, and how to model the target without having the state explosion problem of model checking occurred.

To tackle with the aforementioned limitations, we propose an approach to requirements validation using an extended version of the formal language Timed Abstract State Machine (TASM), which contains new constructs *TASM Event* and *TASM Observer*. Additionally, TASM has shown its success in the area of systems verification in [7] [8], with some distinctive features: 1)

TASM supports the formal specification of both functional behaviors and non-functional properties of safety-critical systems w.r.t. timing and resource consumption and, 2) It is a literate language being understandable and usable without requiring extensive mathematical training, which avoids obscure mathematics formulae and, 3) TASM provides a toolset [9] to execute the pertaining TASM models for the purposes of analysis. The *Observer* technique [10] has an origin in the model-based testing domain where it has been used to specify and observe coverage criteria as well as verify such observable properties, but without changing the system's behaviors. The applications and advantages of using the *Observer* technique inspire us to exploit it to perform requirements validation, which makes a detour on the state explosion issue of model checking by not adding new states in the analysis. To be specific, our approach consists of three main steps:

- ***Requirements modeling*** models requirements by using various constructs in TASM.
- ***Features modeling*** translates features into our newly defined TASM observers that are used for the later analysis.
- ***Requirements validation*** contains four kinds of validation checking on focus, i.e., *Logical Consistency Checking*, *Auxiliary Machine Checking*, *Coverage Checking*, and *Model Checking*, as in the consistency and completeness checking of requirements.

The main contributions of this work are three-fold: 1) We extend the TASM language with two newly defined constructs in terms of *Event* and *Observer* and, 2) We propose a novel approach to requirements validation by using the extended TASM language and, 3) We demonstrate the applicability of our approach through a case study. The remainder of this paper is organized as follows: An introduction to the TASM language and its extension is presented in Section 6.2. Section 6.3 introduces the Brake-by-Wire (BbW) system and its requirements. Our approach to requirements validation is described and demonstrated by using the BbW system in Section 6.4. Section 6.5 discusses the related work, and finally concluding remarks and future work are drawn in Section 6.6.

6.2 TASM Language and Its Extension

Figure 6.1 shows the meta-model of the extended TASM language in UML class diagram. The constructs included in the dashed rectangle are the new

TASM constructs defined in this work. Section 6.2.1 gives an overview of the TASM language and Section 6.2.2 presents the extension of TASM.

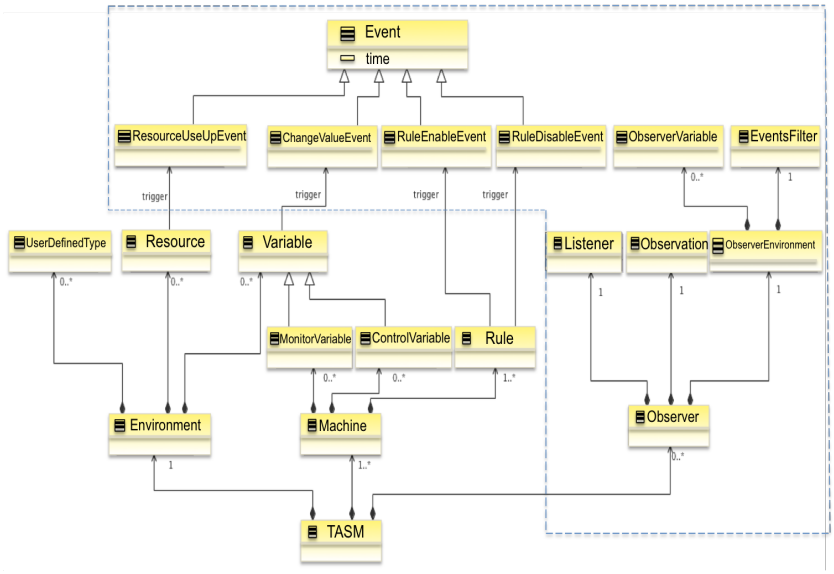

Figure 6.1: The Meta-model of the extended TASM language.

6.2.1 Overview of TASM

TASM [9] is a formal language for the specification of safety-critical systems, which extends the Abstract State Machine (ASM) [11] with the capability of modeling timing properties and resource consumption of applications in the target system. TASM inherits the easy-to-use feature from ASM, which is a literate specification language understandable and usable without extensive mathematical training [12]. A TASM model consists of two parts – an environment and a set of main machines. The environment defines the set and the type of variables, and the set of named resources which machines can consume. The main machine is made up of a set of monitored variables which can affect the machine execution, a set of controlled variables which can be modified by machines, and a set of machine rules. The set of rules specify the machine execution logic in the form of "if *condition* then *action*", where *condition* is an

expression depending on the monitored variables, and *action* is a set of updates of the controlled variables. We can also use the rule "else then *action*" which is enabled merely when no other rules are enabled. A rule can specify the annotation of the time duration and resource consumption of its execution. The duration of a rule execution can be the keyword *next* that essentially states the fact that time should elapse until one of the other rules is enabled.

TASM describes the basic execution semantics as the computing steps with time and resource annotations: In one step, it reads the monitored variables, selects a rule of which *condition* is satisfied, consumes the specified resources, and after waiting for the duration of the execution, it applies the update set instantaneously. If more than one rules are enabled at the same time, it nondeterministically selects one to execute. As a specification language, TASM supports the concepts of parallelism which stipulates TASM machines are executed in parallel, and hierarchical composition which is achieved by means of auxiliary machines which can be used in other machines. There are two kinds of auxiliary machines - *function* machines which can take environment variables as parameters and return execution result, and *sub* machines which can encapsulate machine rules for reuse purpose [9]. Communication between machines, including main machines and auxiliary machines, can be achieved by defining corresponding environment variables.

6.2.2 The Extension to TASM

Our extension to TASM consists of two main parts, i.e., *TASM Event* and *TASM Observer* (Event and Observer hereafter, respectively) as shown in Figure 6.1.

Definition 4. *TASM Event (EV). TASM Event E defines the possible types of an event instance, including ResourceUsedUpEvent, ChangeValueEvent, RuleEnableEvent, and RuleDisableEvent. An event instance e is triggered by the corresponding TASM construct, which is a tuple $< E, t >$, where E is the type of the event instance, and t is the time instant when the instance occurs.*

The events of $ChangeValueEvent$ type is triggered by a specific TASM environment variable whenever its value is updated,, which can be referenced in the form of *VariableName→EventType*. The $ResourceUsedUpEvent$ is triggered by the case whenever the resource of the application is consumed totally, which can be referenced in the form of *ResourceName→EventType*. The $RuleEnableEvent$ and $RuleDisableEvent$ are triggered whenever a specific TASM rule is enabled or disabled, respectively, which can be referenced in the form of *MachineName→RuleName→EventType*.

Definition 5. *TASM Observer. An observer is a tuple* $< ObserverEnvironment, Listener, Observation >$*, where:*

- $ObserverEnvironment$ *is a tuple* $< ObserverVariable, EventsFilter >$*, where* $ObserverVariable$ *is a set of variables that can be used by both* $Listener$ *and* $Observation$*, and* $EventsFilter$ *can be configured to filter out events irrelevant to the observer.*

- $Listener$ *specifies the observer execution logic in the form of "**listening** condition **then** action", where the condition is an expression describing the sequence of the occurrence of events and the action is a set of actions updating the value of observer variables when the condition evaluates to be true.*

- $Observation$ *is a predicate of the TASM model, which can evaluate to be either true or false, depending on the value of corresponding observer variables.*

In this work, we only introduce the informal execution semantics of *Observer*, as depicted in Figure 6.2, and the formal semantics is considered as part of our future work. Basically, in the runtime, the TASM model often produces massive events according to the modeled application. After the $EventsFilter$ removes the irrelevant events, the remaining events will be logged in the local database, namely $EventsLog$. Next, the *Listener* defined in *Observer* will evaluate its *condition* based off of the sequence of logged events. Since *regular expression* is usually used as a sequential search pattern, the specification of the event sequence follows the syntax and semantics of regular expression. If the *condition* is satisfied, then the *action* will start to update the observer variables. Once all of the updates are executed, the *Observation* will be concluded based on the updated observer variables. A running TASM model (representing the target system) can be observed by several observers at the same time.

For a better understanding, we give an example of *Observer* as shown in Figure 6.3, where *eventA* and *eventB* are $RuleEnableEvent$ type, and *eventC* and *eventD* are $RuleDisableEvent$ type. The observer variables include a Boolean variable ov (initiated as false) and a Time variable $time$ (initiated as zero). $ChangeValueEvent$ and $ResourceUsedUpEvent$ are regarded as irrelevant events and removed by the $EventsFilter$, the $RuleEnableEvent$ and $RuleDisableEvent$ events are logged in the *Eventslog* database. As shown in line 9 in Figure 6.3, the expression of the $Listener$ condition in

regular expression, represents the event sequence that begins with $eventA$, followed by arbitrary events (represented by ".*") in the middle, and ends with two events in terms of either $eventB\ and\ eventD$, or $eventC\ and\ eventD$. If the condition evaluates to be satisfied, the observer variable ov will be assigned as true, and $time$ as the interval between $eventA$ and $eventD$. In this example, if the events sequence in the condition is detected and the interval $time$ is larger than 100, the *Observation* will be concluded as a true predicate.

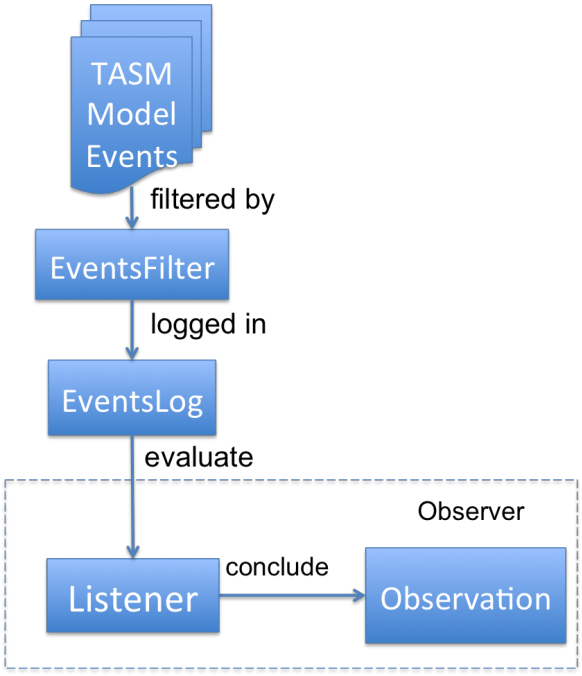

Figure 6.2: The workflow of the Observer execution.

6.3 Case Study

Our case study is a Brake-by-Wire (BbW) system which is a demonstrator at a major automotive company [13]. The BbW system aims to replace the mechanical linkage between the brake pedal and the brake actuators. Further, the

```
1 ObserverVariables:{
2       Boolean ov := false;
3       Time time := 0;
4 }
5 EventsFilter:{
6       filter out: ChangeValueEvent, ResourceUsedUpEvent;
7 }
8 Listener:{
9       listening eventA.*(eventB | eventC)eventD then
10     ov := true;
11     time := eventD.t - eventA.t;
12 }
13 Observation:{
14   ov == true and time > 100;
15 }
```

Figure 6.3: An example of the TASM Observer.

BbW system consists of micro-controller units, sensors, actuators and communication bus, which interprets driver's operation and operating conditions, through sensors, to decide on the desired brake torque of the brake actuators for appropriate brake force on each wheel.

The features that the BbW system should possess are described as follows:

- **Req H1**: The system shall provide a base brake functionality where the driver indicates that she/he wants to reduce speed so that the braking system starts decelerating the vehicle.
- **Req H2**: When the brake pedal is not pressed, the brake shall not be active.
- **Req H3**: The time from the driver's brake request till the actual start of the deceleration should be no more than 300 ms.

The list of requirements for the BbW system in our work is as follows:

- **Req L1**: The brake torque calculator shall compute the driver requested torque and send the value to the vehicle brake controller, when a brake pedal displacement is detected.
- **Req L2**: The vehicle brake controller shall decide the required torque on each wheel and each of the required wheel torque values is sent together with the sensed vehicle velocity to the Anti-lock Braking System (ABS) function on respective wheel.

- **Req L3**: The ABS function shall decide appropriate braking force on each wheel, based on the received torque request, current vehicle velocity and wheel angular velocity.

6.4 The TASM-based Approach to Requirements Validation

In this section, we will introduce our approach that addresses the issue of formalizing and validating requirements specifications written in natural language. Further, our approach is based on the use of the extended TASM language to formalize both requirements and features. We will go into details about each step by introducing the adhering sub-steps and show an illustration by using the BbW system. Specifically, Section 6.4.1 and Section 6.4.2 discuss modeling of the requirements and features respectively, and Section 6.4.3 presents the analysis and results of requirements validation of the BbW system.

6.4.1 Requirements Modeling

The first step of our approach is to analyze the low-level requirements (i.e., , requirements) in natural language and formalize them by using the corresponding TASM models. This step contains five sub-steps, as shown in Figure 6.4:

Figure 6.4: The sub-steps of requirements modeling.

- **Step 1: Requirements Preprocessing** distinguishes functional requirements from non-functional requirements.
- **Step 2: Components Identification** extracts the possible software components of the system referred in the functional requirements and maps them onto TASM main machines.
- **Step 3: Connections Identification** identifies the connections between different software components, according to a certain type of interactions.

Main Machine	Quantity	Category	Description
DRIVER	1	External Entity	model the driver's behavior
VEHICLE	1	External Entity	model the behavior of the vehicle
TORQUE_CALC	1	Micro-controller	calculate the driver's requested torque
BRAKE_CTRL	1	Micro-controller	calculate the requested torque per wheel
ABS_CTRL	4	Micro-controller	calculate the brake force on each wheel
BRAKE_ACTU	4	Actuator	perform the brake force on each wheel
WHLSPD_SENSOR	4	Sensor	sense the rotating speed of each wheel
VCLSPD_SENSOR	1	Sensor	sense the moving speed of the vehicle
PEDAL_SENSOR	1	Sensor	sense the position of the brake pedal
COMMU_BUS	1	Bus	the communication bus

Table 6.1: The TASM main machines model the entire Brake-by-Wired system.

- **Step 4: Behavior Specification** specifies the behaviors of components, which implement different system functionalities.

- **Step 5: Property Annotation** adds timing and resource consumption annotations to the relevant TASM model.

Requirements Preprocessing.

At this step, we need to distinguish functional requirements from non-functional requirements. The functional requirements will be formalized into executable TASM models, and non-functional requirement in terms of timing and resource consumption requirements can provide useful information for property annotation. In the BbW system, all the requirements, i.e., *ReqL1*, *ReqL2* and *ReqL3*, are functional requirements.

Components Identification.

The identification of the system components and the mapping of each component onto a TASM main machine is of importance in the process. In order to do so, we recommend the following two tasks:

- *Identification of the external (or environmental in other words) components that interact with the system.*

- *Identification of the internal components that compose of the system.*

At this step, a list of main machines will be defined for the BbW system, as shown in Table 6.1.

Connections Identification.

In the TASM model, asynchronous communication between different main machines can be implemented by using a set of variables, which ignores the transmission delay between machines. On the contrary, the common form of interprocess communication (IPC) is message-passing, which considers the transmission delay and bandwidth consumption as unavoidable. To this end, we define a main machine with the annotation of time and bandwidth as a means of modeling the communication bus. In our case study, the sensors in the BbW system communicate with the corresponding controllers through ports using signals, where transmission delay can be ignored. Further, a specific TASM main machine i.e., COMMU_BUS (in Table 6.1) models the communication bus, which is responsible for the communication between the brake controller and the ABS controllers.

Behavior Specification.

There is no silver-bullet to model the behaviors of various components in TASM. Based on our experiences, we recommend the following steps:

- *Identification of possible states of the target system*: A user-defined type is used to represent the possible states, and a state variable is defined to denote the current state of the system.
- *Identification of the transition conditions of states*: The conditions of a certain machine rule are given, according to the corresponding value of the state variable and the transition conditions.
- *Identification of the actions when the system enters a specific state*: The actions of machine rules are specified, based on the behaviors of a component and the next possible state.

In the BbW system, all of the identified components (i.e., TASM main machines) are divided into five categories according to different functionalities: external entity, micro-controller, actuator, sensor, and bus. For reasons of space, we do not list all the rules used by the identified TASM main machines. Instead, we list the rules of four typical templates in our case study, i.e., micro-controller, actuator, sensor, and bus. In order to have a better understanding on the proposed sub-steps, we discuss the specification of a micro-controller component in detail.

A micro-controller component is activated by an event, and it reads a set of variables and performs a sequence of computation after being activated. When it finishes execution, the result will be used by other components. Therefore, the micro-controller component typically has three possible states – WAIT (initial state), COMPUTE, and SEND: The WAIT sate denotes that the micro-controller is waiting for activation and, the COMPUTE state represents that the micro-controller is performing computation. The SEND state introduces that the micro-controller is sending the results to other components. Figure 6.5 shows the rules of the TASM main machine, which models the micro-controller. PERFORM_COMPUTATION() and SEND_RESULT() are sub machines.

Figure 6.6 shows the machine rules that model an actuator, and PERFORM_A- CTUATION() is a sub machine. Figure 6.7 shows the rules of the TASM main machine, which models a sensor. Measure_Quantity() is a function machine. Figure 6.8 shows the machine rules, which models the communication bus. Get_Message() is a function machine and TRANSMITTING_MESSAGE() is a sub machine.

```
R1:Activation{
  if ctrl_state=wait and new_event=True then
    ctrl_state := compute;
    new_event  := False;
}
R2:Computation{
  t:=computation_time;
  if ctrl_state = compute then
    PERFORM_COMPUTATION();
    ctrl_state := send;
}
R3:Send{
  if ctrl_state = send then
    SEND_RESULT();
    ctrl_state := wait;
}
R4:Idle{
  t := next;
  else  then
    skip;
}
```

Figure 6.5: The TASM main machine models the micro-controller component.

```
R1:Trigger{
 if actu_state=wait and new_event=True then
   new_event  := False;
   actu_state := actuate;
}
R2:Actuation{
 t:=actuation_time;
 if actu_state=actuate then
   PERFORM_ACTUATION();
   actu_state := wait;
}
R3:Idle{
 t:= next;
 else then
   skip;
}
```

Figure 6.6: The TASM main machine models the actuator component.

```
R1:Sample{
  if sensor_state = sample then
    sensor_value := Measure_Quantity();
    sensor_state := send;
}
R2:Send{
  if sensor_state = send and sensor_value >= threshold then
    observer_value  := sensor_value;
    new_sample_value:= True;
    sensor_state    := wait;
}
R3:Wait{
  t := period;
  if sensor_state = wait then
    sensor_state := sample;
}
```

Figure 6.7: The TASM main machine models the sensor component.

Non-functional Property Annotation.

The accurate estimation of the pertaining non-functional properties of the target system is playing a paramount role in performing non-functional requirements validation. The Property Annotation step can be carried out in the following ways:

- The estimates can be determined based upon the non-functional requirements specified in the low-level requirements.

```
R1:Transmit{
  if bus_state=idle and new_message=True then
    bus_message := Get_Message();
    bus_state := engaged;
}
R2:Send{
  t:=bus_delay;
  band:= bandwidth;
  if bus_state = engaged then
    TRANSMITTING_MESSAGE();
    bus_state := idle;
}
R3:Wait{
  t := next;
  else then
    skip;
}
```

Figure 6.8: The TASM main machine models the communication bus component.

- The estimates can be obtained by using existing well-known analysis methods, e.g., Worst-Case Execution Time (WCET) Analysis [14] for time duration of rules.
- The estimates can be determined based upon the information in the related hardware specifications, e.g., the time duration and power consumption of a communication bus transferring one message.
- However, in some cases, the estimates can also be given by the experiences of domain experts, if the accurate estimation is not possible.

We annotate the aforementioned TASM models with time duration and resource consumption, and the annotation terms *computation_time*, *actuation_time*, *period*, *bus_delay* and *bandwidth* are either a specific value or a range of values, which are given by our domain knowledge for simplicity.

6.4.2 Features Modeling

Our approach proceeds with the formalization of high-level requirements, i.e., , features. At this step, each feature will be translated into corresponding TASM observer(s). The formalization consists of the following sub-steps:

- **Step 1: Listener Specification** specifies the possible events sequence which represents the observable functional behaviors or non-functional

properties required by the feature, and the corresponding actions taken on observer variables when the sequence is caught by the Listener.

- **Step 2: Observation Specification** formalizes a predicate depending on the observer variables. If the predicate of the Observation holds, i.e., evaluates to be true, it implies that the satisfaction of the feature can be observed in the system.
- **Step 3: Events Filtering** identifies the interesting events and filters out the irrelevant events by specifying $EventsFilter$.
- **Step 4: Traceability Creation** links the specified Observer to the textual requirements. The link is used for requirements traceability from the formalization to natural language requirements in order to perform coverage checking.

In the BbW system, there are three features i.e., *ReqH1*, *ReqH2* and *ReqH3*. The specification of *Observer* is illustrated by applying the proposed steps to *ReqH1*, as shown in Figure 6.9. To be specific, *ReqH1* states "The system shall provide a base brake functionality where the driver indicates that she/he wants to reduce speed so that the braking system starts decelerating the vehicle", and the interesting events sequence consists of three parts. The first part "PEDAL_SENSOR→Send→RuleEnableEvent" denotes the event that is triggered when the Send rule of the PEDAL_SENSOR main machine is enabled, which models the behavior that the brake pedal is pressed by the driver. The second part ".*" has the same semantic with the counterpart defined in *regular expression*, which means an arbitrary number of events regardless of their type. The last part "BRAKE_ACTU→Actuation→RuleEnableEvent" represents the event that is triggered after the Actuation rule of the BRAKE_ACTU main machine is executed, i.e., disabled, which models the behavior that the brake actuator acts on the wheels i.e., decreases the speed of the vehicle. When the events sequence is detected, the Observation "ov == true" evaluates to be true, which indicates that the satisfaction of *ReqH1* can be observed in the TASM model.

6.4.3 Requirements Validation

Validation of the formalized requirements aims at increasing the confidence in the validity of requirements. In this work, we assume that there is a semantic equivalence relation between the requirements and TASM models, and

```
ObserverVariables:{
        Boolean ov := false;
}
EventsFilter:{
        filter out: ChangeValueEvent, ResourceUsedUpEvent, RuleDisableEvent;
}
Listener:{
        listening PEDAL_SENSOR->Send->RuleEnableEvent
        .*
        BRAKE_ACTU->Actuation->RuleEnableEvent then
                ov := true;
}
Observation:{
        ov == true;
}
```

Figure 6.9: The Observer of Req H1.

between features and observers. This is built upon the fact that the TASM models and observers are derived from the documented requirements and features, by following the proposed modeling steps based on our thorough understanding of the BbW system. The validation goal is achieved by following several analysis steps, based on the use of the derived TASM models and observers which may help to pinpoint flaws that are not trivial to detect. Such validation steps in our approach are:

- **Logical Consistency Checking**. The term of logical consistency can be intuitively explained as ”free of contradictions in the specifications”. In our work, the logical consistency checking can be performed on the executable TASM models, i.e., requirements, by our developed tool TASM TOOLSET. Two kinds of inconsistency flaws can be discovered. One kind of flaw is that two machine rules are enabled at the same time, which is usually caused by the fact that there exist unpredictable behaviors in the requirements. The other is that different values are assigned to the same variable at the same time, which is usually caused by the fact that there exist hidden undesired behaviors in the requirements.

- **Auxiliary Machine Checking**. Auxiliary machines include function machine and sub machine. When the TASM TOOLSET starts to execute the TASM model, if there exists any undefined auxiliary machine, the tool will detect this situation, stop proceeding, and generate an error message. The existence of undefined auxiliary TASM machines, in terms of functions and sub machines, violates the completeness of

TASM model specifying requirements. The undefined auxiliary TASM machines are usually caused by the lack of detailed descriptions of the proposed system's behaviors.

- **Coverage Checking**. Coverage checking corresponds to checking whether the desired behaviors specified in features can be observed in the TASM model, which is an important activity of requirements completeness checking. To perform the coverage checking, all of the features are translated into observers which observe the execution of TASM models at runtime. If the *Observation* holds, the corresponding feature can be regarded as covered by the requirements.

- **Model Checking**. The TASM machines can be easily translated into timed automata through the transformation rules defined in [9]. The transformation enables the use of the UPPAAL model checker to verify the various properties of the TASM model. This check aims at verifying whether the TASM model is free of deadlock and whether an expected property specified in a feature is satisfied by the TASM model. It is necessary to stress that the essential difference between *Model Checking* and *Coverage Checking* is whether a property is exhaustively checked against a model or not. Although a sound property checking is desired, in some cases *Model Checking* will encounter state explosion problem, which limits its usefulness in practice.

We follow the validation steps to check the validity of the requirements of the BbW system. First, we use the TASM TOOLSET to perform *Logical Consistency Checking* on the formalized TASM model. As in the fact that there are no inconsistency warnings reported by the tool, we therefore proceed the validation steps with *Auxiliary Machine Checking*. As shown in Figure 6.5, 6.6, 6.7 and 6.8, there exist some undefined auxiliary machines in the TASM models of those typical components, which also have been detected by our TASM TOOLSET. For instance, in the *ABS_CTRL* main machine (a microcontroller component), the *PERFORM_COMPUTATION* sub machine is not defined, which implies that the requirements need to specify in more details about how *"The ABS function shall decide appropriate braking force on each wheel"*. Next for *Coverage Checking*, since the observations are determined to be held according to the results of the TASM observers in the runtime, the satisfaction of requirements towards features is therefore reached. On the note about *Model Checking*, we first translate the TASM model into timed automata, and then check the deadlock property as well as the *ReqH3* requirement via the

UPPAAL model checker. The corresponding results are: 1) *Deadlock free* is *satisfied* and, 2) the *ReqH3* is *satisfied*. Although the case study is a demonstrator, it is an illustrative example to show how to follow our proposed approach to perform requirements validation at various levels.

6.5 Related Work

In addition to the aforementioned related work, there are some other interesting pieces of work deserved to be mentioned as follows. Event-B [15] is a formal state-based modeling language that represents a system as a combination of states and state transitions. Iliasov [6] showed how to use Event-B for systems development, where the system constraints are formalized as a set of visualized proof obligations which can be synthesized as use cases. Such proof obligations are then reasoned about their satisfaction in the corresponding Event-B model. Mashkoor et al. [16] proposed a set of transformation heuristics to validate the Event-B specification by using animation.

Cardei et al. [17] presented a methodology that first converts SysML requirements models into a requirements model in OWL, and then performs the rule-based reasoning to detect omissions and inconsistency. Becker et al. [18] provided a formalization for self-adaptive systems and the corresponding requirements, which enables a semi-automatic analysis of performance requirements for self-adaptive systems. Cimatti et al. [5] introduced a series of techniques that have been developed for the formalization and validation of requirements for safety-critical systems. Specifically, the methodology consists of three main steps in terms of informal analysis, formalization, and formal validation. Scandurra et al. [19] proposed a framework to automatically transform use cases into ASM models, which are used to validate the requirements through scenario-based simulation. MARTE [20] is a UML profile for modeling and analysis of RTES, covering both functional and non-functional properties of the system. Nevertheless, to our best knowledge, there has not been any work about using MARTE for the purposes of requirements validation.

6.6 Conclusions and Future Work

In this paper, we have proposed a novel TASM-based approach to requirements validation. The approach 1) uses the extended TASM language to model the documented requirements and, 2) performs the requirements validation by

using two tools in terms of the TASM TOOLSET and the model checker UPPAAL. Our case study using a Brake-by-Wire (BbW) system developed by a major automotive company, has shown that our approach can achieve the goal of requirements validation via *Logical Consistency Checking*, *Auxiliary Machine Checking*, *Coverage Checking*, and *Model Checking*. Even if limited in complexity, the BbW system consists of a number of parts presenting the real world safety-critical systems, such as micro-controllers, sensors, actuators, and communication buses.

In this work, the validity of our TASM model towards requirements and features is built upon our thorough understanding of the BbW system, and hence TASM models are semantic preserving. Moreover, we have observed model validation issue as a common problem with model-based approaches. This is getting more complicated when the system's non-functional properties are considered. To address the situation, as future work, we will combine our proposed modeling approach with a set of assistant techniques, such as rule/pattern-based algorithm to semi- or fully-automatically transform natural languages into TASM models. The future work also includes a wider industrial validation of our approach, and the improvement of our current TASM TOOLSET. Such improvement will not only facilitate our evaluation but also power up our analysis with statistical methods [14] and probabilistic modeling patterns.

Bibliography

[1] A. Terry Bahill and Steven J. Henderson. Requirements development, verification and validation exhibited in famous failures. *Syst. Eng*, 2005.

[2] AdrianF. Ellis. Achieving safety in complex control systems. In *Proceedings of SCSC'95*, pages 1–14. Springer London, 1995.

[3] Nancy G. Leveson. *Safeware: System Safety and Computers*. ACM, NY, USA, 1995.

[4] Didar Zowghi and Vincenzo Gervasi. The three cs of requirements: Consistency, completeness, and correctness. In *Proceedings of REFSQ'02*, 2002.

[5] Alessandro Cimatti, Marco Roveri, Angelo Susi, and Stefano Tonetta. From informal requirements to property-driven formal validation. In *Proceedings of FMICS'09*, pages 166–181. Springer-Verlag, Berlin, Heidelberg, 2009.

[6] Alexei Iliasov. Augmenting formal development with use case reasoning. In *Proceedings of Ada-Europe'12*, pages 133–146. Springer Berlin Heidelberg, 2012.

[7] Zhibin Yang, Kai Hu, Dianfu Ma, and Lei Pi. Towards a formal semantics for the AADL behavior annex. In *Proceedings of DATE'09*, pages 1166–1171, 2009.

[8] Jiale Zhou, Andreas Johnsen, and Kristina Lundqvist. Formal execution semantics for asynchronous constructs of aadl. In *Proceedings of ACES-MB'12*, pages 43–48, 2012.

[9] M. Ouimet. *A formal framework for specification-based embedded real-time system engineering*. PhD thesis, Department of Aeronautics and Astronautics, MIT, 2008.

[10] Johan Blom, Anders Hessel, Bengt Jonsson, and Paul Pettersson. Specifying and generating test cases using observer automata. In *Proceedings of FATES'04*, pages 125–139. Springer-Verlag, 2005.

[11] Egon Börger and Robert F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.

[12] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, December 1996.

[13] MAENAD. `http://www.maenad.eu`, 2013.

[14] Yue Lu. *Pragmatic Approaches for Timing Analysis of Real-Time Embedded Systems*. PhD thesis, Mälardalen University, June 2012.

[15] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[16] Atif Mashkoor, Jean-Pierre Jacquot, and Jeanine Souquières. Transformation Heuristics for Formal Requirements Validation by Animation. In *Proceedings of SafeCert'09*, York, United Kingdom, 2009.

[17] I. Cardei, M. Fonoage, and R. Shankar. Model based requirements specification and validation for component architectures. In *Systems Conference, 2008 2nd Annual IEEE*, pages 1–8, 2008.

[18] Matthias Becker, Markus Luckey, and Steffen Becker. Performance analysis of self-adaptive systems for requirements validation at design-time. In *Proceedings of QoSA'13*, pages 43–52, New York, USA, 2013. ACM.

[19] Patrizia Scandurra, Andrea Arnoldi, Tao Yue, and Marco Dolci. Functional requirements validation by transforming use case models into abstract state machines. In *Proceedings of SAC'12*, pages 1063–1068, NY, USA, 2012. ACM.

[20] OMG. `http://www.omgmarte.org/`, 2013.

Chapter 7

Paper C: Towards Feature-Oriented Requirements Validation for Automotive Systems

Jiale Zhou, Yue Lu, Kristina Lundqvist, Henrik Lönn, Daniel Karlsson, Bo Liwång.
Proceedings of the 22nd IEEE International Requirements Engineering Conference (RE'14), Karlskrona, Sweden, August 2014.

Abstract

In the modern automotive industry, feature models have been widely used as a domain-specific requirements model, which can capture commonality and variability of a software product line

through a set of features. Product variants can thus be configured by selecting different sets of features from the feature model. For feature-oriented requirements validation, the variability of feature sets often makes the hidden flaws such as behavioral inconsistencies of features, hardly to avoid. In this paper, we present an approach to feature-oriented requirements validation for automotive systems w.r.t. both functional behaviors and non-functional properties. Our approach first starts with the behavioral specification of features and the associated requirements by following a restricted use case modeling approach, and then formalizes such specifications by using a formal yet literate language for analysis. We demonstrate the applicability of our approach through an industrial application of a Vehicle Locking-Unlocking system.

7.1 Introduction

With the growing maturity and standardization of the automotive domain, requirements specifications for automotive systems tend to center around the concept of feature models. Feature models [1] are proposed to capture the commonality and variability within a software product line by using features, between which there are relations and constraints. Further, a feature [2], [3] is a logical unit of functionality comprehensible to end-users, which consists of the requirements (i.e., the feature requirement hereafter) associated with the feature and the corresponding behavioral specification (i.e., the feature behaviors hereafter). A product can be configured by selecting a set of features (i.e., the feature set hereafter) from a feature model. The validity of a feature set refers to two situations: 1) one is from the structural perspective, i.e., the selected features should conform to the constraints defined by the feature model and, 2) the other is from the functional perspective, i.e., no undesirable behaviors exist between two or more (as integrated) feature behavioral specifications. In order to increase the confidence of the validity of the feature set, several feature-oriented requirements validation techniques [3], [4], [5], [6], [7], [8] have been developed. However, it is well recognized that with the increasing size of feature models, the inherent variability of the feature sets leads to an inevitable issue that the hidden flaws of features are difficult to avoid [9]. Especially, the feature interaction problem (referring to the situation that two or more features exhibit unexpected behaviors) cannot be detected when the features are used in isolation.

As the unexpected behaviors can result in uncertainties and even hazards of automotive systems, adequate efforts on detecting the unexpected feature interactions thereby must be applied in the early stages of the pertaining development process. In the literature, there are many examples [7], [10], [11] where the process starts with translating the natural language specification (NLS) of a feature into a formal language specification (FLS) of the feature behaviors, and then the requirements validation is performed based on the generated formalisms. To our best knowledge, the main drawbacks of using NLS lie in: 1) ambiguities in the NLS cause imprecise definitions and even wrong understanding of the feature behaviors and, 2) the direct translation from the NLS to a FLS tends to be very costly and, 3) the NLS hinders to a large extent the possibility of performing automatic feature-oriented requirements validation.

To challenge the feature interaction problem and make up for the deficiency in the current practice, in this paper we propose a model-based approach to feature-oriented requirements validation. To be specific, our approach firstly

specifies features by using an informal yet restricted natural language (from scratch) without losing ease of use, and then formalizes a set of executable models based upon the aforementioned intermediate specifications to perform the requirements validation. The approach is comprised of four steps as follows:

- **Feature Specification** specifies the behaviors and requirements of features by following the restricted use case modeling (RUCM) approach [12], which adopts a generic use case template and several restriction rules to reduce ambiguity and facilitate further analysis.

- **Feature Behaviors Formalization** formalizes the feature behaviors in terms of a formal yet literate specification using the extended Timed Abstract State Machine (eTASM) language [13], which generates executable models for analysis.

- **Feature Requirements Formalization** models the feature requirements by using the Observers technique [13] in eTASM for validation purpose.

- **Feature Validation** performs three kinds of model-based validation checking to detect the hidden flaws in the selected features, including *Logical Consistency Checking*, *Coverage Checking*, and *Model Checking*.

We also demonstrate the applicability of our approach through an illustration application, and the remainder of this paper is organized as follows: An introduction to the background knowledge is presented in Section 7.2. Section 7.3 introduces the illustration application i.e., the Vehicle Locking-Unlocking (VLU) system. Our approach to feature-oriented requirements validation is described and illustrated by using the VLU system in Section 7.4. Section 7.5 discusses the related work, and finally concluding remarks and future work are drawn in Section 7.6.

7.2 Background

In this section, we briefly introduce the RUCM approach [12] and the formal specification language eTASM [13] used in our approach for a better understanding.

7.2.1 Restricted Use Case Modeling

The restricted use case modeling (RUCM) [12] is a use case modeling approach that extends the UML [14] use case diagram by proposing a use case template and 26 restriction rules for reducing ambiguity and easing automated analysis. In our work, we specify features by populating the proposed use case template and following the restriction rules. In order to meet our needs, we make two slight modifications to the template. First, for the purpose of traceability, the use case name is required to follow the form of *FeatureName_UseCaseName* or merely *FeatureName*. Second, the *Basic Description* entry is replaced by *Feature Requirement* which specifies the requirement that the feature is associated with. Figure 7.1 shows the modified template and the brief explanation for each entry.

Use Case Name	In the form of *FeatureName_UseCaseName* or merely *FeatureName*.	
Feature Requirement	Specifies the feature requirement.	
Precondition	What should be true before the use case is executed.	
Primary Actor	The actor who initiates the use case.	
Secondary Actors	Other actors the system relies on to accomplish the functionality of the use case	
Dependency	Include and extend relationships to other use cases.	
Generalization	Generalization relationships to other use cases.	
Basic flow steps	Specifies the main successful path in terms of a sequence of steps and a postcondition	
	Steps (numbered)	Flow of events
	Postcondition	What should be true after the basic flow executes.
Specific Alt. Flow	Applies to one specific step of the reference flow	
	RFS	A reference flow step number where flow branches from.
	Steps (numbered)	Flow of events
	Postcondition	What should be true after the basic flow executes.
Bounded Alt.Flow	Applies to more than one step of the reference flow, but not all of them.	
	RFS	A list of reference flow steps where flow branches from.
	Steps (numbered)	Flow of events
	Postcondition	What should be true after the basic flow executes.
Global Alt.Flow	Applies to all the steps of the reference flow.	
	Steps (numbered)	Flow of events
	Postcondition	What should be true after the basic flow executes.

Figure 7.1: The modified use case template of RUCM.

The feature behaviors are specified via use case flows, which are composed of one basic flow and one or more alternative flows. The basic flow specifies the main execution path in terms of a sequence of steps and a postcondition. Al-

ternative flows specify execution branches when deviations occur somewhere in the reference flow that can be the basic flow or an alternative flow. There are three types of alternative flows: a specific alternative flow refers to a specific step in the reference flow; a bounded alternative flow refers to more than one step in the reference flow; a global alternative flow refers to all steps in the reference flow. RUCM defines 16 restriction rules to constrain the use of natural language, as shown in Figure 7.2. A set of keywords are also defined in the other 10 rules to specify control structures. For example, the keyword VALIDATES THAT (as shown in Figure 7.4) is used for condition checking. In particular, if the condition evaluates to be true, the current flow continues, otherwise an alternative flow will be executed. The detailed description of all the restriction rules and keywords are provided in [12].

#	Description	Explanation
R1	The subject of a sentence in basic and alternative flows should be the system or an actor	Enforce describing flows of events correctly. These rules conform to our use case template (the five interactions).
R2	Describe the flow of events sequentially	
R3	Actor-to-actor interactions are not allowed	
R4	Describe one action per sentence. (Avoid compound predicates.)	Otherwise it is hard to decide the sequence of multiple actions in a sentence.
R5	Use present tense only	Enforce describing what the system does, rather than what it will do or what it has done.
R6	Use active voice rather than passive voice	Enforce explicitly showing the subject and/or object(s) of a sentence.
R7	Clearly describe the interaction between the system and actors without omitting its sender and receiver	
R8	Use declarative sentences only. “Is the system idle?” is a non-declarative sentence.	Commonly required for writing UCSs
R9	Use words in a consistent way.	Keep one term to describe one thing
R10	Don’t use modal verbs (e.g., might)	Modal verbs and adverbs usually indicate uncertainty; therefore metrics should be used if possible
R11	Avoid adverbs (e.g., very).	
R12	Use simple sentences only. A simple sentence must contain only one subject and one predicate	Reduce ambiguity and facilitate automated natural language parsing.
R13	Don’t use negative adverb and adjective (e.g., hardly, never), but it is allowed to use not or no	
R14	Don’t use pronouns (e.g. he, this).	
R15	Don’t use participle phrases as adverbial modifier.	
R16	Use “the system” to refer to the system under design consistently.	Keep one term to describe the system; therefore reduce ambiguity

Figure 7.2: The rules of RUCM to constrain the use of natural language.

7.2.2 The Extended TASM Language

eTASM [13] is a formal language for the specification of safety-critical systems, which extends the Timed Abstract State Machine (TASM) language [15] with the Observer and Event constructs. eTASM inherits the easy-to-use characteristic from TASM, which is a literate specification language understandable and usable without extensive mathematical training. An eTASM model consists of three parts – an environment, a set of machines, and a set of observers. The environment defines the set and the type of machine variables which machines can monitor or control, and the set of named resources which machines can consume. An machine consists of a set of monitored variables (which can affect the machine execution), a set of controlled variables (which machines can modify), and a set of machine rules. The set of rules specify the machine execution logic in the form of "if *condition* then *action*", where *condition* is an expression depending on the monitored variables, and *action* is a set of updates of the controlled variables. We can also use the rule "else then *action*" which is enabled merely when no other rules are enabled. A rule can specify the annotation of the time duration and resource consumption of its execution. The duration of a rule execution can be the keyword *next* that essentially states the fact that time should elapse until one of the other rules is enabled. The observers will monitor the events triggered by the execution of machines, and each observer represents one correctness property of interest that should be satisfied by the proposed system. In the eTASM language, four types of events can be triggered: The *ChangeValueEvent* type is triggered by a specific eTASM environment variable whenever its value is updated, the *ResourceUsedUpEvent* is triggered by the case whenever the resource of the application is consumed totally, and the *RuleEnableEvent* and *RuleDisableEvent* are triggered whenever the corresponding eTASM rule is enabled or disabled, respectively. An observer is made up of an ObserverEnvironment, a Listener, and an Observation. The ObserverEnvironment defines a set of observer variables and an EventsFilter that filters out irrelevant event types to the observer. The Listener specifies the expected events sequence following the syntax and semantics of *regular expression*. The Observation indicates the monitoring result, i.e., whether the correctness property monitored by the observer is satisfied.

eTASM describes the basic execution semantics as the computing steps with time and resource annotations: In one step, it reads the monitored variables, selects a rule of which *condition* is satisfied, consumes the specified resources, and after waiting for the duration of the execution, it applies the update set instantaneously. During the execution, eTASM triggers events whenever

possible. The events sequence is monitored by observers. Once an expected sequence is observed, the corresponding monitoring result will be concluded. As a specification language, eTASM supports the concepts of parallelism (which stipulates that eTASM machines are executed in parallel) and hierarchical composition (which is achieved by means of auxiliary machines which can be used in other machines). There are two kinds of auxiliary machines - *function* machines that can take machine variables as parameters and return an execution result, and *sub* machines that can encapsulate machine rules for reuse purpose [15]. Communication and interaction between machines can be achieved by defining corresponding environment variables.

7.3 Illustration Application

In this section, we describe a simplified Vehicle Locking-Unlocking (VLU) system, as a running example to illustrate our approach in this work. The proposed VLU system aims to replace the mechanical key, as a control access to a vehicle, and it follows a common pattern in feature-oriented requirements specification: The basic functionality is encapsulated as an individual feature, and additional/optional enhancements are specified as features that provide the increments in functionality. Specifically, such features are Central Locking (CL), Auto-lockout (AUL) and Anti-lockout (ANL). Figure 7.3 shows the features of the VLU system in the form of technical feature model tree which is presented in EAST-ADL [16].

Central Locking (a basic feature) locks and unlocks all the doors of the vehicle upon receipt of a command from the user key fob.

Auto-lockout (an optional feature) locks all the doors of the vehicle when a timeout expires after the vehicle has stopped. It provides a theft protection in case that the driver forgets locking the doors manually.

Anti-lockout (an optional feature) enables unlocking of the doors while a key is in ignition after the vehicle has stopped, of which purpose is to prevent the driver from being locked out of the vehicle.

In simple applications such as the one above, it is possible to manually analyze the interactions between features for requirements validation. However, the real-world systems often have a large number of complex features, making the pertaining manual analysis extremely time-consuming and error-prone. The main motivation for our approach is to provide a semi-automatic technique for feature-oriented requirements validation for automotive systems, by performing undesirable feature interaction analysis. In the rest of the paper,

we will use the aforementioned simplified application to illustrate our approach for features modeling, features specification, and auto-detection of undesired interactions.

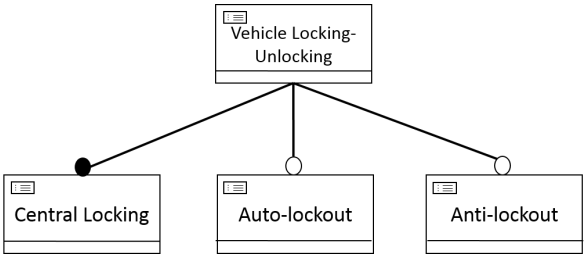

Figure 7.3: The technical feature model tree of the VLU system.

7.4 The Approach to Feature-Oriented Requirements Validation

In this section, we will introduce our approach that addresses the issue of formalizing and validating feature-oriented requirements specifications. In general, our approach is conducted in four steps as follows:

- **Step 1: Feature Specification** specifies the behaviors and requirements of features by using the RUCM use cases, which facilitates the further analysis.
- **Step 2: Feature Behaviors Formalization** formalizes the feature behaviors using the eTASM machines, which are executable analysis models.
- **Step 3: Feature Requirements Formalization** formalizes the feature requirements by using the eTASM Observer technique.
- **Step 4: Feature Validation** performs three types of checking by using model-based analysis techniques, to detect the hidden flaws in feature specifications.

We will go into details about each step by introducing the adhering substeps and show a running example to illustrate our approach. Specifically,

Table 7.1: The identified use cases and corresponding feature requirements.

Feature	Use case name	Feature requirements
Central Locking	CL_Lock	The system shall lock the doors
	CL_Unlock	The system shall unlock the doors
Auto-lockout	Auto-Lockout	The system shall automatically lock the doors after 20 seconds, when the vehicle has stopped
Anti-lockout	Anti-Lockout	The system shall anti-lock the doors if the key is in ignition and the vehicle has stopped

Section 7.4.1 introduces feature specification using the RUCM use cases. Section 7.4.2 and Section 7.4.3 discuss modeling of the behaviors and requirements of features respectively. Section 7.4.4 presents the analysis and results of feature validation of the VLU system.

7.4.1 Feature Specification

The Feature Specification step describes the features of a system in a restricted natural language, which can facilitate the further transformation from an informal specification to the formal one, for the purpose of validation. In this step, each feature will be specified by following the RUCM approach, and there are two sub-steps in our work as follows:

- **Step 1.1: Use Cases Identification**: Since a feature captures a set of cohesive functionalities in the form of requirements and corresponding behaviors, it is therefore necessary to split the functionalities and identify the possible use cases based on the expert's understanding of the feature.

- **Step 1.2: Use Cases Specification**: Use cases are specified by filling the RUCM template using a restricted natural language. In order to facilitate the further analysis, some predefined restriction rules must be followed.

In the VLU system, there are three selected features i.e., *CL*, *AUL* and *ANL* (as introduced in Section 7.3). The specification of features, as an example, is illustrated by applying the proposed steps to the *CL* feature. To be specific, since the *CL* feature describes two opposite functionalities, two use cases can be thereby identified in terms of *CL_Lock* and *CL_Unlock*, as shown in Table 7.1. Figure 7.4 and Figure 7.5 describe the filled use case templates of *CL_Lock* and *CL_Unlock*, respectively.

Use Case Name	**CL_Lock**			
Feature Requirement	The system shall lock the doors of the vehicle			
Precondition	None			
Primary Actor	Key fob	**Secondary Actors**	Doors, Lights	
Dependency	None	**Generalization**	None	
Basic flow steps	1) Key fob sends a ”lock” command to the system. 2) The system **VALIDATES THAT** the doors are close. 3) The system locks the door. 4) The system flashes lights to indicate the completion of locking. **Postcondition**: Doors are locked. Lights is off. The system is idle.			
Specific Alt. Flow (RFS Basic flow 2)	1) The system does nothing. **Postcondition**: Doors remain open. Lights are off.			
Bounded Alt.Flow	None	**Global Alt.Flow**	None	

Figure 7.4: The CL_Lock use case of the CL feature.

Use Case Name	**CL_UnLock**			
Feature Requirement	The system shall unlock the doors of the vehicle			
Precondition	None			
Primary Actor	Key fob	**Secondary Actors**	Doors, Lights	
Dependency	None	**Generalization**	None	
Basic flow steps	1) Key fob sends an ”unlock” command to the system. 2) The system unlocks the door. 3) The system flashes lights to indicate the completion of unlocking. **Postcondition**: Doors is close but unlocked. Lights is off. The system is idle			
Specific Alt. Flow	None			
Bounded Alt.Flow	None	**Global Alt.Flow**	None	

Figure 7.5: The CL_Unlock use case of the CL feature.

7.4.2 Feature Behaviors Formalization

This step is to analyze the specified RUCM templates and formalize the corresponding feature behaviors by using eTASM models which are executable simulation models for analysis. The Feature Behaviors Formalization step contains four sub-steps:

- **Step 2.1: System Constituents Identification** extracts the relevant system constituents referred in the RUCM use cases and specifies them in

eTASM machines.

- **Step 2.2: Constituents Interaction Identification** identifies the interactions between different system constituents referred in the RUCM use cases and specifies them in eTASM environment variables.
- **Step 2.3: Machine Rules Specification** analyzes the possible states of identified machines and specifies feature behaviors by using a set of eTASM machine rules.
- **Step 2.4: Property Annotation** adds non-functional property annotations to the relevant eTASM machines.

System Constituents Identification

The identification of the system constituents from the use cases is of importance in the process to formalize the behaviors of the proposed system and model the scenarios for model-based validation. In order to do so, we recommend the following two tasks:

- *External Constituents Identification*: Use case actors are considered as external constituents which interact with the proposed system. The external constituents will be modeled to simulate the execution scenarios.
- *Internal Constituents Identification*: Each use case is considered to be an internal constituent, making up the proposed system. The internal constituents will be modeled to simulate the proposed system.

In this step, a list of eTASM machines w.r.t. the identified constituents is defined for the VLU system, as shown in Table 7.2.

Constituents Interaction Identification

Two types of interaction between the sending constituent (i.e., sender) and receiving constituent (i.e., receiver) are considered in our approach:

- *Data Transmission Interaction (DTI)* represents that data (such as the state information and various sensor values) are transferred from the sender to the receiver, which are modeled as eTASM environment variables. The variables are named as *sender_datatype* which denotes the transferred data. Line 2 in Figure 7.7 shows an example.

Table 7.2: The eTASM machines identified for the VLU system.

Machine	Quantity	Category	Brief Description
KEY_FOB	1	External	model the key fob's behavior
LIGHT	1	External	model the light's behavior
DOORS	1	External	model the behavior of doors
IGNITION	1	External	model the behavior of ignition
VEHICLESPEEDSENSOR	1	External	model the behavior of speed sensor
CL_LOCK	1	Internal	lock doors
CL_UNLOCK	1	Internal	unlock doors
AUTO_LOCKOUT	1	Internal	lock doors when timeout expires
ANTI_LOCKOUT	1	Internal	anti-lock doors if key is in ignition

- *Data Modification Interaction (DMI)* represents that the data of the receiver is directly changed by the sender, which are modeled via directly modifying the value of the receiver's environment variable. The variables are named in the form of *receiver_datatype*, which denotes the modified data. One example can be found in Lines 9 and 10 in Figure 7.7.

Since RUCM requires the interaction between a system and an actor to be clearly described without omitting some information about its sender and receiver, it is therefore easy to identify interactions between constituents from the use case models. Figure 7.6 shows the identified interactions in our VLU system, which are twelve interactions. Further, the solid lines represent DTIs, and the dashed lines represent DMIs.

Machine Rules Specification

The restricted use case flow sentences (e.g., in basic and/or alternative flows) can to a large extent facilitate the transformation from use case models to analysis models [17]. Based on the sentences specified in use case flows, we recommend the following sub-steps to specify the eTASM machine rules:

- *Identification of possible states of the corresponding constituent*: The possible states of a constituent can be identified via analyzing the adjectives and verbs in the use case flows. A user-defined type is used to represent the possible states, and a state variable is defined to denote the current state of the constituent.
- *Identification of the transition conditions of states*: The conditions of a certain machine rule are given, according to the pertaining values of the state variables and the transition conditions.

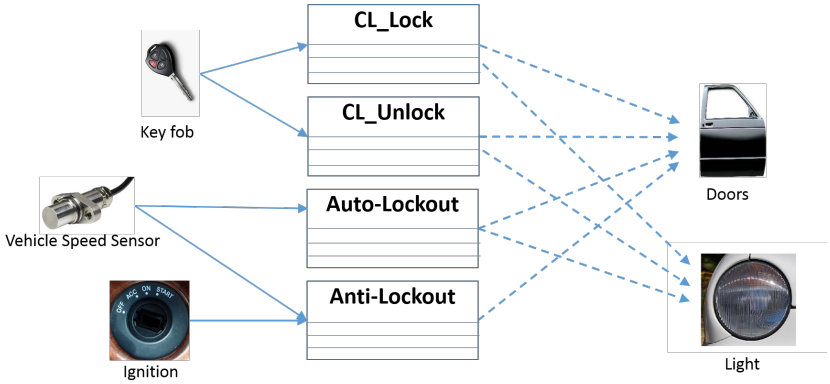

Figure 7.6: The identified interactions between constituents in the VLU system.

- *Identification of the actions when the system enters a specific state*: The actions of machine rules are specified, based on the behaviors of a constituent and the next possible state.

In our VLU application, there are five external constituents and four internal constituents under consideration, as shown in Table 7.2. The KEY_FOB machine simulates the behaviors of a user's key fob. This machine has two possible states *lock* and *unlock*, in which the *lock/unlock* state denotes that the lock/unlock command is sent to the proposed VLU system. The LIGHT machine simulates the behaviors of the vehicle light, which has two states i.e., the *flashed* state and the *off* state. The *flashed* state denotes that the light flashes for several times. The DOORS machine simulates the behaviors of the vehicle doors, which has three possible states *open*, *close* and *locked*. The IGNITION machine simulates the behaviors of the vehicle ignition, which has two possible states *haskey* and *nokey*. The VEHICLESPEEDSENSOR machine simulates the behaviors of the vehicle speed sensor, which has two possible states *still* and *running*.

The CL_LOCK machine, as shown in Figure 7.7, models the CL_Lock use case. The machine has two possible states in terms of *idle* and *lockdoor*. The Rule *ReceiveCommand* represents that the system receives the lock command from the key fob. The Rule *Locking* represents that the system locks the doors. The Rule *Idle* keeps the machine alive and represents the system is idle. The

CL_UNLOCK machine has similar rules, which will not be introduced for simplicity.

```
R1:ReceiveCommand{
  if cllock_state = idle and keyfob_cmd = lock then
    cllock_state := lockdoor;
    keyfob_cmd := NONE;
}
R2:Locking{
  t:=locking_time;
  if cllock_state = lockdoor then
    door_state := locked;
    light_state := flashed;
    cllock_state := idle;
}
R3:Idle{
  t := next;
  else  then
    skip;
}
```

Figure 7.7: The eTASM main machine models the CL_Lock use case.

The AUTO_LOCKOUT machine, as shown in Figure 7.8, models the behaviors of the Auto-Lockout use case. The Rule *Timeout* represents that when the vehicle stops and the doors are close, the feature will be activated upon the timeout expires (i.e., 20s in our case). The Rule *Autolock* represents that the system is to automatically lock the doors. The Rule *Timer* represents the timer measuring time intervals. Rule *Idle* keeps the machine alive and represents that the timer will be reset either when the doors are open or when the vehicle starts running.

The ANTI_LOCKOUT machine, as shown in Figure 7.9, models the behaviors specified in the Anti-Lockout use case. The Rule *HasKey* represents that when the vehicle stops, the feature will be activated if the key is in ignition. The Rule *Antilock* represents that the system is to unlock the doors after activated. The Rule *Idle* keeps the machine alive and represents that the system is idle.

Property Annotation

Validation of non-functional requirements in this stage relies on the estimates of the pertaining non-functional properties of the proposed system. This step can be carried out in the following ways:

```
R1:Timeout{
  if aul_state = idle and door_state = close and
    vehicle_state = still and timer = 20 then
    aul_state := timeout;
    timer   := 0;
}
R2:Autolock{
  t:=locking_time;
  if aul_state = timeout then
    door_state := locked;
    light_state:= flashed;
    aul_state := idle;
}
R3:Timer{
  t := 1;
  if aul_state = idle and door_state = close and
     vehicle_state = still and timer < 20  then
    timer := timer + 1;
}
R4:TimerReset{
  t := next;
  else  then
    timer := 0;
}
```

Figure 7.8: The eTASM main machine models the Auto-Lockout use case.

```
R1:HasKey{
  if anl_state = idle and ignition_state = haskey and
     vehicle_state = still then
    anl_state := antilock;
}
R2:Antilock{
  t:=unlocking_time;
  if anl_state = antilock then
    door_state := close;
    anl_state := idle;
}
R3:Idle{
  t := next;
  else  then
    skip;
}
```

Figure 7.9: The eTASM main machine models the Anti-Lockout use case.

- The properties are determined based upon the non-functional requirements specified in the use cases.
- The properties are determined by using the experience or analysis of

existing systems (in which estimates can be obtained by using existing well-known analysis methods, e.g., Worst-Case Execution Time (WCET) analysis [18], [19] for time duration of rules).

We annotate the aforementioned eTASM models with time durations, as shown in Figures 7.7, 7.8, and 7.9. The annotation terms *locking_time* and *unlocking_time* are either a specific value or a range of values.

7.4.3 Feature Requirements Formalization

Our approach proceeds with the formalization of feature requirements by using the eTASM Observer technique, which consists of four sub-steps as follows:

- **Step 3.1: Listener Specification** specifies the possible events sequence which represents the proposed system's observable functional behaviors and/or non-functional properties required by the feature requirements, and the corresponding actions taken on observer variables when the sequence is caught by a Listener.
- **Step 3.2: Observation Specification** formalizes a predicate depending on the observer variables. If the predicate of the Observation holds, i.e., evaluates to be true, it implies that the property satisfaction of the feature is achieved, as it can be observed in the proposed system.
- **Step 3.3: Events Filtering** identifies the interesting events and filters out the irrelevant events by specifying *EventsFilter*.
- **Step 3.4: Traceability Creation** links a specific Observer to the textual requirements. The link is used for requirements traceability from the formalization to natural language requirements in order to perform coverage checking.

In the VLU system, there are four feature requirements, i.e., *CL_Lock*, *CL_Unlock*, *Auto_Lockout* and *Anti_Lockout*. The specification of an observer is illustrated by applying the proposed steps to the *ANL* feature requirement, as depicted in Figure 7.10. To be specific, the *ANL* feature requirement states "The system shall anti-lock the doors if the key is in ignition and the vehicle has stopped", and the interesting events sequence consists of three parts. The first part "ANTI_LOCKOUT→Haskey→RuleEnableEvent" denotes that the event is triggered when the Rule *HasKey* of the ANTI_LOCKOUT machine is enabled, modeling the behavior that the key is in ignition.

The second part "[^(AUTO_LOCKOUT→Autolock→RuleEnableEvent| CL_LOCK→Locking→RuleEnableEvent)]*" represents an arbitrary number of events that are not triggered by the enabling of either the Rule *Autolock* of the AUTO_LOCKOUT machine or the Rule *Locking* of the CL_LOCK machine. Both of these two rules model the behavior that the doors are locked. The last part "ANTI_LOCKOUT→Idle-→RuleEnableEvent" represents the event that is triggered when the Rule *Idle* of the ANTI_LOCKOUT machine is enabled, which models the situation in which the key is removed. If the events sequence is detected, the Observation "ov == true" evaluates to be true, which indicates the situation in which after the key is in ignition, the doors are not locked before the key is removed, i.e., the *ANL* feature requirement is satisfied in the eTASM model.

```
ObserverVariables:{
        Boolean ov := false;
}
EventsFilter:{
        filter out: ChangeValueEvent, ResourceUsedUpEvent,
                    RuleDisableEvent;
}
Listener:{
        listening ANTI_LOCKOUT→Haskey→RuleEnableEvent
                  [^(AUTO_LOCKOUT→Autolock→RuleEnableEvent|
                     CL_LOCK→Locking→RuleEnableEvent)]
                  ANTI_LOCKOUT→Idle→RuleEnableEvent then
                ov := true;
}
Observation:{
        ov == true;
}
```

Figure 7.10: The observer for the ANL feature requirement.

7.4.4 Feature Validation

Validation of the formalized requirements aims at increasing the confidence of the validity of selected features. In this work, we assume that there is a semantic equivalence relation between the RUCM use cases and eTASM models. This is built upon the fact that the eTASM models are derived, by following the proposed modeling steps as well as our thorough understanding of the VLU system. The validation goal is achieved by following several analysis steps, based on the use of the derived eTASM models which may help to pinpoint flaws that are not trivial to detect. Such validation steps in our approach are:

- **Step 4.1: Logical Consistency Checking**. The term of logical consistency can be intuitively explained as "free of contradictions in the specifications". In our work, the logical consistency checking is performed on the executable eTASM models, by using our developed tool TASM TOOLSET. Furthermore, there are two kinds of inconsistency flaws to discover. One kind of flaws is that two rules in the same machine are enabled simultaneously, which is usually caused by the fact that there exist unpredictable behaviors in the specification of the corresponding feature. The other is that different values are assigned to the same variable simultaneously by different machines, which is usually caused by the fact that there exist hidden undesirable feature interactions in the specifications of the corresponding features.

- **Step 4.2: Coverage Checking**. The coverage checking corresponds to checking whether the feature requirements can be observed in the integrated feature specifications, which is an important activity of requirements completeness checking. To perform the coverage checking, all the feature requirements are translated into observers which monitor the execution of the features specifications, i.e., the derived eTASM models. If an *Observation* cannot hold, it indicates that although the features specifications satisfy their individual requirements in isolation, there are behavioral inconsistencies in the integrated feature specification.

- **Step 4.3: Model Checking**. The eTASM machines can be easily translated into timed automata through the transformation rules defined in [15]. The transformation enables the use of the UPPAAL model checker to verify the various properties of the eTASM model. This type of checking aims at verifying whether the eTASM model is free of deadlock and whether an expected property specified in a feature requirement is satisfied by the eTASM model. It is necessary to stress that the essential difference between *Model Checking* and *Coverage Checking* is whether a property is exhaustively checked against a model or not. Although a sound property checking is desired, in some cases *Model Checking* will encounter state explosion problem, which limits its usefulness in practice.

We follow the aforementioned validation steps to check the validity of the selected features of the VLU system. First, we use the TASM TOOLSET to perform *Logical Consistency Checking* on the formalized eTASM model. Two inconsistencies are detected, one of which is that the Rule *Autolock* of the

AUTO_LOCKOUT machine and the Rule *Antilock* of the ANTI_LOCKOUT machine update the *door_state* variable simultaneously with different values. An analysis of the inconsistency reveals: When the key is in ignition, the ANL feature will keep the doors unlocked. Meanwhile, if the autolock timeout expires, the AUL feature will try to lock the door. Since no rules are explicitly specified in the selected features to handle this situation, undesirable behaviors will occur. The other inconsistency is detected in a similar situation where the Rule *Locking* of the CL_LOCK machine and the Rule *Antilock* of the ANTI_LOCKOUT machine update the *door_state* variable simultaneously with different values. In this work, we correct such inconsistencies by assigning a higher priority (as an extra condition of the corresponding rule) to the Rule *Antilock*, which guarantees that it will be executed at first when both of two rules are enabled at the same time. Note that there are some other methods that can be used to remove the discovered inconsistencies, which are however out of the scope of this paper.

After the removal of the inconsistencies, we proceed to *Coverage Checking*. The TASM TOOLSET is applied, and the result has shown that the observations of all eTASM observers are met. Therefore, the integrated features specifications satisfies the feature requirements, from the *Coverage Checking* perspective.

On the note about *Model Checking*, we first translate the eTASM model into timed automata, and then check the deadlock property as well as the feature requirements via UPPAAL. The corresponding results are: 1) *Deadlock free* is *satisfied* and, 2) the *CL_Lock feature requirement* is *satisfied* and, 3) the *CL_Unlock feature requirement* is *satisfied* and, 4) the *AUL feature requirement* is *satisfied* and, 5) the *ANL feature requirement* is *satisfied*. As a result, the satisfaction of deadlock-free and feature requirements has been achieved.

In summary, our approach has found *two* behavioral inconsistencies in the integrated features specifications. Although the VLU system is not complex, it is enough, as an illustrative example, to show how to perform feature-oriented requirements validation by following our proposed approach.

7.5 Related Work

Kimbler et al. [20] introduce a user-oriented approach to feature interaction analysis. It aims first at creating use case models to describe different possible ways of using the system services, and then building service usage models which simulate the dynamic relations between services. This work is quite

similar with our idea, however its focus is in the telecommunication domain. Moreover, we use the RUCM approach to facilitate the transformation from use case models to subsequent formalisms. Eriksson et al. [21] propose a software product line use case modeling approach i.e., PLUSS to modeling SPL. The difference between their work and ours is the purpose of using use cases. PLUSS aims to utilize use cases to capture variants of SPL, while our approach utilizes use cases to specify behavioral specifications of features. The white paper of EAST-ADL [16] mentions that use cases can be used to specify features but no more details were given. In this work, we have provided a set of steps to specify features and perform requirements validation.

Amyot et al. [10] propose an approach to detecting feature interactions of telecommunication systems, by using Use Case Maps (UCMs) for designing features, and LOTOS for the formal specification of features. Sampath et al. [22] present a formal specification and analysis method for automotive features in the early stages of software development process. This method starts with an empty specification, and then incrementally adds clauses to the specification until all the feature requirements are satisfied. Arora et al. [7] propose a method and algorithms for identifying and resolving feature interactions in the early stages of the software development life-cycle. The work uses *State Machines* to model the behavior of independent features, context diagrams to integrate independent features, and Live Sequence Charts to capture the interactions of features.

7.6 Conclusions and Future Work

In this paper, we have proposed a novel approach to feature-oriented requirements validation by using the RUCM approach and the eTASM language. Our approach 1) specifies the behaviors and requirements of features in the RUCM use cases and, 2) transforms such RUCM use cases to the formal yet literate eTASM models and, 3) performs the requirements validation by using the TASM TOOLSET and the model checker UPPAAL. Our illustration application using a Vehicle Locking-Unlocking (VLU) system has shown that our approach can achieve the goal of feature-oriented requirements validation via *Logical Consistency Checking*, *Coverage Checking*, and *Model Checking*.

As inspired by Scandurra et al. [17] showing the promise of rule-based transformation from RUCM use cases to analysis models, we will in the future combine the proposed modeling approach with such rule/pattern-based algorithms, to achieve a fully automatic transformation between the RUCM use

cases and eTASM models. We are also interested in integrating our approach and related tools for the development of correct-by-construction systems (e.g., developed by EAST-ADL language) in a seamless and cost efficient way. Another part of future work also includes a wider industrial validation of our approach, as well as the improvement of our current TASM TOOLSET.

Bibliography

[1] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Nov 1990.

[2] Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. Addison-Wesley Professional, New York, NY, USA, May 29, 2000.

[3] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. What's in a feature: A requirements engineering perspective. In *Proceedings of FASE'08/ETAPS'08*, pages 16–30, 2008.

[4] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated reasoning on feature models. In *Proceedings of CAISE'05*, pages 491–503. Springer-Verlag, 2005.

[5] Xin Peng, Wenyun Zhao, Yunjiao Xue, and Yijian Wu. Ontology-based feature modeling and application-oriented tailoring. In *Proceedings of ICSR'06*, pages 87–100, 2006.

[6] Marcilio Mendonca, Andrzej Wkasowski, and Krzysztof Czarnecki. Sat-based analysis of feature models is easy. In *Proceedings of SPLC'09*, pages 231–240, 2009.

[7] S. Arora, P. Sampath, and S. Ramesh. Resolving uncertainty in automotive feature interactions. In *Proceedings of RE'12*, pages 21–30, Sep 2012.

[8] Ahmed F. Layouni, Kenneth J. Turner, and Luigi Logrippo. Conflict detection in call control using firstorder logic model checking. In *Proceedings of ICFI'07*, 2007.

[9] Sven Apel and Christian Kstner. An overview of feature-oriented software development, 2009.

[10] D. Amyot, L. Charfi, N. Gorse, T. Gray, L. Logrippo, J. Sincennes, B. Stepien, and T. Ware. Feature description and feature interaction analysis with use case maps and lotos. In *Proceedings of FIW'00*, pages 274–289, 2000.

[11] Michael Poppleton. Towards feature-oriented specification and development with event-b. In *Proceedings of REFSQ'07*, pages 367–381, 2007.

[12] Tao Yue, Lionel C. Briand, and Yvan Labiche. A use case modeling approach to facilitate the transition towards analysis models: Concepts and empirical evaluation. In *Proceedings of MODELS'09*, pages 484–498, 2009.

[13] Jiale Zhou, Yue Lu, and Kristina Lundqvist. A tasm-based requirements validation approach for safety-critical embedded systems. In *Proceedings of Ada-Europe'14*, June 2014.

[14] Object M. Group. OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2. Technical report, November 2007.

[15] M. Ouimet. *A formal framework for specification-based embedded real-time system engineering*. PhD thesis, Department of Aeronautics and Astronautics, MIT, 2008.

[16] Hand Blom, Henrik Lönn, Frank Hagl, Yiannis Papadopoulos, Mark-Oliver Reiser, Carl-Johan Sjöstedt, De-Jiu Chen, and Ramin T. Kolagari. EAST-ADL - An Architecture Description Language for Automotive Software-Intensive Systems. Technical report, The EAST-ADL 2 Consortium, 2012.

[17] Patrizia Scandurra, Andrea Arnoldi, Tao Yue, and Marco Dolci. Functional requirements validation by transforming use case models into abstract state machines. In *Proceedings of SAC'12*, pages 1063–1068, NY, USA, 2012. ACM.

[18] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, April 2008.

[19] Yue Lu. *Pragmatic Approaches for Timing Analysis of Real-Time Embedded Systems*. PhD thesis, Mälardalen University, June 2012.

[20] Kristofer Kimbler and Daniel Søbirk. Use case driven analysis of feature interactions. In *Feature Interactions in Telecommunications Systems, IOS*, pages 167–177, 1994.

[21] Magnus Eriksson, Jürgen Börstler, and Kjell Borg. The pluss approach: Domain modeling with features, use cases and use case realizations. In *Proceedings of SPLC'05*, pages 33–44, 2005.

[22] P. Sampath, S. Arora, and S. Ramesh. Evolving specifications formally. In *Proceedings of RE'11*, pages 5–14, Aug 2011.

Chapter 8

Paper D: The Observer-based Technique for Requirements Validation in Embedded Real-time Systems

Jiale Zhou, Yue Lu, Kristina Lundqvist.
Proceedings of the 1st International Workshop on Requirements Engineering and Testing (RET'14), Karlskrona, Sweden, August 2014.

Abstract

Model-based requirements validation is an increasingly attractive approach to discovering hidden flaws in requirements in the early phases of systems development life cycle. The application of using traditional methods such as model checking for the validation purpose is limited by the growing complexity of embedded real-time systems (ERTS). The observer-based technique is a lightweight validation technique, which has shown its potential as a means to validate the correctness of model behaviors. In this paper, the novelty of our contribution is three-fold: 1) we formally define the observer constructs for our formal specification language namely the Timed Abstract State Machine (TASM) language and, 2) we propose the Events Monitoring Logic (EvML) to facilitate the observer specification and, 3) we show how to execute observers to validate the requirements describing the systems functional behaviors and non-functional properties (such as timing) of ERTS. We also illustrate the applicability of the extended TASM language through an industrial application of a Vehicle Locking-Unlocking system.

8.1 Introduction

Studies [1] [2] have revealed that most of the anomalies discovered in late phases of systems development life cycle can be traced back to hidden flaws in the requirements specification. These deficiencies in the requirements specification will usually lead to extensive rework costs and sometimes even unrecoverable failures. For this reason, requirements validation techniques play a pivotal role in increasing the confidence that the requirements are correct in the sense of consistent and complete. This is particularly true for embedded real-time systems (ERTS) which require a set of stringent requirements to describe their functional behaviors and non-functional properties.

With the growing maturity of the model-based development paradigm, executable requirements specifications (i.e., requirements models) become increasingly attractive to cope with the boosting complexity of modern ERTS as well as to reduce the underlying anomalies. In this scenario, requirements models with well-defined semantics can capture the intended behavior of the system and thus are used as the source of information for validation purpose. Model checking [3] is a rigorous approach to assuring that correctness properties hold for the system under development. In this technique, the system design derived from requirements is specified in terms of analyzable models at a certain level of abstraction. Further, requirements are formalized into verifiable queries and then fed into the models to be checked. In this way, requirements are reasoned about to resolve contradictions, and it is also verified that they are neither so strict to forbid desired behaviors, nor so weak to allow undesired behaviors. However, such validation technique suffers from the state explosion problem.

The need for a lightweight validation technique to avoid the aforementioned problem has motivated the development of our requirements validation technique via observers [4]. To be specific, we choose the Timed Abstract State Machine (TASM) language [5] as the requirements modeling language for ERTS, based upon its distinctive features in terms of the ability to specify systems' functional behaviors and non-functional properties, the low learning costs, and a toolset that supports model execution. Additionally, we extend the TASM language with the *Event* and *Observer* constructs to specify the corresponding requirements. When the TASM models are executed, they will generate a number of events reflecting the functional behaviors and non-functional properties of the system under consideration, which can be abstracted as a linear trace of events. An observer monitors the event trace and determines whether a given correctness property is satisfied by the system under consideration. In our previous work, we assume that the observer representing the

property of interest can be specified by following the logic of regular expressions [6], and the entire event trace generated by the TASM model is available before the observer starts to monitor. The monitoring algorithm is implemented in the same way as searching for the sub-traces that match the observer regular expression. However, the main drawbacks of these assumptions are two-fold: 1) the expressiveness power of regular expressions falls short of expressing unordered fixed-count events where the occurrence multiplicities of these events are pre-defined but the corresponding order is random and, 2) the monitoring algorithm used in [4] can not be applied at runtime because of the assumption that the entire event trace is pre-achieved. Therefore, improving the logic used to specify observers is of paramount importance for our validation technique to achieve success in practice.

In this paper, we enhance our observer-based requirements validation technique via proposing a new observer specification logic that originates from the Extended Regular Expressions (ERE) and introducing a rewriting-based monitoring algorithm. Especially, the novelty of our contributions are three-fold: 1) we formally define the observer constructs for our formal specification language namely the Timed Abstract State Machine (TASM) language and, 2) we propose the Events Monitoring Logic (EvML) to facilitate the observer specification and, 3) we show how to execute observers to validate the requirements describing functional behaviors and non-functional properties of ERTS. We also illustrate the applicability of our technique by using a Vehicle Locking-Unlocking (VLU) system.

The remainder of this paper is organized as follows: An introduction to the background knowledge is presented in Section 8.2. The improved observer-based technique is described in Section 8.3. Section 8.4 illustrates the applicability of the extended TASM through an industrial application of the VLU system. Section 8.5 discusses the related work, and finally concluding remarks and future work are drawn in Section 8.6.

8.2 Background

In this section, we briefly introduce the formal specification language TASM used in our validation approach and ERE for better understanding of our work.

8.2.1 Timed Abstract State Machine

TASM [5] is a formal language for the specification of ERTS, which extends the Abstract State Machine (ASM) [7] with the capability of modeling timing properties and resource consumption of the target system. TASM inherits the easy-to-use feature from ASM, which is a literate specification language understandable and usable without extensive mathematical training [8]. A TASM model consists of two parts – an environment and a set of main machines. The environment defines the set and the type of variables, and the set of named resources which machines can consume. The main machine is made up of a set of monitored variables which can affect the machine execution, a set of controlled variables which can be modified by machines, and a set of machine rules. The set of rules specify the machine execution logic in the form of "if *condition* then *action*", where *condition* is an expression depending on the monitored variables, and *action* is a set of updates of the controlled variables. We can also use the rule "else then *action*" which is enabled merely when no other rules are enabled. A rule can specify the annotation of the time duration and resource consumption of its execution. The duration of a rule execution can be the keyword *next* that essentially states the fact that time should elapse until one of the other rules is enabled. TASM describes the basic execution semantics as the computing steps with time and resource annotations: In one step, it reads the monitored variables, selects a rule of which *condition* is satisfied, consumes the specified resources, and after waiting for the duration of the execution, it applies the update set instantaneously. If more than one rules are enabled at the same time, it non-deterministically selects one to execute. As a specification language, TASM supports the concepts of parallelism which stipulates TASM machines are executed in parallel, and hierarchical composition which is achieved by means of auxiliary machines which can be used in other machines. There are two kinds of auxiliary machines - *function* machines which can take environment variables as parameters and return execution result, and *sub* machines which can encapsulate machine rules for reuse purpose [5]. Communication between machines, including main machines and auxiliary machines, can be achieved by defining corresponding environment variables.

8.2.2 The Extended Regular Expressions

The Extended Regular Expressions (ERE) [6] represent a succinct and useful technique to specify patterns in strings by inductively utilizing the *union* $(+)$,

concatenation ($\cdot$), *repetition* ($*$) and *complementation* ($\wedge$) operators. There are programming and/or scripting languages, such as Perl and Java, which are mostly based on efficient implementations of pattern matching via ERE. Because of their convenience in specifying patterns, ERE have many applications including but not limited to text searching. The observer-based technique (a.k.a. runtime monitoring or runtime verification) is one of the application areas of interest for ERE. Since the running behaviors of computer programs or executable models can usually be abstracted as a linear trace of events or system states, the main idea behind the observer-based technique is to specify a set of observers (i.e., the extended regular expressions) that monitor the received events or system states and report abnormalities. Then, the monitoring process can be regarded as solving the membership problem for an extended regular expression R and a given word $\omega = a_1 a_2 \ldots a_n$ (a_n represents an event or a system state), which is to decide whether ω is in the regular language generated by R.

The observer-based technique usually assumes that the events or system states are received incrementally, i.e., each event is supposed to be processed as it arrives. An efficient implementation of the incremental membership problem are of critical importance to this application. A rewriting-based algorithm has been proposed in [9] for monitoring system events. The intuition is that in order to incrementally check the membership of an incoming trace of events to a given extended regular expression, the algorithm can process the events as soon as they are available by rewriting the extended regular expression contingently. Since the event is consumed incrementally in this way, the event consumption idea is more suitable for runtime monitoring, comparing the monitoring algorithm used in [4].

8.3 The Extension of TASM

In this section, we introduce the extension of TASM in terms of the fundamental concepts, the Events Monitoring Logic, and the observer execution process.

8.3.1 The Fundamental Concepts

The extended constructs comprise two main parts, i.e., *TASM Event* and *TASM Observer* as shown in Figure 8.1, which defines the meta-model of the extended TASM language.

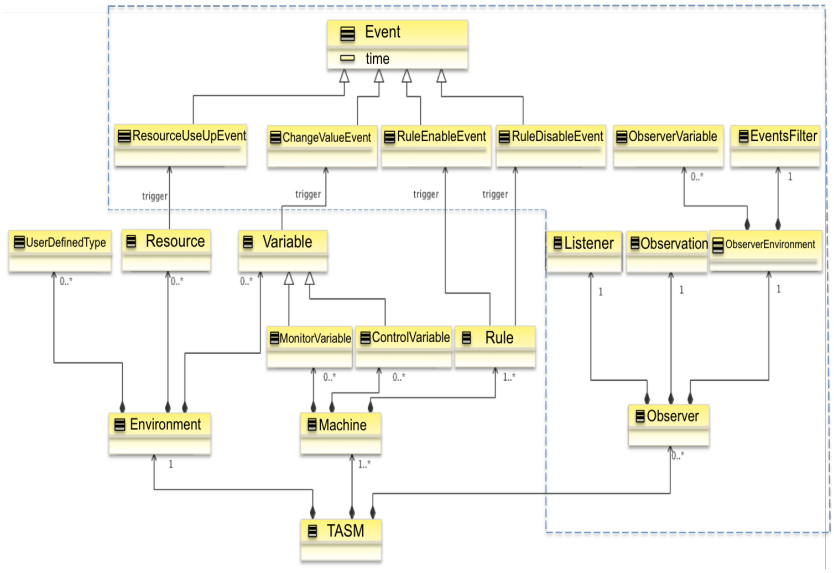

Figure 8.1: The Meta-model of the extended TASM language.

Definition 6. *TASM Event. An event* e *is a tuple* $<E, t, r_1, r_2...>$*, where* E *defines the type of an* e *in the sense of ResourceUsedUpEvent (ReUUE), ChangeValueEvent (ChVE), RuleEnableEvent (RuEE), and RuleDisableEvent (RuDE),* t *records the time stamp when the event occurs,* r_1*,* r_2*, etc. denotes the possible consumed resources by the event.*

In our extension, an event e which is triggered by the corresponding TASM construct c, can be referenced in the form of $c \rightarrow e$. To be specific, the event of $ChangeValueEvent$ type triggered by a specific TASM environment variable whenever its value is updated, is referenced in the form of *VariableName→ChVE*. The $ResourceUsedUpEvent$ type triggered by the case whenever the resource of the application is consumed up, is referenced in the form of *ResourceName→ReUUE*. The $RuleEnableEvent$ (resp. $RuleDisableEvent$) type triggered whenever a specific TASM rule is enabled (resp. disabled), is referenced in the form of *MachineName→RuleName→RuEE* (resp. *MachineName→RuleName→RuDE*). Similarly, the time stamp t of an e is referenced in the form of $e \rightarrow t$ and the consumed resource r in the form of $e \rightarrow r$. Examples illustrating how a certain

event is referenced can be found in Figure 8.10.

Here are some useful definitions which are related to *TASM Event*:

Definition 7. *Event trace. An event trace is a finite sequence of events, denoted by* $\omega = e_1 e_2 \dots e_n$.

Definition 8. *Event pattern. An event pattern is an expression following a certain logic to describe a set of event traces of interest in a compact and succinct way, denoted by* E. *The set of the event traces of interest (i.e., matching the EvML expression* E*), are denoted by* $\mathcal{L}(E)$.

The TASM Observer monitoring events is defined as comprising:

Definition 9. *TASM Observer. An observer* ob *is a tuple* $<OE, L, Obv>$, *where:*

- OE *denotes the* $ObserverEnvironment$, *which is defined as a tuple* $<OV, TU, EF>$, *where*
  - OV *denotes the* $ObserverVariables$, *which defines a set of local typed variables that can only be used by the* L *and* Obv *defined in this observer,*
  - TU *denotes the* $TypeUniverse$, *which is a set of types that include the TASM primitive types (i.e.,* $Reals$, $Integers$, $Boolean$ *and User-defined) and the TASM extended types in terms of* $Time$ *and* $Resource$,
  - EF *denotes the* $EventsFilter$, *which defines a set of events considered to be relevant or irrelevant to the observer.*
- L *denotes the* $Listener$, *which is in the form of "**listening** keyword: condition **then** action", where the keyword can be either compulsory or optional which will be further explained in Section 8.3.3, the condition specifies the event pattern following the Events Monitoring Logic (EvML) which will be defined in Section 8.3.2, and the action is a set of actions updating the value of observer variables when the condition evaluates to be true.*
- Obv *denotes the* $Observation$, *which is a predicate representing the properties needed to validate. An observation can evaluate to be either true or false, depending on the value of corresponding observer variables.*

8.3.2 The Events Monitoring Logic

The Events Monitoring Logic (EvML) is inspired by the extended regular expressions (ERE) that have been widely applied to solve the pattern matching problem. Although ERE can provide a compact and powerful implementation of pattern matching, we encounter an issue when we use it in our case. When specifying an event pattern by using ERE, the occurrence order of events is implied in the expression, e.g., $E = e_1 e_2$ implies that the event trace, where the occurrence of event e_1 is immediately followed by the occurrence of event e_2, will match the pattern. However, in some cases, the occurrence order of events is trivial. For instance, when we monitor the synchronization of two events, we are merely concerned about whether both events do occur in the event trace, rather than which event comes earlier. To describe an event trace like this (i.e., unordered event trace, hereafter), ERE have to list all the possibilities of the occurrence order, which is a clumsy and error-prone task.

Therefore, we formally define a logic, namely the Events Monitoring Logic, which can specify an unordered event trace in a more elegant way. EvML inherits the basic syntax and semantics from ERE, which defines the set of interested events by inductively applying *union* ($+$), *concatenation* ($\cdot$), *repetition* ($*$), and *complementation* ($^\wedge$) operators. To solve the aforementioned issue, we introduce a new delimiter *parallel* and the *event multi-set expression* into EvML, denoted as $\|M\|$, in order to specify unordered event traces.

The EvML Syntax

For an alphabet Σ whose elements are the possible events, an EvML expression E over Σ is defined as follows:

$$E ::= \emptyset | \epsilon | e | E + E | E \cdot E | E^* | ^\wedge E | \| M \|,$$

where $\emptyset$ denoting the empty set, ϵ denoting an empty event, and $e \in \Sigma$ denoting a regular event. M denotes the *event multi-set expression* over Σ, which is defined as a tuple $<\mathcal{A}, m>$:

- $\mathcal{A}$ denotes the underlying set of events composing the unordered event trace,
- $m : \mathcal{A} \rightarrow Mul_{\geq 0}$ is a function indicating the multiplicity of the occurrences of the event $e_\mathcal{A} \in \mathcal{A}$ in the event trace, denoted as $m(e_\mathcal{A}) \in Mul_{\geq 0} = \{0, 1, 2, 3, \cdots\} \cup \{*\}$. The repetition operator ($*$) denotes that the number of an event $e_\mathcal{A} \in \mathcal{A}$ is not explicitly-defined, which can be any number $n \in Mul_{\geq 0}$.

We define a set of additional rules to further facilitate the specification of event pattern:

- In order to keep the expression succinct, we define a meta-character (.) to represent any event in the alphabet Σ.
- The concatenation operators between EvML expressions can be omitted for simplicity (i.e., $E_1 \cdot E_2$ can be denoted as $E_1 E_2$),
- The operators ($^\wedge$) for complementation, ($*$) for repetition, ($\cdot$) for concatenation, and ($+$) for union in the EvML expression are defined in decreasing order of precedence,
- The delimiter "()" for parentheses can increase the precedence of the braced operators,
- The multi-set expression $\|M\|$ can be written in the form of $\|\{e_1, e_2, \dots\}, \{m(e_1), m(e_2), \dots\}\|$

The EvML Semantics

Some new notions and notations are needed before we can define the EvML semantics. For any given event trace ω, we assume that it is easy to calculate the underlying set of events composing the trace (denoted as $\mathcal{C}_\omega$) and the number of occurrences of a given event $e \in \omega$ (denoted as $n_\omega(e)$). The semantics of EvML is defined as shown in Figure 8.2.

$$
\begin{aligned}
\mathcal{L}(\emptyset) &= \emptyset \\
\mathcal{L}(\epsilon) &= \{\epsilon\} \\
\mathcal{L}(e) &= \{e\} \\
\mathcal{L}(E_1 + E_2) &= \mathcal{L}(E_1) \cup \mathcal{L}(E_2) \\
\mathcal{L}(E_1 \cdot E_2) &= \{\omega_1 \cdot \omega_2 | \omega_1 \in \mathcal{L}(E_1) \; and \; \omega_2 \in \mathcal{L}(E_2)\} \\
\mathcal{L}(E^*) &= (\mathcal{L}(E))^* \\
\mathcal{L}(^\wedge E) &= \Sigma^* \backslash \mathcal{L}(E) \\
\mathcal{L}(\|M\|) &= \{\omega | \mathcal{C}_\omega = \mathcal{A}_M \; and \; \forall e \in \omega, n_\omega(e) = m_M(e)\}
\end{aligned}
$$

Figure 8.2: The semantics of EvML

Note that the operator $(+)$ is associative and commutative, and the operator $(\cdot)$ is associative.

$$\begin{aligned}(E_1 + E_2) + E_3 &\equiv E_1 + (E_2 + E_3)\\ E_1 + E_2 &\equiv E_2 + E_1\\ (E_1 \cdot E_2) \cdot E_3 &\equiv E_1 \cdot (E_2 \cdot E_3)\end{aligned}$$

According to the EvML semantics, several simplification equations are defined, including the following:

$$\begin{aligned}E + \emptyset &\equiv E\\ E + E &\equiv E\\ E_1 \cdot E_3 + E_2 \cdot E_3 &\equiv (E_1 + E_2) \cdot E_3\\ \epsilon \cdot E &\equiv E\end{aligned}$$

Figure 8.3 shows some examples to illustrate the EvML semantics.

Examples:
Assume that $\Sigma=\{\epsilon, e_1, e_2, e_3\}$.

"$e_1 + e_2 *$" **denotes** $\{\epsilon, "e_1", "e_2", "e_2e_2", "e_2e_2e_2" \dots\}$

"$(e_1 + e_2) *$" **denotes** $\{\epsilon, "e_1", "e_2", "e_1e_1", "e_1e_2", "e_2e_2", "e_2e_1", "e_1e_1e_1" \dots\}$

"$\|\{e_1, e_2, e_3\}, \{1, 1, 1\}\|$" **denotes** $\{"e_1e_2e_3", "e_1e_3e_2", "e_2e_1e_3", "e_2e_3e_1", "e_3e_1e_2", "e_3e_2e_1"\}$

"$.e_1$" **denotes** $\{"e_1", "e_1e_1", "e_2e_1", "e_3e_1"\}$

"$\hat{}e_1$" **denotes** $\{"e_2", "e_3"\}$

Figure 8.3: The examples of the Events Monitoring Logic

The EvML Operational Semantics

In this work, the event pattern matching algorithm is based on the event consumption idea as well, in the sense that the EvML expression E can consume an event e in the trace and produces another EvML expression denoted as $E\{e\}$, with the property that for any trace ω, $e \cdot \omega \in \mathcal{L}(E)$ if and only if $\omega \in \mathcal{L}(E\{e\})$. Roşu et al. [9] presented a set of rewriting rules and a rewriting-based algorithm to implement the event consumption idea for ERE. In our case,

$$
\begin{aligned}
(E_1 + E_2)\{e\} &\rightarrow E_1\{e\} + E_2\{e\} && (8.1)\\
(E_1 \cdot E_2)\{e\} &\rightarrow (E_1\{e\}) \cdot E_2 + \mathit{if}\ (\epsilon \in \mathcal{L}(E_1))\ \mathit{then}\ E_2\{e\}\ \mathit{else}\ \emptyset\ \mathit{fi} && (8.2)\\
(E^*)\{e\} &\rightarrow (E\{e\}) \cdot E^* && (8.3)\\
(^\wedge E)\{e\} &\rightarrow {}^\wedge(E\{e\}) && (8.4)\\
e_1\{e\} &\rightarrow \mathit{if}\ (e_1 = e)\ \mathit{then}\ \epsilon\ \mathit{else}\ \emptyset\ \mathit{fi} && (8.5)\\
\epsilon\{e\} &\rightarrow \emptyset && (8.6)\\
\emptyset\{e\} &\rightarrow \emptyset && (8.7)\\
\|M\|\{e\} &\rightarrow \mathit{if}\ (e \in \mathcal{A}_M \text{ and } m_M(e) \neq 0)\ \mathit{then}\ m_M(e) = m'_M(e)\ \mathit{else}\ \emptyset\ \mathit{fi} && (8.8)
\end{aligned}
$$

Figure 8.4: The rewriting rules for EvML

since EvML is a further extended version of ERE, we can easily adapt their work to the Events Monitoring Logic. We give the rewriting rules which define the EvML operational semantics recursively, using the structure of the EvML expression, as shown in Figure 8.4. In particular, the Rules 8.1 to 8.7 are defined for the inherited ERE operators. The Rule 8.8 defines that when the available event is found in the specified event multi-set, if the occurrence number of the event is explicitly-defined, the number is decreased by one; otherwise, the number remains not explicitly-defined, where:

$$
m'_M(e) = \begin{cases} n-1 & , m_M(e) = n \text{ and } n > 0 \\ * & , m_M(e) = * \end{cases}
$$

and once all of the explicitly-defined occurrence numbers decrease to zero, we have the rewriting rule:

$$
\begin{aligned}
\|M\| \quad \rightarrow \quad & \mathit{if}\ (\forall e \in \mathcal{A}_M, m_M(e) = 0 \text{ or } *) \\
& \mathit{then}\ \epsilon\ \mathit{else}\ \|M\|\ \mathit{fi}
\end{aligned} \tag{8.9}
$$

The structure "*if then else*" taking a boolean term and two EvML expressions is defined by two rewriting rules:

$$
\begin{aligned}
\mathit{if}\ (\mathit{true})\ \mathit{then}\ E_1\ \mathit{else}\ E_2\ \mathit{fi} &\rightarrow E_1 && (8.10)\\
\mathit{if}\ (\mathit{false})\ \mathit{then}\ E_1\ \mathit{else}\ E_2\ \mathit{fi} &\rightarrow E_2 && (8.11)
\end{aligned}
$$

For the evaluation of the boolean expression $\epsilon \in \mathcal{L}(E)$, we define the fol-

lowing rules:

$$\epsilon \in (\mathcal{L}(E_1) + \mathcal{L}(E_2)) \rightarrow \epsilon \in \mathcal{L}(E_1) \vee \epsilon \in \mathcal{L}(E_2) \quad (8.12)$$

$$\epsilon \in (\mathcal{L}(E_1) \cdot \mathcal{L}(E_2)) \rightarrow \epsilon \in \mathcal{L}(E_1) \wedge \epsilon \in \mathcal{L}(E_2) \quad (8.13)$$

$$\epsilon \in (\mathcal{L}(E^*)) \rightarrow true \quad (8.14)$$

$$\epsilon \in (\mathcal{L}(^\wedge E)) \rightarrow not\ (\epsilon \in \mathcal{L}(E)) \quad (8.15)$$

$$\epsilon \in \mathcal{L}(e) \rightarrow false \quad (8.16)$$

$$\epsilon \in \mathcal{L}(\epsilon) \rightarrow true \quad (8.17)$$

$$\epsilon \in \mathcal{L}(\emptyset) \rightarrow false \quad (8.18)$$

$$\epsilon \in \mathcal{L}(\|M\|) \rightarrow false \quad (8.19)$$

Since we also use the meta-character (.) to specify EvML expressions for simplicity, we have the rewriting rule for it:

$$.\{e\} \rightarrow \epsilon \quad (8.20)$$

Additionally, the Rule 8.20 for the meta-character (.) is a special case of the Rule 8.5, where $e_1 \equiv e$.

These rewriting rules are natural and intuitive. We omit the proof of the *terminating* and *Church-Rosser* property of the rewriting system, and leave it as our future work.

The Event Pattern Matching Algorithm

After introducing the operational semantics of EvML, we present the Algorithm 2 that describes the event pattern matching algorithm. The algorithm will be used in the observer execution process (as stated in Section 8.3.3) to determine that the *first m-events* trace ($\omega_m = e_1 e_2 \ldots e_m$ where $m \leq n$) of an input trace ($\omega = e_1 e_2 \ldots e_n$) matches the event pattern:

8.3.3 The Observer Execution Process

In this section, we introduce the observer execution process operated to enable an observer working with a given event trace. Briefly speaking, with the execution of the TASM model at runtime, different TASM constructs will generate massive events which can be abstracted as a linear sequence of events. The observer can spawn one or more child observers, together to determine the satisfaction of the upcoming events with the event pattern and to evaluate corresponding observations.

Algorithm 2 $EventPatternMatching(E, \omega)$

```
INPUT:  An EvML expression E and an event trace
                          ω = e1 e2 ... en .
OUTPUT: match when ωm = e1 e2 ... em (m≤n) matches E ;
        nomatch when ωm does not match E

let E' be E;                          % start the matching process
let m be 1;
while m≤ n do
        wait until em is available;
        let E' be E'{em};             % consume one event
        if ϵ ∈ L(E') then
                return match;         % the first m-events trace matches E
        if E' = ∅ then                % the current event does not match E
                return nomatch;       % the input trace does not match E
        let m be m+1;                 % consume the next event
return nomatch;                       % the input trace does not match E
```

In particular, the events in the trace are consumed one by one. When an event e is available, the $EventsFilter$ is applied to filter out irrelevant events to the desired property. If the available event is relevant, the expressions of the observer E_o and its child observers E_c (if any exists) will be transformed to the corresponding new expressions E'_o and E'_c by applying the rewriting rules. Regarding the parent observer, if the new expression can match ϵ (i.e., $\epsilon \in \mathcal{L}(E'_o)$), which means the $Listener$'s condition is satisfied, then the action will be executed and the observation will be concluded. If the new expression is the empty set (i.e., $E'_o = \emptyset$), which means the current event can not satisfy the $Listener$'s condition, then the event is dropped and the observer waits for the next available event. If the new expression neither is the empty set, nor matches ϵ, which means the current event is probably the first event of one of the event traces that can satisfy the $Listener$'s $condition$, then the observer will spawn a child observer that inherits the new expression (i.e., $E_c = E'_o$) and the child observer starts to wait for the next available event, as depicted in Figure 8.5.

Regarding the child observers, they will take over monitoring whether the subsequent events match the event expression E_c by applying the event pattern matching algorithm, as illustrated in Figure 8.6. When a new relevant event is available, the child observer event expression will be rewritten into E'_c:

- If the *condition* is satisfied by the subsequent trace (i.e., $\epsilon \in \mathcal{L}(E'_c)$), then the *action* will be immediately executed to update corresponding variables. The observation predicate will be concluded based on the up-

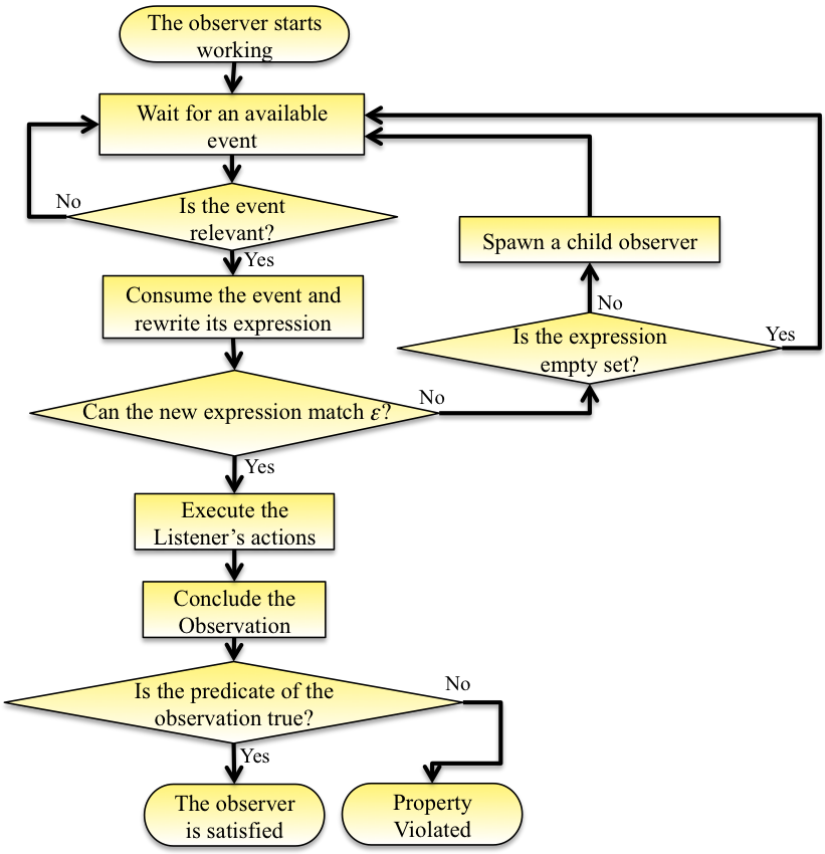

Figure 8.5: The observer execution process

dated observer variables. In this situation, if the value of the predicate is evaluated to be false, which means the represented property is deemed to be violated, its parent observer and all the other child observers will stop monitoring. On the contrary, if the predicate is evaluated to be true, the child observer is deemed to be satisfied. Then, its parent observer and all the other child observers will continue monitoring.

- If the subsequent events violate the event pattern (i.e., $E'_c = \emptyset$), one of the two possible consequences will take place, which depends on the

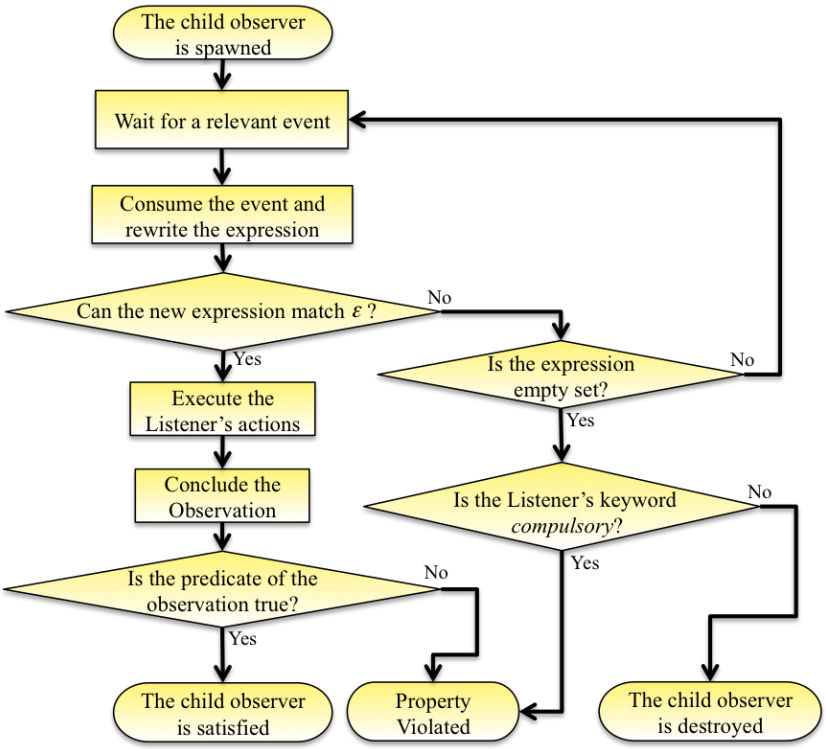

Figure 8.6: The child observer execution process

keyword of the $Listener$. If the *keyword* is specified as $optional$, then the child observer will be destroyed and its parent observer continues monitoring. If the *keyword* is $compulsory$, then the property represented by its parent observer will directly evaluate to be violated and the parent observer and all the other child observers will stop monitoring.

Note that a running TASM model can be observed by several observers at the same time. Meanwhile, an observer can have many active instances (i.e., child observers) simultaneously before the end of monitoring an event trace.

8.4 Illustration Application

In this section, we describe a simplified Vehicle Locking-Unlocking (VLU) system. This is used to illustrate how to specify the observer according to the requirement for validation purpose.

8.4.1 Vehicle Locking-Unlocking

The proposed VLU system aims at replacing the mechanical key, as a control access to a vehicle, and it follows a common pattern in feature-oriented requirements specification [10]: The basic functionality is encapsulated as an individual feature, and additional/optional enhancements are specified as features that provide increments in functionality. Specifically, such features are *Central Locking* (CL), *Auto-lockout* (AUL) and *Anti-lockout* (ANL), where:

- **Central Locking (a basic feature)** locks and unlocks all the doors of the vehicle upon receipt of a command from the user key fob.
- **Auto-lockout (an optional feature)** locks all the doors of the vehicle when a timeout expires. It provides theft protection in case that the driver forgets to manually lock the doors.
- **Anti-lockout (an optional feature)** enables unlocking of the doors while a key is in ignition. The purpose of this feature is to prevent the driver from being locked out of the vehicle.

Figure 8.7 shows the features of the VLU system in the form of technical feature model tree presented in the EAST-ADL language [11].

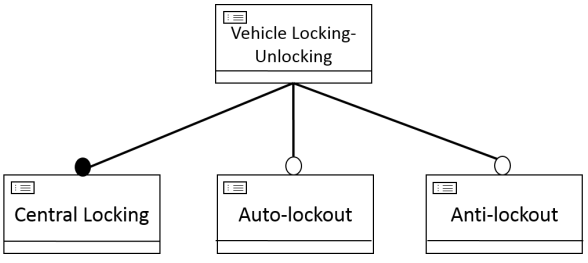

Figure 8.7: The technical feature model tree of the VLU system.

8.4.2 Observer Specification

Assume that we are interested in monitoring the satisfaction of the AUL feature requirement which states that "The system shall lock all the doors of the vehicle when the vehicle is still and a timeout ($timer = 20$ in this case) expires." This feature can show how to use the TASM language and the extended observer-based technique to specify functional behaviors as well as non-functional properties in terms of timing property in this case. In addition, assume that we have two TASM machines in terms of AUL (as shown in Figure 8.8) and $DOOR$ (as shown in Figure 8.9), modeling the behaviors of the AUL feature and the doors, respectively. Recall that each event is time-stamped during the model execution, and the time stamp of the event can be obtained by referencing the time property t of the event.

```
R1:Timeout{   % the name of the rule
  if aul_state = idle and door_state = close and
     vehicle_state = still and timer = 20 then
    aul_state := timeout;
    timer   := 0;
}
R2:Autolock{
  if aul_state = timeout then
    door_action := lock;
    aul_state := idle;
}
R3:Timer{
  t := 1;     % the time duration of the rule
  if aul_state = idle and door_state = close and
     vehicle_state = still and timer < 20  then
    timer := timer + 1;
}
R4:TimerReset{
  t := next;
  else  then
    timer := 0;
}
```

Figure 8.8: The TASM machine models the behavior of the Auto-lockout feature.

The observer is specified as shown in Figure 8.10. The $EventsFilter$ will filter out the events of the ReUUE, ChVE, and RuDE types. Since the auto-locking process can be interrupted by either moving the vehicle or manually locking the doors, the keyword of the $Listener$ is $optional$. The event pattern of the $Listener$ consists of three parts, as shown in Line 9, 10, and 11 respectively. The first part "$||\{AUL \rightarrow Timer \rightarrow RuEE$,

```
R1:Close{
  t:=closing_time;   % the time duration of the rule
  if door_action = close then
    door_state := closed;
}
R2:Lock{
  t:=locking_time;   % the time duration of the rule
  if door_action = lock then
    door_state := locked;
}
R3:UnLock{
  t:=unlocking_time; % the time duration of the rule
  if door_action = unlock then
    door_state := closed;
}
R4:Open{
  t := opening_time; % the time duration of the rule
  if door_action = open then
    door_state := opened;
}
```

Figure 8.9: The TASM machine models the behavior of the doors.

$^{\wedge}AUL{\rightarrow}TimerReset{\rightarrow}RuEE\}$,$\{20,\ *\}||$" models the behavior that the timer starts and then expires, where the event $AUL{\rightarrow}Timer{\rightarrow}RuEE$ is supposed to be triggered 20 times and there could be some other events but $TimerReset{\rightarrow}RuEE$ events that will be triggered during the expiring process. The second part ".*" represents an arbitrary number of arbitrary events that could be triggered by other TASM machines after the timer expires but before the doors are locked. The last part "$DOOR{\rightarrow}Lock{\rightarrow}RuEE$" models the behavior that the doors are locked. If the event pattern is matched, the observation "obt2$-$obt1$==$20$+DOOR{\rightarrow}Lock{\rightarrow}locking_time$" will be evaluated accordingly, which indicates whether the doors are locked properly when the timer expires.

If the observer is violated, it means there must exist some inconsistencies in the requirements. Those inconsistencies cause the doors not being locked properly when the timer expires.

```
1 ObserverVariables:{
2  Time obt1 := 0; Time obt2:=0;
3 }
4 EventsFilter:{
5  irrelevant event types: ReUUE, ChVE, RuDE;
6 }
7 Listener:{
8  listening optional:
9  ||{AUL→Timer→RuEE,^AUL→TimerReset→RuEE},{20,*}||
10  .*
11  DOOR→Lock→RuEE then
12
13  obt1:=AUL→Timer→RuEE(1)→t; %the stamped time of the first event
14                                 of the AUL→Timer→RuEE type
15
16  obt2:=DOOR→Lock→RuEE→t;
17 }
18 Observation:{
19  obt2-obt1 == 20+DOOR→Lock→locking_time;
20 }
```

Figure 8.10: The Observer for the AUL feature requirement

8.5 Related Work

8.5.1 The Monitoring Logic

Giannakopoulou et al. [12] present an approach to checking a running program against Linear Temporal Logic (LTL) specifications. In particular, the LTL formulae representing the properties of interest are translated into finite-state automata, which are used as observers monitoring the program behaviors.

Roşu et al. [9] present lower bounds and rewriting algorithms for testing membership of a word in a regular language described by an extended regular expression. The algorithms are based on an event consumption idea: a just arrived event is consumed by the regular expression, i.e., the extended regular expression modifies itself into another expression dropping the event.

Barringer et al. [13] present a compact and powerful logic, namely Eagle, which is based on recursive parameterized rule definitions over the standard propositional logic operators together with three primitive temporal operators in the sense of a past-state operator, a next-state operator, and a concatenation-state operator.

Basin et al. [14] extend the metric first-order temporal logic (MFOTL) with aggregation operators in order to specify observers that represent the compliance policies on aggregated data. Compliance policies represent normative

regulations, which specify permissive and obligatory actions for system users. The authors provide a monitoring algorithm for the enriched observer specification language as well.

Comparing the aforementioned observer-based techniques, EvML has a more succinct way to express unordered fixed-count events sequence. Moreover, the observers defined in the aforementioned techniques merely use logical expressions to specify the property of interest, but we use the combination of the logical expression (i.e., $Listener'scondition$) and other constructs (i.e., $Listener'saction$, $ObserverEnvironment$ and $Observation$). By using the combination, more expressiveness power is possible to achieve, which will be discussed in detail as our future work.

8.5.2 Other Related Work

Bauer et al. [15] discuss a three-value semantics (false, true, inconclusive) for LTL and TLTL observers on finite traces, where an observer outputs *false* when a finite prefix is impossible to be the prefix of any accepting trace and, *true* when a finite prefix can be accepted by any infinite extension of the trace and, *inconclusive* in other cases. Additionally, Falcone et al. [16] give an related and interesting discussion about the monitorability of properties in the safety-progress classification. Leucker et al. [17] present a brief account of the field of runtime monitoring. They give a definition of runtime monitoring and make a comparison to well-known verification techniques in terms of model checking and testing.

8.6 Conclusion

In this paper, we have enhanced our observer-based requirements validation technique presented in [4] via a proposed new observer specification logic (namely EvML), as well as a newly introduced rewriting-based monitoring algorithm for EvML. EvML originates from the extended regular expressions, and can help to specify the situation in which the occurrence number of the events of interest is predefined and the occurrence order is trivial. The rewriting-based monitoring algorithm implements the incremental event consumption idea which enables runtime monitoring. Our illustration application using a Vehicle Locking-Unlocking system has shown that EvML is capable to specify observers for validation purpose. As a part of our future work, we are interested in having more extensive industrial cooperations for validating our

observer-based technique, as well as improving the current implementation of our TASM TOOLSET.

Bibliography

[1] AdrianF. Ellis. Achieving safety in complex control systems. In *Proceedings of SCSC'95*, pages 1–14. Springer London, 1995.

[2] Nancy G. Leveson. *Safeware: System Safety and Computers*. ACM, NY, USA, 1995.

[3] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[4] Jiale Zhou, Yue Lu, and Kristina Lundqvist. A tasm-based requirements validation approach for safety-critical embedded systems. In *Proceedings of Ada-Europe'14*, June 2014.

[5] M. Ouimet. *A formal framework for specification-based embedded real-time system engineering*. PhD thesis, Department of Aeronautics and Astronautics, MIT, 2008.

[6] Jeffrey E. F. Friedl. *Mastering Regular Expressions*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2 edition, 2002.

[7] Egon Börger and Robert F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.

[8] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, December 1996.

[9] Mahesh Viswanathan. Testing extended regular language membership incrementally by rewriting. In *Proceedings of RTA03*, pages 499–514. Springer-Verlag, 2003.

[10] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, CMU/SEI-90-TR-21, ESD-90-TR-222, November 1990.

[11] Hand Blom, Henrik Lönn, Frank Hagl, Yiannis Papadopoulos, Mark-Oliver Reiser, Carl-Johan Sjöstedt, De-Jiu Chen, and Ramin T. Kolagari. EAST-ADL - An Architecture Description Language for Automotive Software-Intensive Systems. Technical report, The EAST-ADL 2 Consortium, 2012.

[12] Dimitra Giannakopoulou and Klaus Havelund. Automata-based verification of temporal properties on running programs. In *Proceedings of ASE'01*, pages 412–416, Nov 2001.

[13] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *Proceedings of VMCAI'04*, pages 44–57, 2004.

[14] David Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zălinescu. Monitoring of temporal first-order properties with aggregations. In *Proceedings of RV'13*, pages 40–58, 2013.

[15] Andreas Bauer, Martin Leucker, and Christian Schallhart. Monitoring of real-time properties. In *Proceedings of FSTTCS'06*, pages 260–272. Springer, 2006.

[16] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. Runtime verification of safety-progress properties. In *Runtime Verification*, pages 40–59. Springer-Verlag, Berlin, Heidelberg, 2009.

[17] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.