DISSERTATION

A UNIFIED MODELING LANGUAGE FRAMEWORK FOR SPECIFYING AND ANALYZING TEMPORAL PROPERTIES

Submitted by

Mustafa Al Lail

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2018

ABSTRACT

A UNIFIED MODELING LANGUAGE FRAMEWORK FOR SPECIFYING AND ANALYZING TEMPORAL PROPERTIES

In the context of Model-Driven Engineering (MDE), designers use the Unified Modeling Language (UML) to create models that drive the entire development process. Once UML models are created, MDE techniques automatically generate code from the models. If the models have undetected faults, they are propagated to code where they require considerable time and effort to detect and correct. It is therefore mandatory to analyze UML models at earlier stages of the development life-cycle to ensure the success of the MDE techniques in producing reliable software. One approach to uncovering design errors is to formally specify and analyze the properties that a system has to satisfy. Although significant research appears in specifying and analyzing properties, there is not an effective and efficient UML-based framework that specifies and analyzes temporal properties.

The contribution of this dissertation is a UML-based framework and tools for aiding UML designers to effectively and efficiently specify and analyze temporal properties. In particular, the framework is composed of 1) a UML specification technique that designers can use to specify temporal properties, 2) a rigorous analysis technique for analyzing temporal properties, 3) an optimization technique to scale the analysis to large class models, and 4) a proof-of-concept tool. An evaluation of the framework using two real-world studies shows that the specification technique can be used to specify a variety of temporal properties and the analysis technique can uncover certain types of design faults. It also demonstrates that the optimization technique can significantly speed up the analysis.

ACKNOWLEDGEMENTS

All praise goes to Allah the Almighty, the Beneficent the Merciful, for guiding me and blessing me with his favors that have enabled me to reach this milestone in my life.

I am truly grateful to many people for their help and support while doing this dissertation. I would not have been able to complete the dissertation without their support and help.

I owe special thanks to my late advisor, Dr. Robert B. France, for accepting me to be his Ph.D. student and offering me Graduate Research Assistantship. Dr. France has been a tremendous source of inspiration. He has been very supportive and patient with me through the years. He guided me when I was confused, encouraged me when I was depressed, and inspired me when I did not know what to do. It was a tragedy to me, and to many others, when we lost him. Even though he is no longer with us, he will always be a source of inspiration. I am deeply grateful for his patience, guidance, and support.

I would like to express my sincere thanks to my advisor, Dr. Indrakshi Ray, for her untiring support, encouragement, and willingness to always help me. I believe she stands out not only by her scholarship activities, but also by her excellent human characteristics. She always cares for me and all her other students. I am genuinely proud to be her student and I am grateful to her for helping me in many steps through my career.

I would like to thank my Ph.D. committee members: Dr. Indrajit Ray, Dr. Geri Georg, Dr. Yashwant K. Malaiya, and Dr. Idris Samawi Hamid for their valuable advice and help. A special thanks goes to Dr. Georg for her invaluable contributions to improve the quality of this dissertation. I also would like to give a special thanks to Dr. Malaiya who not only served n my committee, but also for his invaluable constructive feedback on my teaching when I was a teaching assistant for his class. Special thanks also goes to Dr.Indrajit Ray who wrote me a letter of recommendation to be accepted to the graduate program at CSU. Many thanks goes to Dr. Hamid for being my mentor through the years here at CSU, for great intellectual conversations, and the many favors he has done for me.

I am also grateful to the faculty of the computer science department who taught me and gave me great advice and constructive feedback throughout the years, especially Dr. Sudipto Ghosh, and Dr. Jim Bieman. I would like to thank the staff of the computer science department, especially Kim, Sharon, and Wendy, for always granting me favors.

I would like to extend many thanks to my friends and fellow graduate students at CSU, specially, the members of the Software Engineering and DBSec research groups. Out of them, I would recognize Ramadan Abdulnabi and Wuliang Sun who contributed to and participated in research activities of this dissertation. I thank them for their inspiring discussions and willingness to help in all situations.

I would like to extend many thanks to my friends and fellow students from my home country here at CSU. I thank all of them for the great time, vacations, and activities that we spent together. I would like to greatly express my gratitude to my sponsor, the Ministry of Higher Education in the Kingdom of Saudi Arabia, for supporting me and covering my study expenses here in the U.S.

Last, but not least, I would like to express my deepest gratitude to my family. I start with my father, Hussain Al Lail, and my mother, Zahraa Al Salah, who raised me to love knowledge and scholarship. Unfortunately, I lost both of them while going through the different stages of the scholarship path. I would like to express my sincere gratitude to all my brothers and sisters for their love, support, encouragement, and doaa. I am especially indebted to my wife, Zainab Al Salah, for her unconditional love, support, sacrifice, and for taking care of our kids while I was away for many hours, days, and weeks, working on this dissertation. Finally, my children: Ali, Mohammed, and Fatima have been the great drive that kept me going.

DEDICATION

To The AWAITED.

To The INFALLIBLE FOURTEEN.

To my MOTHER and FATHER.

To my WIFE and KIDS.

To my SISTERS and BROTHERS.

To my FRIENDS.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Software plays a critical role in a wide variety of products from phones to air travel. However, modern software systems are complex and often include flaws, leading to various problems. For example, software errors can destroy business's reputations (e.g., IBM Pentium FDIV bug [2]), cost a significant amount of money (e.g., NASA Mars Climate Orbiter [3]), and lead to failure of critical missions (e.g., European Space Agency (ESA) Ariane 5 [4]). Errors in software can even cause tragedies, such as the death of people (e.g., Therac25 [5]). These problems pose a significant challenge to the Software Engineering community. The challenge is to create techniques and tools that assist designers in reducing, or eventually, eliminating errors in software systems, regardless of their complexity.

In this introductory chapter, we situate the research described in this dissertation in the broad area of complex software development. We then explain the problem, express the aim and the objectives, present the contributions, and list the publications that have resulted from our research.

1.1 Research Context

Software Engineering focuses on the development process of software. Software development is inherently difficult for complex solutions, as it entails creating large program codes from sophisticated customer requirements [6] [1]. As software complexity increases, so does the occurrence of problems associated with these complexities. Problems associated with software complexity occur because the existing development methods are ineffective, resulting in a situation commonly known as 'Software Crisis' [7] [2]. Techniques that require extensive handcrafting of code fail to cope with the complexity [8]. They introduce accidental complexities, i.e., complexities that are

[1]"Complexity, I would assert, is the biggest factor involved in anything having to do with the software field" a quote by Robert L. Glass [6].

[2] Fred Brooks describes software crises as the state of software development practice in which too many software projects take longer than expected, experience large operational failure, or are simply canceled [7]

not inherent in the software projects, but rather are introduced by the development techniques used. Therefore, overcoming software crisis is one of the main concerns of the Software Engineering community.

Model Driven Engineering (MDE) is a new software development methodology that is proposed to minimize the accidental complexities associated with software development [8, 9]. An MDE method shifts the focus from program coding to modeling activities that reduce the gap between customer requirements and program code through the building, refining, and maintaining of software models [3]. Software development focuses on creating and evolving models at different levels of abstraction. Abstract models, such as Platform Independent Models (PIMs), are mechanically transformed to detailed Platform Specific Models (PSMs) that are eventually used to automatically generate substantial portions of the code. Ideally, software development tools automate the various tasks with minimal human involvement.

As of 2018, MDE has been established as a modern software development methodology and has been adopted successfully in many industries including: the automotive industry, aerospace, telecommunications, and business information systems [10, 11, 12]. However, MDE is arguably still a niche technology [13], and it needs to be developed further. While some modeling languages, such as the Unified Modeling Language (UML), have become widespread [14], MDE techniques have not reached their full potential [15, 16]. According to Bran Selic, an MDE scientist, this situation has resulted due to a combination of social, economical, and technical factors [16]. Further, the use of models to automatically generate code has been relatively rare [13].

For an MDE approach to succeed, software designers must integrate the development process with practical techniques to improve model quality. If a model has unresolved design faults, they are propagated to the code where they can be more difficult to uncover and more expensive to remove. Model analysis enables design fault detection, it also facilitates better understanding of systems. Model analysis, therefore, is one of the potential benefits of MDE techniques as it

[3]A model is a simplification of the reality that still retains the elements relevant to the problem being investigated. Computer scientist Robin Milner argues that to make software engineering parallel to other engineering fields we need to “establish modeling as the basis of informatics”.

increases the designer's ability to specify properties of a system formally and analyze its models early in the development process. The property specification and analysis process reduces the possibility of producing faulty designs; hence, saving time and effort to correct at the code level.

To ensure the most rigorous analysis possible, design must take into account the need to describe software model and properties precisely. As of 2018, the state of MDE practice suggests the use of UML [17, 18] and Object Constraint Language (OCL) [19, 20]. UML and OCL are the *de facto* standards for both description of software models and specification of their properties.

UML provides a rich set of visual modeling concepts to describe the structural and behavioral aspects of software at different levels of abstraction [4]. For example, class models are structural and they explain the relationship between a collection of classes, types, and interfaces. Behavioral models, such as sequence, activity, and state machine models, can be used to describe the behavior.

OCL is the standard for specifying constraints and structural properties on UML models. For example, OCL is capable of expressing invariants, such as a class's attribute value that must not exceed a certain value. OCL can also be used to specify behavioral properties as pre- and postconditions of operations. However, OCL, based on first-order logic, lacks good support for temporal properties, types of behavioral properties that require higher order logic. Further, as of 2018, the MDE community has not produced a standard language for describing temporal properties, though there are some OCL extensions for this purpose.

The research described in this dissertation focuses on specifying and analyzing temporal properties. Temporal properties are useful in capturing a broad range of relevant system properties and requirements [22, 23]. To recognize the significance of temporal properties and their role in software specification and verification, the Association for Computing Machinery (ACM) granted Amir Pnueli the 1996 ACM Turing Award, the most prestigious award in computer science for his "seminal work introducing temporal logic into computing science ..." [24]. Temporal properties are broadly categorized into two groups: safety and liveness. Safety properties specify that

[4]David Harel presented interesting arguments for using visual representation in software development. He argues that graphical modeling languaes, such as statecharts and UML, are befinicial in communicating design among different stackholders [21].

bad things never happen, and liveness properties specify that *good things should eventually happen*. Software designers can use temporal logic formalisms, (e.g., Linear Temporal Logic [22] and Computation Tree Logic [25]) to formally specify properties.

1.2 Problems and Challenges

Researchers have proposed many approaches to specifying and analyzing temporal properties in the context of UML models; however, (as presented in the next chapter, the literature review) evidence indicates that UML designers face some challenges that hinder the practical use of these approaches. As of 2018, a complete "native" UML-based framework, techniques, and tools have not been developed to overcome these difficulties. In this work, a "native" framework is defined as a UML-based framework that exclusively uses UML notations and tools. The primary aim of the research described in this dissertation, is to investigate whether it is feasible to develop such a framework.

Most of the state-of-the-art UML-based approaches, to specifying and analyzing temporal properties, use *model checking* (see Figure 1.1). *Model checking* is the prominent paradigm-independent technique for analyzing temporal properties [1, 26]. A *model checking* technique determines whether a behavioral model of a system satisfies a temporal property. To recognize the significance of the *model checking*, ACM granted its inventors, Edmund Clarke, Allen Emerson, and Joseph Sifakis, the 2007 ACM Turing Award for "their role in developing Model-Checking into a highly effective verification technology that is widely adopted in the hardware and software industries." [27]. Further, ACM awarded Gerard Holzmann the 2001 ACM Software System Award "For SPIN, a highly successful and widely used software model-checking system . . . It has made advanced theoretical verification methods applicable to large and highly complex software systems." [28].

Figure 1.1 depicts the *model checking* process consisting of three tasks: modeling, specification, and verification. A full *model checking* framework consists of the following components, corresponding to the three tasks:

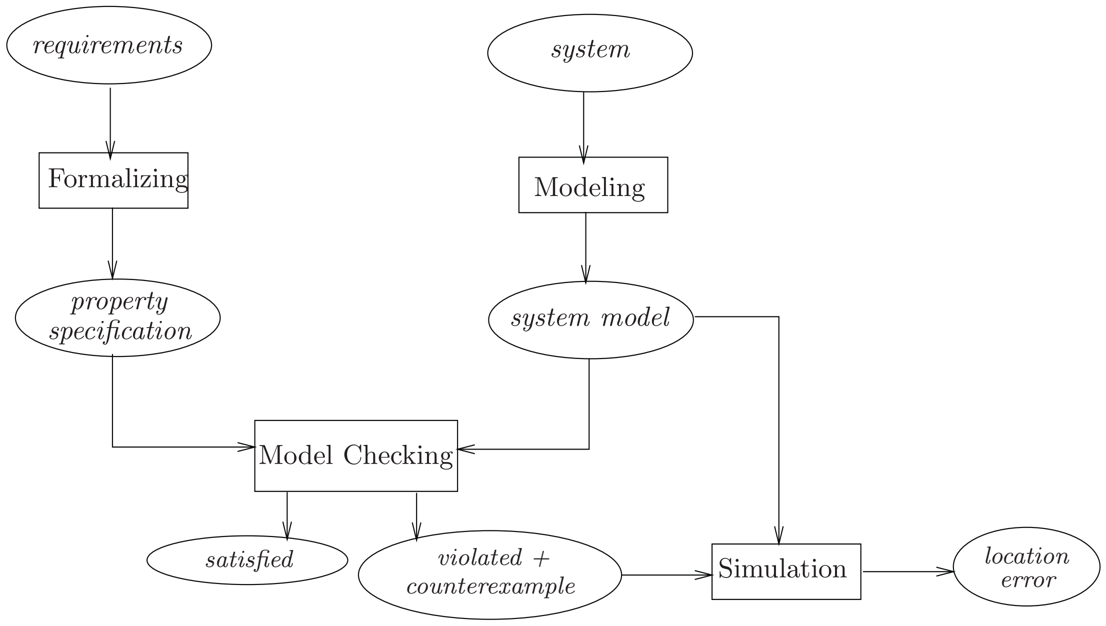


Figure 1.1: *Model checking* process, taken from the Principles of Model Checking book [1].

1. A modeling language for describing the behavior of a system.

2. A specification language for formulating the property requirements.

3. An analysis method and tool that check if the model satisfies the property specification.

Generally, designers describe the behavior of a system as a model and specify a property requirement in a manual manner. *Model checking* tools are then used to conduct the analysis automatically. For example, in SPIN [29] framework, developers use PROMELLA language to model the system behavior, LTL language [30] to specify temporal properties, and a toolkit that implements the *model checking* algorithms.

Instead of developing a "native" UML-based *model checking* framework, the majority of the state-of-the-art approaches integrate UML with existing *model checking* technologies. Their goals is to utilize the existing powerful *model checking* techniques and tools. The state-of-the-art methods use UML as the modeling language (the Modeling step in Figure 1.1). However, the methods rely on non-UML based techniques and tools to formalize and analyze properties, (the Formalizing and Model Checking steps in Figure 1.1). Members of the precise UML group initially proposed, and argued for, the integrated-methods approach, in which UML is integrated with other formal techniques, Evans et al. [31]. However, as will be explained shortly, this approach introduces difficulties by demanding UML designers to have specialized skills and knowledge of multiple non-UML techniques and tools. Examples of such approaches include: [32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44].

Chapter 2 provides evidence indicating those approaches are incompatible with UML designers and introduce some challenges in effectiveness and efficiency. On the theoretical side, the approaches can sometimes miss design faults, which makes them ineffective. On the practical side, they present problems with usability and introduce other accidental complexities, which makes them inefficient.

A core cause of this incompatibility is the fundamental differences between UML and the *model checking* languages in their design philosophy. In particular, UML was designed to be a general language to model different kinds of systems, while the individual *model checking* languages were created to specify and analyze temporal properties effectively. On the *model checking* side, the syntax and semantics of every construct are mathematically defined. This formality paves the way for developers to build a toolset for the languages. On the UML side, however, many UML constructs are either left undefined or are not precisely defined. These constructs are referred to as the UML semantic variation points. For example, the UML standard declares three variation points for the UML state machines: time management (synchronous vs. asynchronous), the event selection policy, and the transition selection policy [45]. These three semantics points are vital when specifying and analyzing temporal properties. While they are precisely defined by the *model checking* languages, they are left undefined in the UML standard. Bernhard Rumpe and Robert France argue that the variability in UML language and semantics introduces substantial difficulties when building tool support for semantically related tasks, e.g., analyzing temporal properties of a system [46].

Additionally, UML is a graphical language whereas *model checking* languages are mostly textual. On one hand, since the invention of *model checking* more than thirty years ago, *model* *checking* scientists have utilized and built on the already existing theoretical foundation of textual programming languages. By doing so, scientists have developed both a theoretical foundation and systematic techniques in order to define the syntax, semantics, and the pragmatics of their *model checking* languages. However, there are important differences between graphical and textual languages that prevent a simple transfer of methods developed for the former to the latter. Consequently, in the field of graphical modeling languages such as UML, the MDE community needs significant theoretical and pragmatic developments to parallel *model checking* technologies. For example, there is a lack of unified theories on how to design graphical modeling languages, model synchronization, and model transformation. Bran Selic contends that the MDE community needs substantial advancements in these areas to utilize UML properly [16].

In an attempt to reconcile these differences and utilize the powerful *model checking* techniques and tools, the research trend focuses on the use of *model transformation*. That is, the state-of-the-art approaches transform UML models to the input languages of *model checking* tools. We believe that the use of model transformation in this context is a core cause of the challenges that hinder UML designers to use these methods.

First, the use of these approaches requires UML designers to develop in-depth working knowledge of non-UML-based *model checking* techniques to specify and analyze temporal properties. Many of these approaches rely on heavyweight techniques that require extensive time to learn and mathematical maturity [47]. To specify properties, the methods necessitate the use of mathematical temporal logic such as LTL and CTL. Edmund Clark and other *model checking* expert have recognized the difficulty in the specification step that can be challenging for most *model checking* practitioners [48, 49]. One can argue that if *model checking* practitioners (who may have better mathematical training than traditional MDE designers) find property specification difficult, UML practitioners may confront more problems than their *model checking* counterparts.

Second, in addition to the difficulties in using these heavyweight approaches, the analyses may yield misleading results because their reliability depends on whether the transformation is semantic-preserving. A transformation is semantic-preserving when the behavior of the model before the transformation can be simulated by the behavior of the model after the transformation. To illustrate, if a model does not allow a system to reach a deadlock, a semantic-preserving transformation produces a model that prevents deadlock as well. Showing that two models are semantically-equivalent is possible by a technique called bisimulation introduced by Robin Milner [50]. Despite that, to apply bisimulation between two models, they must have an identical semantic domain. This is not the case when the transformation takes place between UML and the *model checking* languages.

The lack of semantic-preserving transformation renders the state-of-the-art approaches ineffective. Intuitively, as with any translation between two languages (natural languages included), the analysis of the constructs in the target language may not uncover faults that exist in the original language constructs, unless the translation preserves the meaning. A translation that does not maintain the meaning can also introduce new meaning that did not exist in the original constructs. Similarly, if a UML model and the derived model (by model transformation) do not semantically bisimulate one another, then errors can occur. To rely upon the analysis results, someone needs to show that transformation between the two languages is semantic-preserving. Researchers have shown that obtaining proof that a transformation is correct is not possible [51, 52].

Lastly, to examine the analysis results from the *model checking* tool, the results must then be presented to designers in UML notations. The going-back-to-UML process involves another transformation. Traceability mechanisms [53] are applied to transfer the analysis results back to UML. However, when two languages are designed for two different purposes, there is a lack of one-to-one mappings between the two language constructs. To illustrate, two closely related languages (UML and Alloy) lack one-to-one mappings between their constructs. For example, the UML class attributes and class associations are both mapped to Alloy signature fields (see Anastasakis et al. [54, 55]). The lack of one-to-one mappings (between UML constructs and Alloy constructs) results in a challenging model transformation between the two languages [54]. In the context of the state-of-the-art approaches, the UML and *model checking* languages are designed for different purposes. The lack of one-to-one mappings from UML constructs to the target language constructs results in the same challenges that we face when first transforming the models from UML to the languages of the *model checking* tool used for analysis.

Instead of following the integrated methods approach, we investigate developing a "native" lightweight UML framework consisting of the three components of a complete *model checking* framework. The framework is defined as lightweight because of two aspects [47, 56]. First, the framework does not rely on mathematical notations and only uses UML notations. The languages used for modeling a system, and specifying its temporal properties, should be UML-based combined with the tools used for the analysis. Second, the framework searches in constrained system executions, unlike *model checking* that exhaustively explores all possible executions [47, 57, 58].

1.3 Aim and Objectives

This section states the aim and objectives of the dissertation. The aim is the overall purpose of the dissertation, and the objectives are precise tasks that must be completed to achieve the aim.

- **Aim:** To develop a UML-based framework consisting of techniques and tools to formally specify and analyze temporal properties of software designed using UML.
- **Objectives:**
  1. To explore the state-of-the-art techniques and tools in order to identify research gaps and challenges in the field of *model checking* UML models.
  2. To develop a UML-based analysis technique that exclusively uses UML notations and tools.
  3. To streamline the process of specifying temporal properties for UML designers by developing a specification technique that uses UML notations.
  4. To develop an optimization technique that reduces the time needed for analysis, allowing the analysis to be scaled to larger UML models.
  5. To provide a proof-of-concept tool by implementing the specification, analysis, and optimization techniques.
  6. To evaluate the framework through an actual software specification and analysis projects.

To achieve the first objective, we examined the state-of-the-art approaches to analyzing temporal properties of systems that are modeled using UML. We categorized the material into specification and analysis techniques and then we conducted a Systematic Literature Review (SLR) to define and obtain answers to specific questions that are related to the two types of methods. The answers to the SLR questions helped identify open problems and gaps in the state-of-the-art research.

The identification of the open problems and challenges was the primary results of the SLR study. Many of the state-of-the-art approaches and tools are inadequate, and they introduce accidental complexities that hinder their effectiveness and efficiency. We formulated the challenges as open research questions that, if properly addressed, may lead to improvements toward achieving the research objectives. We categorized the open research questions into the following groups: (1) specification and analysis integration, (2) model transformation, (3) property discovery and patterns, and (4) tooling.

The second objective is concerned with developing a lightweight analysis technique that exclusively uses UML notations and tools. The proposed analysis method uses an OCL extension for temporal properties, TOCL [59] and USE Model Validator [60], as the back-end analysis device. Moreover, our technique uses and builds on an algorithm for generating *Snapshot Transition Models* (STMs) from UML design class model. The algorithm was defined by Yu et al. [61], members of the Software Engineering group at Colorado State University. However, ordinary STMs, as described by Yu et al. [62], do not support the specification and analysis of temporal properties. Towards this end, we augmented the STMs generation process by defining the *Snapshot Traversal Query Operations*, and STM invariants. In particular, using OCL, we defined the following six operations: *getNext(), futureClosure(), getPost(), getPrevious(), pastClosure(), and getPre()*. These operations facilitate defining the TOCL properties in OCL expressions. Further, the original algorithm does not consider some UML constructs such as inheritance hierarchies between classes. We provided support for these constructs.

We incorporated two methods in the analysis techniques to facilitate the lightweight analysis approach. The first method is the small-scope hypothesis that has been suggested by Daniel Jackson at Massachusetts Institute of Technology (MIT). This concept has been implemented and used by the Alloy Analyzer [63]. The second method is the search-depth that researchers in the area of Bounded Model Checking have applied [64]. Finally, the analysis results produced by USE Model Validator is given in a UML object diagrams. When the analysis requires large scopes and search-depths, the results are hard to debug to uncover the design faults. As a first step to better debugging, we developed an algorithm that converts the output of the USE tool to a sequence diagram.

With regards to the third objective, improving the process of specifying temporal properties, we developed a specification technique that uses UML notations. In 2003, Ziemann and Gogolla defined the TOCL language at the University of Bremen in Germany [59]. However, they have not developed a tool that parses and analyzes TOCL expressions. They also have not provided a systematic method to specify temporal properties in TOCL. Dwyer et al. [49] have defined the Property Specification Patterns that aid the process. They have described their patterns in formal temporal logics such as LTL, CTL, and QRE. Despite that practitioners have used the specification patterns, OCL, and TOCL languages in different context, the patterns have not been expressed in TOCL or OCL. Towards streamlining this process, we carried out two tasks utilizing Dwyer's et al. patterns. First, we defined 40 TOCL specification patterns that UML designers can use when specifying properties on UML design class models. Second, for analysis purposes of TOCL expressions, we provided 40 OCL patterns that interpret TOCL on STMs that, in turn, can by analyzed by the Model Validator. Consequently, UML designers can use UML notations to specify properties and can then check them by UML tools.

The fourth objective aims to develop an optimization technique that reduces the time needed for analysis. The technique is based on an existing technique that slices a UML class model to scale the analysis to larger models. Wuliang Sun [65] developed the original slicing method. However, similar to the original STMs, the original version of slicing algorithm did not target the analysis of temporal properties. Specifically, the initial slicing algorithm takes standard UML class models augmented by OCL constraints and it produces slices of the class models that have the relevant model elements to the OCL properties. For our purpose, we needed a technique for UML class models and TOCL properties. We collaborated with Wuliang to add two steps to the original algorithm. First, before applying the slicing procedure, we needed to convert the design class model to an STM augmented with the snapshot traversal operations. Second, we needed to apply the interpretation rules to the TOCL expressions to obtain the OCL that I can provide to the slicing algorithm. Adding these two steps into the algorithm, we attained the technique that is suitable for optimizing temporal properties analysis.

The fifth objective is concerned with implementing the specification, analysis, and optimization techniques. Kayle Hoehn and Wuliang Sun have implemented the algorithm that generates STMs from UML design class models. We extended the implementation to add the necessary code for creating the Snapshot Traversal Query Operations required by the analysis technique. Moreover, we developed a partial implementation of the specification technique. Specifically, we implemented the only widely used patterns, namely the Response and the Universality patterns, as they cover 80% of properties used in the industry according to a study done by Dwyer et al. [66]. Finally, we modified the code of the slicing algorithm that was developed by Wuliang to provide the required support for the optimization technique. Specifically, we incorporated my version of the STMs' generation implementation as well as the partial implementation of the TOCL interpretation.

To achieve the sixth objective, we validated the framework by using the proof-of-concept tool to specify and analyze two real-world case studies. The first study produced a novel software model that is called the Generalized Spatio-Temporal Role-Based Access Control Model(GSTRBAC) Abdunabi et al. [67]. The development of GSTRBAC focused on addressing the many application requirements of wireless and mobile devices that make use of the spatiotemporal information of a user, to provide better functionality. Such applications necessitate authorization models where access to a resource depends on the credentials of the user and also on the location and time of access. In the context of the research discussed in this dissertation, we used the specification and the analysis techniques to formalize the model in UML and OCL notations. The second case study is based on the Steam Boiler Control System (SBCS) specification problem [68]. The SBCS specification problem has been used extensively to assess the effectiveness of many software specification and verification approaches. Using SBCS, therefore, provides a benchmark study that can be used to compare our framework with other current methods.

1.4 Dissertation Contributions

The overall contribution of this dissertation is a "native" UML framework for the specification and analysis of temporal properties. The framework consists of techniques and tools that seek to accomplish the objectives. The result of every objective adds significant knowledge to the field. Therefore, this knowledge is considered one of the main contributions of this dissertation. The contributions are listed below:

1. A state-of-the-art SLR on *model checking* UML models.
2. A lightweight analysis technique for analyzing temporal properties on UML class models.
3. A pattern-based UML property specification technique that uses the TOCL and OCL notations to specify temporal properties.
4. An optimization technique that is used to enhance the applicability of the analysis technique to larger models.
5. A proof-of-concept tool based on the the proposed techniques.
6. The Generalized Spatio-Temporal Role-Based Access Control Model(GSTRBAC).

1.5 List of Publications

The research activities done to complete this dissertation have yielded multiple publications.

1. Mustafa Al-Lail, Ramadan Abdunabi, Robert B. France, Indrakshi Ray: Rigorous Analysis of Temporal Access Control Properties in Mobile Systems. ICECCS 2013: 246-251

2. Mustafa Al-Lail, Ramadan Abdunabi, Robert B. France, Indrakshi Ray: An Approach to Analyzing Temporal Properties in UML Class Models. MoDeVVa@MoDELS 2013: 77-86

3. Mustafa Al-Lail: A Framework for Specifying and Analyzing Temporal Properties of UML Class Models. Demos/Posters/StudentResearch@MoDELS 2013: 112-117

4. Mustafa Al-Lail, Wuliang Sun, Robert B. France: Analyzing Behavioral Aspects of UML Design Class Models against Temporal Properties. QSIC 2014: 196-201

5. Ramadan Abdunabi, Mustafa Al-Lail, Indrakshi Ray, Robert B. France: Specification, Validation, and Enforcement of a Generalized Spatio-Temporal Role-Based Access Control Model. IEEE Systems Journal 7(3): 501-515 (2013)

1.6 Dissertation Structure

The rest of the dissertation is organized as follows. Chapter 2 presents the SLR study on *model checking* of UML models. The chapter also highlights and details the state-of-the-art challenges in the field. Chapter 3 describes the details of the proposed UML framework and how it mitigates some of these challenges in the field. The framework consists of the analysis, specification, and optimization techniques along with the proof-of-concept tool. Chapter 4 focuses on the evaluation of the proposed analysis, specification, and optimization techniques. In Chapter 5, we summarize the contributions and discuss limitations of the framework and outline some of the future research activities that can be done to enhance the framework.

Chapter 2

Systematic Literature Review

The focus of this chapter is achieving the first objective of the dissertation, which is "to explore the state-of-the-art approaches to identify research gaps and challenges in the field of model checking UML models." A substantial portion of this chapter is written to fulfill the requirement of my Ph.D. Research Examination, i.e., the qualifying exam.

Many temporal property analysis methods exist. Based on an agreement, and the advice of my late adviser Dr. France, the scope of this survey is limited to *model checking*. The reason is that *model checking* is the most prominent approach to specifying and verifying temporal properties. Further, it is the most successful and most used approach in both industry and academia. Surveying the UML-based model checking technique can be more beneficial than finding open problems in other methods that have not matured, been implemented, or used in the industry. Therefore, identifying challenges and open problems in UML-based model checking methods paves the way to developing better techniques that are applicable to industry.

We conducted a Systematic Literature Review (SLR) on model checking of UML models. A systematic review follows a clear method to identify, analyze, synthesize, evaluate, and compare literature works relevant to a particular research topic [69]. The SLR method allows researchers to collect and summarize existing evidence when answering specific questions. The answers help in identifying current limitations and open problems, hence, suggesting possible future research directions.

Our application of the SLR process revealed a vast number (i.e., order of tens) of research papers that meet the criteria to be included in this study. We included in this chapter the most significant studies based on quality metrics (i.e., the number of citations and the place of publication). We classified the papers into specification and analysis approaches, as these are the methods that fit the dissertation's aim and objectives. Specification approaches are those that focus on defining languages for specifying temporal properties in UML contexts for model checking analysis.

Analysis approaches are those that concentrate on the analysis of temporal properties using model checking techniques. Ideally, UML designers aspire for an “effective” and “efficient” MDE model checking framework. However, the results of this SLR study show that many challenges need to be addressed to develop such a framework.

The rest of the chapter is organized as follows: we present the evaluation criteria (the research questions defined by the SLR methodology) in Section 2.1. Section 2.2 surveys and evaluates the specification approaches. The analysis approaches are studied and assessed in Section 2.3. In Section 2.4, we discuss several challenges that are expressed as open research questions. Section 2.5 summarizes and concludes the chapter.

2.1 Literature Review and Evaluation Methodology

This section presents the details of the SLR process. SLR consists of defining the following: (1) the research questions to be answered, (2) the search process by which the studies considered for the survey are identified, and (3) the inclusion and exclusion criteria. The list of the included studies is given at the end of the section along with a discussion of the threats to the validity of the study.

2.1.1 Research Questions

Specifying the research questions is the most important part of any systematic review as they drive the entire systematic review methodology. The questions are based on the criteria defined in Table 2.1. These criteria capture our understanding of the strengths, challenges, and what needs to be done in applying model checking.

Researchers have shown that many of model checking tools are very useful in uncovering faults in real systems such as IEEE Futurebus+ standard [70]. On the other hand, many UML-based model checking methods apply their approaches in toy examples to demonstrate that they can find design faults. The effectiveness criterion in Table 2.1 checks if the method has been used to find design faults in real-world systems.

Table 2.1: Criteria used to define the research questions

Criteria	**Description**
Effectiveness	Has the approach been applied to uncover design faults in real systems?
Scalability	Does the approach use an optimization technique to cope with the state explosion problem?
Tool	Is there tool support?
Industrial adoption	Has the approach been used in industrial projects?
Usability	Does the approach use mathematical notations and tools that require a steep learning curve to use?
Formality	Are the approach's languages formally defined?
Pattern support	Are temporal specification patterns defined?

The scalability criterion in Table 2.1 examines if a proposed approach applies any optimization technique to cope with the state explosion problem. The main challenge of model checking is the state explosion problem [48]. That is, it is difficult to check real-life complex systems in which the state space is usually too large to be efficiently verified. To counter this problem, modern model checkers apply optimization techniques such as: (a) symbolic representation of the state space by partial order reduction or binary decision diagrams, (b) on-the-fly verification, or (c) abstraction techniques.

In addition to the optimization techniques, another strength of model checking is that many tools exist that are applicable to the analysis of many areas such as: hardware, software, and communication protocols. The tool criterion in Table 2.1 identifies if the approach is supported by a tool-set.

The industrial adoption criterion in Table 2.1 asks if the approach has been used in industrial projects. One reason of the widespread use of models checking is that it has been successfully used to uncover design faults in many industrial projects such as at IBM, Intel, and NASA.

Model checking techniques require mathematical maturity and steep learning curves. These characteristics affect their usability, particularly for the UML designers who might not have strong mathematical training. The usability criterion in Table 2.1 assesses if developers need to use mathematical notations and tools.

The formality criterion in Table 2.1 checks if the syntax and semantics of the languages used in the approach are formally defined. For example, the formal description of the syntax and semantics of CTL and LTL temporal properties plays a vital role in developing tools for model checking.

Specification of temporal properties in LTL and CTL is another main challenge that faces the application of model checking. Studies have shown that only certain types of properties are relevant in practice and are represented as property specification patterns. The pattern support criterion in Table 2.1 assesses if a method is used to provide property patterns that assist the specification process.

Specification Approach Research Questions

A specification approach aims to define a language that enables UML designers to specify temporal properties of their UML models. Recall from Chapter 1 that in model checking terminology, the specification component refers to the process of describing temporal properties in a formal language, and the modeling component relates to defining the system behavior in a modeling language. Though specification languages, such as Z, can be used to specify a model and properties of a system. In this dissertation, a “specification language” refers to temporal logic specification while a “modeling language” specifies the behavioral models.

Table 2.2 lists the research questions that we need to answer for a particular specification approach. The table also shows the related evaluation criteria for every question. Because scalability criterion (as defined in Table 2.1) is concerned with alleviating the state explosion problem with an optimization technique, none of the questions in Table 2.2 impact scalability. However, it is worth investigating how we can define temporal property patterns that not only ease the specification but also help in developing better optimization technique to cope with the state explosion problem. This investigation is out of the scope of this research. The questions are ordered based on their significance.

Many modeling and specification languages developers use English to define the syntax and semantics (e.g., UML initially was defined this way). This way of describing languages leads to ambiguity and difficulties to develop tools for these languages. In particular, without a language’s

Table 2.2: Specification approach research questions

ID	Question	Related Criteria
SRQ1	Is the language syntax formally defined?	Tool, Usability, Formality
SRQ2	Are the language semantics formally defined?	Effectiveness, Tool, Formality
SRQ3	Does the language have tool support?	Effectiveness, Tool, Industrial adoption
SRQ4	Are temporal property patterns defined?	Usability, Industrial adoption, Pattern support

formal syntax and semantics, developers rely on assumptions to create CASE and IDE tools that parse and perform semantically related activities such as analysis. Further, because of many ways to understand English, developers may create different implementations of the same language, this leads to interoperability problems between different tools.

SRQ1 and SRQ2 in Table 2.2, check if the language developers have precisely defined the syntax and semantics. Whether the language syntax uses UML notations or not is important for the UML designers who are familiar with these notations. The types of properties that can be analyzed depend on the semantics used to define the language. Usually, temporal property specification language semantics are formally defined either using linear-time logic or branching-time logic, also called computation tree logic. Lamport studied the differences between the two formalisms and concluded that some properties could be specified by each logic but not the other, and vice-versa [71].

Additionally, MDE aims to create models that are useful for communication as well as for analysis. This goal effects both the syntax and the semantics of a MDE language; graphical syntax is more communicable than textual syntax and as precise semantics is necessary for effective analysis approaches. The primary concern in the design of temporal logics language has been that they must be precisely interpreted and analyzed. So far, I have not seen any temporal specification language that is both analyzable and communicable.

In the context of automated software analysis, formally defining the syntax and semantics is not enough; designers need tools to analyze their designs automatically. SRQ3 in Table 2.2 asks if a specification approach provides tool support. In other words, without a tool, a formally specified language may provide little benefit when analyzing complex software.

SRQ4 in Table 2.2 asks whether property specification patterns are defined for the language. As discussed earlier, one of the main challenges in applying model checking is the difficulty of specifying temporal properties; therefore, it is important to alleviate this problem [48, 49]. One way is through better education, and another way is through the use of patterns that facilitate the specification.

Analysis Approach Research Questions

An analysis approach provides techniques for ensuring that temporal properties of UML models are satisfied. Table 2.3 lists the research questions that are defined to evaluate the analysis methods. The questions are ordered based on their significance, which is discussed next.

Concerning VRQ1 in Table 2.3, whether an analysis approach involves model transformation (from UML to model checking languages), or not, is significant. As discussed in Chapter 1, model transformations introduce a lot of accidental complexities. VRQ1 is broken into sub-questions, each of which is concerned with a particular difficulty that the transformation introduces. VRQ1a checks if the transformation has been shown to be correct. For exogenous transformation, proving the correctness constitutes a major challenge in MDE (see Baudry et al. [51, 52]). Many model transformation rules are defined verbally in English, but no tool is developed to do the transformation. Automation is one of the objectives and means of MDE; whether the transformation is automatic or not is checked using VRQ1b. VRQ1c checks whether the produced counterexample is transferred back to UML to be examined. The ability to debug counterexamples in UML notations is a major factor because UML designers may not have high knowledge in the target model checking languages to debug counterexamples. Further, other specialized skills, such as using specific optimization techniques, that are not UML related, might be needed. This is checked using VRQ1d.

VRQ2 in Table 2.3 asks whether the property to check is specified in UML notations. This is important because UML designers should not be forced to learn notations they are not accustomed to using.

Table 2.3: Analysis Approach Research questions

ID	**Question**	**Related Criteria**
VRQ1	Does the approach transform to a model checking?	Effectiveness
a)	Is the model transformation proven?	Effectiveness, Formality
b)	Is the transformation automatic?	Tool, Industrial adoption, Usability
c)	Are analysis results in UML term?	Effectiveness, Usability
d)	Are non-UML specialized skills needed?	Usability
VRQ2	Are properties specified in UML notations?	Usability, Industrial adoption
VRQ3	Is any optimization technique used?	Effectiveness, Scalability
VRQ4	Is the approach heavyweight or lightweight?	Usability, Industrial adoption
VRQ5	Has the approach been used in an industrial project?	Industrial adoption

The state explosion problem is a major challenge that all model checkers suffer [48]. The problem is more severe in model checking software than hardware because of the sheer number of different software components and their data types. VRQ3 in Table 2.3 asks whether the proposed analysis approach provides a technique to address this problem.

VRQ4 asks if the approach is lightweight, in the sense that it uses UML notations and tools, and fits the working habits of the UML designers. Consequently, lightweight approaches are usable to UML designers as they fit their working style and therefore minimize accidental complexity, while heavyweight methods demand specific working environments to use [47].

As discussed above, one of the main strengths of *model checking* is that it has been shown to be successful in uncovering faults within real complex systems. VRQ5 addresses this concern.

Table 2.4: Sources that are used to search for studies

Electronic	
Source	**Web address**
ACM Digital Library	Portal.acm.org
Springer Link	www.springerlink.com
Google Scholar	scholar.google.com
Manual	
Journal of Software and Systems Modeling (SoSyM)	
IEEE Transactions on Software Engineering (TSE)	
ACM Transactions on Software Engineering and Methodology (TOSEM)	
MODELS Conference Series	
Manual search through the references of identified primary studies	

2.1.2 Search Process

The search process defines how the search for primary studies to be reviewed is undertaken. It specifies: (1) the studies' sources that will be searched, (2) the study string that will be used when searching electronic resources, and (3) the inclusion/exclusion criteria. The next subsections elaborate on these steps.

Studies Sources

Table 2.4 provides the list of the resources that were used in this study. Two search methods were used to obtain studies: (1) electronic, this includes searching different digital libraries and databases using the search terms, and (2) manual, this includes searching major journals and conferences in the field.

The Search String

A general approach to identifying a valid search string is to break down the structured search questions into individual facets (i.e., population, intervention, comparison, outcome, context, and study designs) [69]. Only two facets are relevant to this SLR study: (1) the population, which comprises the concepts, technologies, or languages that are being treated, and (2) the intervention, referring to techniques to do specific tasks (such as testing, analysis, or requirement specification). In this study, population terms are related to UML language and the OCL, and intervention terms are related to model checking.

Table 2.5: Search string terms

Population Terms
UML, OCL, Unified Modeling Language, Object Constraint language, patterns, Temporal properties
Intervention Terms
Model checking, verification, validation, analysis, extension, extended, transformation, checking

Table 2.5 shows the population and intervention terms. After defining these terms, the search string was constructed by combining words in each group with the OR operator, then combining the two groups by the AND operator.

2.1.3 Study Inclusion/Exclusion Criteria

Study inclusion/exclusion criteria are intended to identify those primary studies that provide direct evidence about the research questions (Tables 2.2 and 2.3). Because the research described in this dissertation focuses on developing UML-based model checking techniques, algorithms, and tools, only approaches that aim to address this goal were included in the SLR.

UML and model checking have been integrated to model and verify real-time reactive systems and probabilistic systems. In checking a real-time reactive system, explicit time constraints (e.g., a system responds in 5 milliseconds to a particular signal) are tested to ascertain that a system satisfies them. Probabilistic systems are those in which the analysis guarantees that the system will not fail in an x percentage of the time (e.g., 90% of the time). However, these types of systems are out of the scope of this SLR study since the research described in this dissertation is aimed at developing a framework for relative-time systems exclusively. Real-time and probabilistic model checking approaches are therefore irrelevant.

The number of obtained studies that could be included in this study was sizable. The scope of the study was limited to the approaches listed in Table 2.6. The table includes the five initial papers selected by the research examination committee members, and selected ones obtained by the SLR process, based on the following metrics: (a) number of citations, and (b) the quality of conference or journal in which the approach was presented (i.e., top-tier conferences and journals Software Engineering)

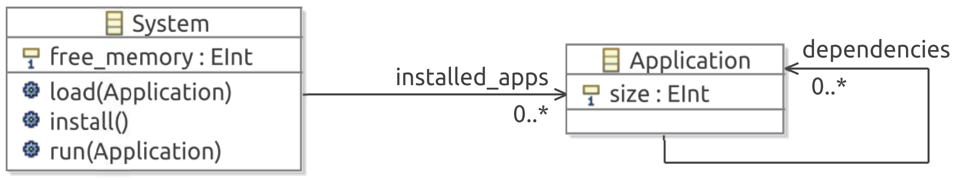
The next two sections survey the specification and the analysis approaches and then evaluate them concerning the research questions.

Table 2.6: Included studies

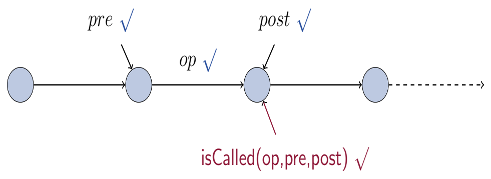
Specification Approaches	
Approach	**Reference**
Kanso and Taha(2012) Section 2.2.1	Bilal Kanso and Safouan Taha. Temporal Constraint Support for OCL. In SLE, pages 83-103, 2012.
Ziemann and Gogolla(2003) Section 2.2.1	Paul Ziemann, Martin Gogolla: OCL Extended with Temporal Logic. Ershov Memorial Conference 2003.
Flake and Muller(2003) Section 2.2.1	Stephan Flake and Wolfgang Muller. Formal Semantics of Static and Temporal State-Oriented OCL Constraints. Software and System Modeling, 2(3):164-186, 2003.
Soden and Eichler(2009) Section 2.2.1	Michael Soden, Haio Eichler: Temporal Extensions of OCL Revisited. ECMDA-A 2009
Brandfield et al.(2002) Section 2.2.1	Julian C. Bradfield,Juliana Kuster Filipe, Perdita Stevens: Enriching OCL Using Observational Mu-Calculus. FASE 2002
Analysis Approaches	
Baresi et al. (2013) Section 2.3.1	Luciano Baresi, Gundula Blohm, Dimitrios S. Kolovos, Nicholas Matragkas, Alfredo Motta, Richard F. Paige, Alek Radjenovic, and Matteo Rossi. Formal verification and validation of embedded systems: the uml-based mades approach. Software Systems Modeling, pages 1-21, 2013.
Moffett et al. (2013) Section 2.3.1	Yann Moffett, JÃijrgen Dingel, Alain Beaulieu: Verifying Protocol Conformance Using Software Model Checking for the Model-Driven Development of Embedded Systems. IEEE Trans. Software Eng.39(9): 1307-1325 (2013)
Eshuis (2006) Section 2.3.1	Rik Eshuis:Symbolic model checking of UML activity diagrams. ACM Trans. Softw. Eng. Methodol. (TOSEM) 15(1):1-38 (2006)
Zurowska and Dingel (2013) Section 2.3.1	Karolina Zurowska, Juergen Dingel: Model Checking of UML-RT Models Using Lazy Composition. MoDELS 2013: 304-319
Lilius and Paltor(1999) Section 2.3.1	Johan Lilius, Ivan Paltor: Formalising UML State Machines for Model Checking.UML1999: 430-445
Zalila et al. (2013) Section 2.3.1	Faiez Zalila, Xavier CrÃl'gut, Marc Pantel: Formal Verification Integration Approach for DSML. MoDELS 2013: 336-351

2.2 Specification Approaches

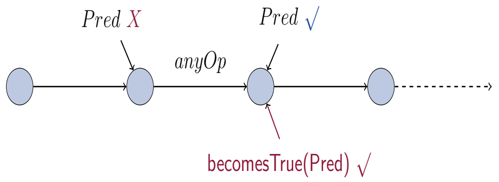
In Section 2.2.1, a summary and an example of how to specify a temporal constraint are presented for five approaches. Section 2.2.2 evaluates and discusses the approaches based on the specification research questions that were defined in Table 2.2.



(a) Class model



(b) isCalled event



(c) becomesTrue event

Figure 2.1: Kanso's specification language examples [72]

2.2.1 Survey of the Specification Approaches

Kanso and Taha (2012)

Kanso and Taha [72] propose a language that extends OCL with events and property specification patterns [49]. An example explains the use of the approach. Figure 2.1 shows how temporal properties can be specified using the approach. This approach specifies temporal properties on class models. The class model in Figure 2.1 represents the structure of a simple software system. The system can be used to load, install, or run an application. One of the temporal requirements is that an application can be loaded one time, at most. This is a safety property and it is specified formally as:

```
context System
temp Safety: let apptoInstall : Application in
     eventually isCalled(load(app:Application), pre: app=apptoInstall)
                at most 1 time
     globally
```

The expression uses *isCalled(load(app:Application), pre=apptoInstall)* event. As shown in Fig 2.1(b) this event has the name of the operation to be called, the pre- and the postcondition.

This event is a generic construct that merges the operation (call/start/end) events with state change events that exist in the standard OCL. This temporal property is an example of the response pattern in the global scope that will be described in Chapter 3, along with other patterns and scopes. The response pattern with global scope describes cause-effect relationships between a pair of events in the entire system execution. The important point to note is that this approach uses patterns to specify temporal properties.

The other event is *becomesTrue(Pred)*, which is shown in Figure 2.1(c). To illustrate this event, consider the following liveness property: each loaded application needs to be eventually installed after it has been loaded. This requirement is expressed as:

```
context System
temp Liveness: let apptoInstall : Application in
               becomesTrue(self.installed_app->includes(apptoInstall))
     following isCalled(load(app:Application), pre:app=apptoInstall))
     globally
```

In the expression above, the *becomesTrue* clause indicates that the predicate states the application to install (i.e., apptoInstall) is included in the installed applications after a call to load the application is invoked on the same application.

Ziemann and Gogolla (2003)

Ziemann and Gogolla [59] presented an extension of OCL with elements of linear temporal logic. Past and future temporal operators are introduced. The extension is called Temporal OCL (TOCL), and it allows the specification of temporal properties on UML class model. The introduced temporal operators are integrated with the common OCL syntax. The researchers also define the semantics of UML object models that represent the structural aspects of the system. The semantics of TOCL operators are defined using the definition of this object model. They adopted the definition of class models and augmented it with a description of state sequences.

An object model is a tuple M=(CLASS, ATTc, OPc, ASSOC, associates, roles, multiplicities,) that consists of a set of classes (CLASS) with each class c having attributes (ATTc) and operation

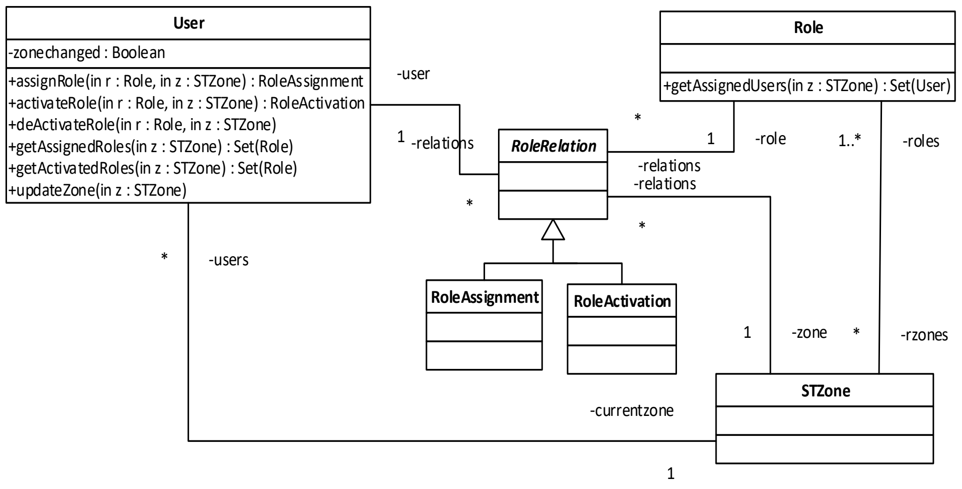
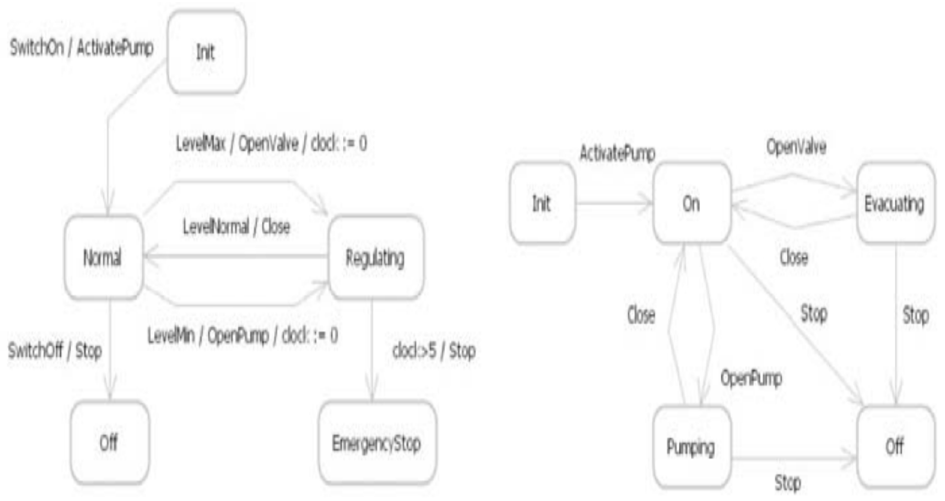

User
-zonechanged : Boolean
+assignRole(in r : Role, in z : STZone) : RoleAssignment
+activateRole(in r : Role, in z : STZone) : RoleActivation
+deActivateRole(in r : Role, in z : STZone)
+getAssignedRoles(in z : STZone) : Set(Role)
+getActivatedRoles(in z : STZone) : Set(Role)
+updateZone(in z : STZone)
Role
+getAssignedUsers(in z : STZone) : Set(User)
RoleRelation
RoleAssignment
RoleActivation
STZone
-user
-relations
-role
-roles
-users
-zone
-rzones
-currentzone
1..*

Figure 2.2: Partial GSTRBAC model, taken from Al-Lail et al. [73]

(OPc) assigned to it. Associations (ASSOC) connect classes with each other. The function "associates" maps each association name to a finite list of participating classes. The function "roles" assigns a role name to each participating class. Objects of associated classes can be connected with links. The number of links an object can be associated with is specified by the function 'multiplicities.' A system state, also called snapshot, is defined as a tuple $\sigma(M) = (\sigma Class, \sigma ATT, \sigma Assoc)$ where $\sigma Class$ is a set of current objects and their current attribute values is σATT and $\sigma Assoc$ are the links connecting the objects. An infinite state sequences for a model M is denoted as $\sigma^{\wedge}(M) = \langle \sigma 0, \sigma 1, ... \rangle$. The semantics of the introduced temporal operators are precisely defined over state sequences.

As an example, consider the following GSTRBAC persistence property defined on the model in Figure 2.2: "When a user changes his spatiotemporal zone in one state, then all the activated roles must be deactivated in the next state." The TOCL property given below formally specifies this property.

```
context User inv:
self.zonechanged=true implies next
self.getActivatedRoles(currentzone)->isEmpty()
```



(a) Controller (b) Boiler

Figure 2.3: The steam-boiler control system as two state machines [75]

Flake and Mueller (2002)

Flake and Mueller [74] presented a UML profile extension to OCL, called temporal OCL. The profile is based on the OCL metamodel and it uses Clocked CTL to define semantics. The authors formally defined the notion of *Statechart configuration* that is a description that captures the set of currently activated states of a UML statechart. They also integrated this notion into the formal object model, as defined by Ziemann and Gogolla 2003 [59], and extended it with overall system state. Based on this formal definition, the authors defined formal semantics of a *trace* over a given model. In this formal model, a trace is an infinite sequence of system states. Temporal constraint semantics are defined based on the notion of traces. Finally, a formal definition is provided for the operation *oclInState(s:OclState)*, which was only defined syntactically in the OCL standard.

Soden and Eichler (2009)

Soden and Eichler [75] proposed a language, called Linear Temporal OCL (LT-OCL), that extends OCL with linear temporal operators such as next, until, always, and eventually. The concrete syntax of LT-OCL is defined by extending the attributed EBNF grammar that is mapped to the abstract syntax defined as a MOF metamodel. The semantics is defined with M3Action language over this metamodel [76]. M3Action language describes behavior with a graphical syntax similar

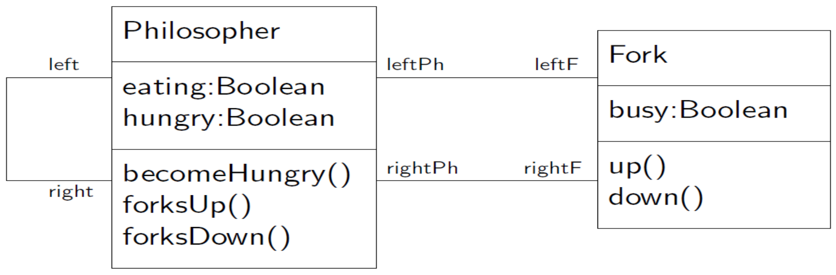


Figure 2.4: Class diagram for dining philosophers [77]

to UML Activities/Action, but with a precise update semantics defined for instances of MOF metamodels. Actions that specify the operational behavior are defined in the extended metamodel as specific MOperations. For example, elementary actions include OCL query operations to navigate the model, an assign action to update class properties, and a create actions to instantiate a class.

To illustrate the approach, consider steam boiler control system [68]. The UML state machines of the control program and the boiler components are depicted in Figure 2.3. The role of the controller is to regulate the water supply to the boiler by switching on and off the pump as well as the valve. The following LT-OCL expression specifies the requirement that if the water level is below the minimum limit, the pump will eventually be started.

```
always(Event.allInstances()->forAll( value='LevelMin' implies
 eventually(pump.activeState='Pumping')))
```

Brandfield et al.(2002)

Bradfield et al. [77] proposed the use of the observational mu-calculus to define the semantics of their OCL temporal extension language $\theta\mu(OCL)$. Using $\theta\mu(OCL)$ requires an understanding of temporal logic with fixed points, which might be hard for UML designers to learn. To alleviate this problem, the authors provided some useful design templates of standard usage. The templates use OCL notations and they are given semantics by translating them into $\theta\mu(OCL)$.

To illustrate the approach consider the class model of the dining philosophers depicted in Figure 2.4. One of the liveness properties of the model is "whenever a philosopher is hungry, he/she will be able to eat eventually.” To formally specify this requirement, a UML modeler can use the provided template named *after/eventually*. This template takes the following form:

```
context Classifier:
   after: oclExpression
   eventually: oclExpression
```

Like an invariant or a pre/post condition, this template is written in the context of a classifier. The after clause expresses some trigger for the contract. Once the condition in the after class becomes true, the contract specifies that the condition expressed in the eventually clause will eventually become true. This template can be instantiated to specify the liveness property above as follows:

```
context Philosopher:
   after: self.hungry=true
   eventually: self.eating=true
```

The next section evaluates these methods.

2.2.2 Evaluation and Discussion of the Specification Approaches

Table 2.7 shows the evaluation results of the five included specification approaches that are summarized in previous section 2.2.1. The results show that all the approaches do provide an object-oriented syntax to their languages. One can see that all of these approaches provide a textual language that extends the OCL standard. None of them provide graphical notations for defining temporal properties. Although these approaches provide suitable notations to enable UML designers to write textual temporal constraints that can be analyzed, graphical notations might be helpful to communicate the properties of a system among different stakeholders.

Semantic-wise, the five approaches formally define precise semantics for their language. The results also show that only two approaches, i.e., Kanso and Taha (2012) [72] and Bradfield et al. (2002) [77], provide tool support that performs analysis tasks. Kanso and Taha implemented a

Table 2.7: Evaluation of specification approaches

Approach	SRQ1:Syntax	SRQ2:Semantics	SRQ3:Tool	SRQ4:Pattern
Kanso and Taha(2012) Section 2.2.1	OCL+ events	Yes, traces	parser	Yes
Ziemann and Gogolla(2003) Section 2.2.1	OCL+ temporal operators	Yes, traces	No	No
Flake and Muller(2003) Section 2.2.1	OCL+CTL	yes, Clocked CTL	No	No
Soden and Eichler(2009) Section 2.2.1	OCL+ temporal operators	Yes, traces	No	No
Brandfield et al.(2002) Section 2.2.1	OCL+template clauses	*mu*-calulus	Yes	Yes, templates

parser for their proposed language while Bradfield et al. (2002) developed a more advanced tool that is capable of doing model checking analysis tasks. Other approaches lack the tool support.

Concerning pattern support, only Bradfield et al. (2002) [77] and Kanso and Taha (2012) [72] define techniques to ease the specification of temporal properties. Kanso and Taha described their language based on Dwyer's patterns. However, the language only supports the specification of temporal properties that are instances of Dwyer's patterns, i.e., if a property cannot be specified using any of the patterns, then it cannot be specified by the proposed language. The reason for this shortcoming is that patterns are limited in their expressiveness power, which has been shown by their designers [78]. Consequently, designers cannot specify certain properties using the Kanso and Taha language. Bradfield et al. (2002) do not strictly provide support for the Dwyer's patterns, but their temporal specification templates are easy to use and may be as expressive as Dwyer's patterns. For example, the *after/eventually* template provides the cause/effect relationship among different actions/states in a system. This relationship is defined as the response pattern by Dwyer et al. [49]. Using Dwyer's pattern, this relationship can even be restricted to a portion of the system execution (scope) in which it should hold, but this feature is not supported by the templates provided by Bradfield et al [77].

Among the surveyed specification approaches, the Bradfield et al. (2002) approach stands out. The method defines the language formally, provides tool support, and augments the language with templates that ease the specification of temporal properties. One shortcoming of this approach is that it transforms the templates to observational *mu*-calculus and no proof of correctness for this transformation is provided, as it is shown to be a difficult task [52].

2.3 Analysis Approaches

The previous section focused on surveying and evaluating selected specification approaches identified by the SLR. This section focuses on surveying and evaluating selected SLR-identified analysis approaches. Recall that the specification approaches focus on defining languages for specifying temporal properties in UML contexts for model checking analysis. Analysis approaches concentrate on the analysis of temporal properties using model checking techniques. Though some analysis approaches provide ways to specify properties, they focus on the analysis not on the specification.

This section is organized as follows: Section 2.3.1 provides summaries of the included analysis approaches listed in Table 2.6 and Section 2.3.2 evaluates them based on the analysis questions defined in Table 2.3.

2.3.1 Survey of the analysis Approaches

Baresi et al. (2013)

The designers of this approach provided a tool-chain that can be used for analysis and closed-loop simulation. They use the MADES UML notation to model a system. MADES integrates a subset of UML notations and elements from MARTE (the UML profile for modeling and Analysis of Real time and Embedded systems). To specify temporal properties, the authors suggest the use of TRIO language, a first-order linear temporal logic that supports a metric on time.

The approach proposed to combine several existing and mature technologies to perform the analysis and the closed-loop simulation. During the analysis, MADES models are transformed to a set of formulae expressed in the TRIO language. Then analysis uses a model checker Zot to perform the analysis task. The analysis results are then returned to be presented to the designers in MADES terms. The implementation employs a traceability tool to translate the results to MADES notations. The closed-loop simulation aims to provide a trace of the system that both satisfy a model of the system and a model of its environment.

Moffett et al. (2013)

This paper proposes an analysis approach that finds property violations in UML component models. A UML component model consists of some components that exchange messages through their interfaces. A behavior of a structured class can be described using a Behavioral State Machine (BSM). Each interface of a component has a Protocol State Machine (PSM) that is used to describe allowed message exchanges between two components. A message arriving at an interface may trigger a transition in the BSM of the class owning the interface. If the BSM is not ready to receive a message arriving at one of its interfaces, the message is dropped. This paper proposes an approach to make sure that the behavior of a component, defined by its BSM, conforms to the behavior defined by its interfaces, and all its PSMs. UML-RT is used to model a system as a component diagram which includes BSMs of components and their corresponding PSMs.

Three types of property violations can occur: (1) input safety violation, (2) output safety violation, and (3) progress violation. An input safety violation occurs when a message is received at one of the interfaces (ports) (PSM), but the behavior of the component (BSM) is not ready to receive such a message. The output safety violation happens when a component tries to send a message to another component through an interface, but the state of the interface does not allow it to send a message. A progress violation occurs when the behavior of the component does not force the component to make progress (e.g., send a message through a particular interface). In other words, the behavior of the component allows an infinite loop in which the sending of the message is not part of the loop.

To perform a conformance check against these violations, designers need to do three steps. First, the behavior of the components and their ports must be formally defined and composed into an automaton called a Verification Finite State Automaton (VFSA). The conformance check is based on the exhaustive exploration of the state space of the VFSA. Second, the different types of violations must be formally specified as LTL properties. The conformance check is then defined as follows: a component behavior is said to conform to the behavior of its ports if and only if the verification automaton (VFSA) does not violate any of the three LTL properties representing the three types of violations. Lastly, to automatically perform the check, the VFSA and the LTL properties are transformed to the input language of the Java PathFinder (JPF) model checker. If a violation exists, a UML designer needs to debug the counterexample produced by JPF.

Eshuis (2006)

Eshuis [37] proposed an approach to verify UML activity diagrams using the NuSMV model checker. The objective of the analysis process is to check the data integrity constraints of UML activity and class diagrams by specifying temporal properties. For example, data cannot be referenced before it is created or after it is deleted. Eshuis developed two tools to transform the activity diagrams to the input language of NuSMV model checker, which in turn performs a symbolic model checking using Binary Decision Diagrams to reduce the search space. Furthermore, this approach uses propositional linear temporal logic with both past and future operators (PLTL) to specify the temporal requirements to be analyzed.

Eshuis applied two types of transformations that he calls translations: requirement-level translations and implementation-level translations. The first transformation is based on Harel's statecharts semantics and can be efficiently verified, assuming the perfect synchrony hypothesis, i.e., the statechart responds immediately to input events from its environment. Though these machines can be efficiently verified, they are unrealistic. In reality machines are not perfectly synchronous. Because these machines abstract from any specific implementation and only focus on external requirements, they are called requirement-level state machines; the translation to them is called the requirements-level translation.

The second transformation is based on UML state machines that are more realistic because they do not use the perfect synchrony hypothesis, i.e., state machines that use input queues. Because these statemachines are closer to the implementation-level, this translation is called the implementation-level translation.

Though the two types of machines, UML state machines and Harel's statecharts, have similar syntax, they have been shown to have semantic differences [79]. Eshuis showed that these two transformations are equivalent if the property checked is linear, separable and stuttering closed. Linear properties are expressed in past linear temporal logic [23]. A temporal property is separated if it can be written as boolean combinations, that is, using the Boolean operators disjunction, conjunction, and negation of stuttering-closed local properties [80]. A property is stuttering closed if the "next" time operator and its past time equivalent are not used. Eshuis claims that for a large number of activity diagrams, when analyzing separable temporal, it does not matter which type of translation is used because the outcome of the analysis is the same. Therefore, the requirement-level translation, that uses the verifiable statecharts, is unrealistic because of the perfect synchrony assumption can be used to verify the implementation-level models (UML state machines) for linear, separable, and stuttering closed temporal properties.

Zurowska and Dingle (2013)

Zurowska and Dingle (2013) [35] proposed a novel UML-based model checking approach. Their approach uses a UML profile targeting the modeling of real-time systems. Further, to alleviate the state explosion problem, the authors introduced a symbolic execution optimization technique to reduce the number of components that need to be composed and considered during the analyses. Their optimization technique, called lazy composition, reduces the number of the composed components by leveraging the communication topology and the temporal property being analyzed. To express properties and apply the lazy composition technique, they defined a new temporal logic language as an extension of the CTL logic. This language, however, does not use UML notions. Additionally, the authors described and proved the correctness of an algorithm to check the satisfaction of the properties defined in the new temporal logic. They implemented the model checking algorithm, and the lazy composition optimization technique in the prototype checker (SAUML2).

The approach does not transform UML-RT models to any model checking languages. Zurowska and Dingle argue for this approach by stating, "As opposed to several approaches to verify statecharts including, e.g., [19,15], our technique avoids translation of models into the input language of an existing model checker, which often introduces additional complexity to the analysis and the interpretation of the results. Instead, UML-RT models are analyzed with the help of a formal language designed to capture the core features of UML-RT, such as modularity, hierarchies and communication."

Lilius and Paltor (1999)

Lilius and Paltor (1999) formalized UML state machines for code generation, simulation, and model checking [81]. The authors provided a complete formalization that is based on two steps. First, they formalized the structure of a UML state machine by transforming it to a term rewriting system. In the second step, the authors defined the operation semantics of state machines that can be used for verification purposes using model checking.

Based on this formalization, Lilius and Paltor developed the vUML tool. vUML transforms a UML state machine to PROMELA specifications, the input language of SPIN model checker. Temporal properties to be checked by this method are specified in LTL. SPIN then verifies if the behavior of the model does not violate the property. If SPIN finds a counterexample, vUML transfers the SPIN trace to a UML sequence diagram to be presented to UML designers.

Zalila et al. (2013)

The authors of this paper provided an analysis approach that enables designers to rely on their domain-specific languages (DSMLs) notations while receiving the benefits of model checking tools. To specify temporal properties the authors proposed to use TOCL at the front, but instead of directly verifying TOCL properties they transform them to LTL properties that are subsequently verified. For the modeling stage of the system, the XSPEM DSML language, defined by a UML profile, is used. To enable the analysis, a model conforming to XSPEM and a TOCL property are transformed to the FIACRE formal language that is the front-end to several analysis toolboxes such as the Tina and CADP model checkers. To present the analysis results in XSPEM terms, the counterexample has to go through two levels of transformation, i.e., to Fiacre and then to XSPEM.

Table 2.8: Evaluation of analysis approaches

Approach	VRQ1 MT	1-a proof	1-b automatic	1-c Anl. results	1-d Sp. skills	VRQ2 Prop.	VRQ3 Optim.	VRQ4 heavy /light	VRQ5 industry
Baresi et al. (2013) Section 2.3.1	Yes	No	Yes	UML	No	No (LTL)	Yes	heavy	Yes
Moffett et al. (2013) Section 2.3.1	Yes	No	Yes	No	Yes	No (LTL)	Yes	heavy	No
Eshuis (2006) Section 2.3.1	Yes	No	Yes	No	No	No (PLTL)	Yes	heavy	No
Zurowska and Dingel (2013) Section 2.3.1	No	NA	NA	NA	NA	No (CTL)	Yes	heavy	No
Lilius and Paltor(1999) Section 2.3.1	Yes	No	Yes	UML	Yes	No (LTL)	No	heavy	No
Zalila et al. (2013) Section 2.3.1	Yes	No	Yes	No	No	Yes (TOCL)	No	heavy	No

2.3.2 Evaluation and Discussion of the Analysis Approaches

Table 2.8 shows the evaluation results of the six included analysis approaches. As the results show (VRQ1-MT), all the approaches, except Zurowska and Dingle (2013), transform different UML models to the input languages of the targeted model checking tools. As explained earlier, model transformation introduces accidental complexities that need to be addressed. None of the approaches that use model transformation handle all issues properly. In term of proving the correctness of a transformation (VRQ1 a), none of the approaches provides such a proof. Though all of the surveyed studies have tools that automatically transform UML models to a targeted model checking language, only Baresi et al. (2013) and Lilius and Paltor (1999) transform the analysis results back to UML. However, even with automatic transformation, all the approaches require special skills that are not UML related. For example, Lilius and Paltor (1999) transforms UML state machines to PROMELA/SPIN and counterexamples to UML sequence diagram. However, this approach demands UML designers to use LTL notations to specify properties. Further, most of the approaches require UML designers to intervene to apply an optimization technique to cope with the state explosion problem. Otherwise, these methods would be not scalable. It is noteworthy that only Zurowska and Dingle (2013), described in section 2.3.1, provide a UML-based optimization technique.

VRQ2 asks if the approach applies an object-oriented technique to specify a temporal property formally. The results show that only Zalila et al. (2013) employs one of the property specification approaches discussed in Section 2.2, specifically TOCL that is developed by Ziemann and Gogolla (2003). All of the other analysis methods rely on notations that are not object-oriented such as LTL or CTL notations. This fact suggests that future research work could focus on integrating the specification and analysis in a UML-based framework.

With regards to optimization techniques, only the technique proposed by Zurowska and Dingle (2013) provides a UML-based optimization method. All other approaches either rely on the optimization techniques provided by the targeted model checkers or do not support any optimization. Further, all the approaches are categorized to be heavyweight, as they require a considerable effort to be learned and applied effectively or require the used of mathematical notations such as LTL and CTL to specify properties. As the results show, only Baresi et al. (2013), described in section 2.3.1, applied their proposed approach in industrial context. All other methods used examples that are not from the industry to explain how the approaches are used.

Among the surveyed analysis approaches, Zurowska and Dingle (2013) is promising. The approach tries to eliminate the accidental complexities of transformation by directly defining model checking algorithms and tools. Further, the method provides a UML-based optimization technique to alleviate the state explosion problem. Though the approach is a promising one, it could be improved by (1) providing an object-oriented specification language for specifying temporal properties, and (2) showing that the approach applies to analyze industrial systems.

2.4 Open Problems and Future Research Directions

Based on the SLR results, this section presents several open problems for the specification and analysis of temporal properties in the context of the UML using model checking. In particular, it can be argued that the MDE community needs advances to overcome the problems that are related to the following categories:

1. Integrating specification and analysis approaches.
2. Model transformation.
3. Property specification.
4. Tooling.

For each of the following subsections, a category of challenges is discussed first and then research questions are presented. Although these categories are discussed individually below, some of them are quite interrelated.

2.4.1 Challenges to integrating specification and analysis

As shown in Table 2.7, many of the existing specification approaches lack analysis tools. Further, as shown in Table 2.8, only one of the analysis methods supports a UML-based property specification approach. This current situation results in two problems for UML designers. On one hand, UML designers get few benefits by formally specifying a temporal property without the ability to analyze it. For example, section 2.2.1 describes the Kanso and Taha (2012) approach that includes an implementation to parse their proposed language. This is not sufficient, as the primary motivation is the analysis of the specified properties. On the other hand, it is cumbersome to demand UML designers to be proficient in LTL or CTL notation to specify a property that will be checked by a provided analysis approach. This situation is the case for most of the analysis approaches, as shown in Table 2.8. Table 2.8 also shows that some of the approaches transform the analysis results and counterexamples to UML in order to be examined by UML designers. However, these approaches lack the simulation feature that is provided by traditional model checking. Simulation of the counterexample is necessary to locate the fault. The methods that do provide the counterexamples in UML do not consider doing the simulation of the counterexample using the UML initial model.

To improve the situation, these are the research questions that need to be addressed:

1. What are the challenges that make the integration of specification and analysis in a single UML-based framework hard?
2. How to present and simulate the analysis results in a form understandable by UML designers?
3. How do we improve tractability mechanisms?
4. What are the technical factors, if any, that complicate the development of model checking techniques, algorithms, and tools that exclusively use UML?

2.4.2 Model Transformation Challenges

Model transformations are an integral part in MDE and are used in a variety of purposes such as: code generation from models and defining translational semantics of a domain-specific language. This SLR study shows that many approaches use model transformations from UML to languages that have model checking support. These approaches aim to utilize the analysis tools that are provided by such languages. In this context, model transformation and traceability mechanisms bridge the gap between UML used for specification and model checking languages used for analysis [53].

However, these approaches that use model transformation introduce challenges that affect their efficiency and effectiveness. First, the SLR showed that all methods lack proof of correctness of the transformation. The accuracy of the analysis results depends on whether the transformation is semantics-preserving or not. Proving the termination of the transformation process is another difficulty. Second, UML designers may not have a strong knowledge of the targeted model checking languages and tools. As shown by SLR, many of these approaches require the use of specialized skills that are not UML related. For example, designers need to specify properties in LTL, CTL, or TRIO logic. UML designers must also be experts in the target language to effectively define some

UML constructs, such as inheritance, that cannot be transferred easily to the target model checking languages. Third, the results of the analysis performed by the back-end analysis tool must be presented to developers in UML terms to be examined, thus requiring another transformation process.

These are the research questions that need to be answered:

1. How can we prove that a transformation produces semantically equivalent models?
2. How can we formally prove that a transformation terminates?
3. How can we eliminate the accidental complexities that are introduced by transformation/tracability mechanisms?
4. How can we present or process extremely large counterexamples to locate faults?
5. Can we use model checking optimization techniques in the context of UML verification?

2.4.3 Challenges in Property Specification

As the results show, only one specification approach, Kanso and Taha (2012), has support of Dwyer's patterns [49]. Bradfield et al. (2002) does not support the patterns, but introduced property templatest that are similar to Dwyer's patterns. Although common knowledge of the Dwyer's patterns suggests that they are useful, this knowledge is only based on one study performed by the designers of the patterns in 1999 [78]. They claim that 92% out of 555 different properties are specifiable using their patterns. The specifications are taken from different application domains that include: hardware protocols, communication protocols, GUIs, control systems, abstract date types, avionics, operating systems, distributed object systems, and databases. However, more recent evaluation studies are needed to evaluate the expressiveness of the patterns in defining properties in new kinds of systems and new types of requirements.

Another open research direction could be devoted towards discovering properties. The analysis approaches assume that the temporal properties of a particular system are known, and are ready to be specified and analyzed. This is not the case in complex systems. If designers are not aware of the properties, they do not get specified and analyzed, which, in turn, could lead to undesired system behavior. Therefore, developing techniques to discover properties enhances the reliability of a system because such methods will determine the set inherent system properties that make sure the system is not under-specified.

These are the research questions that need to be answered for advancing property specification:

1. How can we formulate temporal properties in a graphical manner that supports both human communication as well as analysis amenability?
2. How can we determine if two temporal properties are semantically equivalent?
3. What are the types of properties that cannot be expressed by Dwyer's patterns? How can we evaluate the expressiveness of the patterns?
4. If Dwyer's patterns are limited, how can we extend them, yet maintain analyzablity and expressiveness?
5. How can we discover temporal properties that a system needs to satisfy?

2.4.4 Tooling Challenges

Software tools are the at the front-end of any software development paradigm. Software engineers base their opinion of the usefulness of a paradigm based on the quality of tools provided [82, 83]. Having quality tools maximizes the benefits of a paradigm and minimizes the difficulties designers go through to learn the paradigm. The importance of having quality tools in the success of a particular technique can be demonstrated by comparing tools in model checking and MDE. On one hand, one of the main strengths of model checking is its powerful tools that are capable of uncovering faults in industrial systems, as discussed in Secttion 2.1. On the other hand, many researchers have pointed out that inadequate MDE tools are one of the main barriers to industrial adoption of MDE [16, 84, 85, 86, 87].

Like other MDE tools, the current UML-based model checking frameworks suffer the inadequacy problem. These frameworks lack features that make them efficient, effective, and usable for use by UML designers. As discussed above and shown by Table 2.8, many of the approaches use transformation. France and Rumpe [8] contend that it is challenging to ensure that the transformation is semantically correct, and it is difficult to hide the complexities of the target model checking languages and tools from UML designers. Consequently, these two challenges limit both the effectiveness and the efficiency of these methods. The SLR results also show that many of the approaches do not provide optimization techniques specific to UML models; this restricts the scalability to check realistic systems. Finally, all the approaches are heavyweight; as such, they demand a significant investment from the users to learn and to use the notations.

These are the research questions that need to be answered for building better UML-based tools:

1. What are the desirable features of UML-based model checking tools that minimize accidental complexity?

2. What are the aspects that make the development of an effective UML-based model checker difficult?

3. How can we leverage the model checking optimization techniques in the context of model checking UML models?

4. Is developing a lightweight analysis approach usable in this context?

This subsection discusses the challenges developers using UML face when using the current UML-based model checking tools. The tooling challenges listed above agree with the results that have been observed by many MDE practitioners and researchers. For example, Bran Selic [16] asserts that overcoming the shortcomings of MDE tools is probably the most challenging issue that needs to be addressed to make MDE a standard practice in the industry. He declares “Specifically: the tools are far too complicated for most developers." The results of the SLR support Selic’s contention.

Though inadequate tools are one factor of the slow adoption of MDE practices, one can argue that the current state of MDE approaches makes developing useful tools difficult. In particular, many MDE researchers, such as Selic, France, Rumpe, and Whitle document many of the MDE technical challenges [8, 16, 83]. Adequately addressing these difficulties will pave the way for developers to produce effective tools.

2.5 Chapter Summary

This chapter examined the field of applying model checking techniques in the analysis of systems that are specified using UML. The material was categorized into specification and analysis approaches and the SLR method was applied to define and obtain answers to specific questions that are related to the two types of methods. The answers to the questions helped us identify open problems and gaps in the research.

The primary results of the SLR study are the identified challenges and open problems. Many of the existing approaches and tools are inadequate, and they introduce accidental complexities that affect their usability and effectiveness. The challenges were formulated as open research questions, that if properly addressed, will improve the state-of-the-art research. The open research questions were classified into the following groups: (1) specification and analysis integration, (2) model transformation, (3) property specification and patterns, and (4) tooling. It is a mission of the MDE community to seek answers to these questions, as they might yield insights that apply to other areas of MDE, such as model transformation and development of MDE-based tools. The research discussed in this dissertation intends to develop a framework that addresses some of the challenges presented in this chapter. The following chapter describes the proposed UML framework.

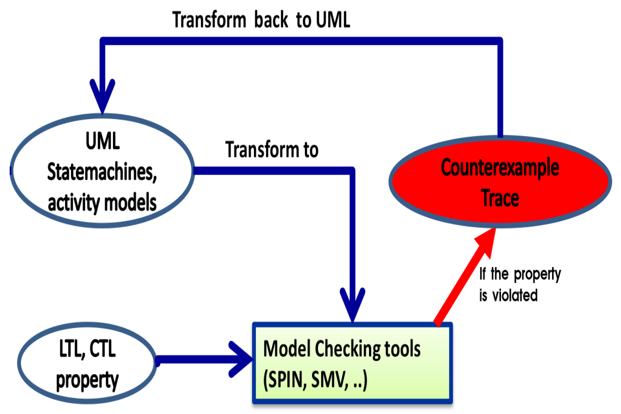
Chapter 3

The Proposed UML Framewrok

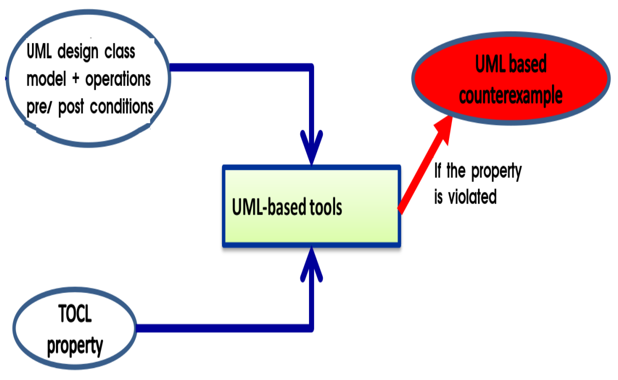
The results of the SLR study, discussed in the previous chapter, identifies some challenges. These difficulties are grouped into four categories: (1) challenges related to integrating specification and analysis, (2) model transformation challenges, (3) temporal property specification challenges, and (4) challenges associated with existing MDE tools. In this chapter, we focus on addressing some of the challenges related to integrating specification and analysis, and property patterns. In the following chapter, we address some of the tooling challenges. This chapter discusses and provides details of how the proposed UML framework addresses some of the problems. Specifically, the framework consists of the following techniques and their associated implementations:

1. A UML-based analysis technique that exclusively uses UML notations and tools
2. A property specification technique that improves the process of specifying temporal properties for UML designers
3. An optimization technique that reduces the time needed for analysis, allowing the analysis to be scaled to larger UML models

The rest of the chapter is organized as follows. An overview of the proposed approach is given in Section 3.1. In Section 3.2, I illustrate the proposed analysis technique using a simple example. Section 3.3 describes how UML designers can use the specification method to specify temporal properties. In Section 3.4, I describe the optimization technique.


Transform back to UML
UML Statemachines, activity models
Transform to
Counterexample Trace
If the property is violated
LTL, CTL property
Model Checking tools (SPIN, SMV, ..)

(a) Current approaches


UML design class model + operations pre/ post conditions
UML based counterexample
If the property is violated
UML-based tools
TOCL property

(b) Proposed approach

Figure 3.1: A comparison between the research trend and the proposed approach. Note that using the proposed approach, no transformations are required to/from model checking languages and tools, and the model type used is a design class model instead of state machine or activity models.

3.1 Overview of the Framework

3.1.1 The framework design decisions

Figure 3.1 shows how the proposed framework deviates from the state-of-the-art approaches. Two design decisions make the framework unique:

1. **Design decision 1:** Building "native" UML-based techniques and tools to do the specification and the analysis tasks. Recall that a native UML-based technique is defined as one that exclusively uses UML notations and tools.

2. **Design decision 2:** Using the UML design class model to specify both the system structure and behaviors that are expressed in OCL pre- and postconditions instead of UML state machines or activity models.

Instead of transforming UML models to model checking languages and tools (as many state-of-the-art approaches), the proposed framework uses UML notations and tools for specifying and analyzing temporal properties.

Design decision 1 has some consequences. On the positive side, this design decision is intended to eliminate the accidental complexities that the UML designers face when they transform UML designs to model checking languages and tools. It is also aimed at making the framework more usable to the UML designers since they are not required to gain expertise of the model checking techniques used to either specify temporal properties or to understand the analysis results. Therefore, the proposed framework is more "fit-the-purpose" for UML designers than other methods requiring learning technologies that UML designers are not accustomed to using.

On the negative side, UML designers lose the power that model checking techniques and tools provide. For example, the model checking community has developed some of the best software systems such as SPIN [5]. A set of mature optimization algorithms also exists that alleviate the state explosion problem. Also, the property specification patterns (Dwyer et al. [49]) are defined in formal languages that are processed by different model checkers. By developing a "native" UML-based framework, UML designers lose these powerful tools and techniques.

To address the negative side of the design decision 1, we need to develop techniques and tools whose functionality are similar to those of model checking techniques. This chapter discusses the analysis, specification, and the optimization techniques that aim to address the functionality.

Traditionally, the behavior of a system is modeled using UML state machines or UML activity diagrams. Unlike the state-of-the-art approaches that use UML state machines and activity diagrams, the proposed framework utilizes UML design class models that are augmented with operation specifications to specify the behavior.

This design decision has some consequences as well. On the positive side, UML class models play a central role in MDE software development. All other types of models require a class model to be consistent. For example, a UML state machine is a model that represents the behavior of objects that are instances of classes in a class model. Relying on UML design class models to specify the behavior brings some advantages. First, UML designers are more accustomed to using class models to specify their designs than other types of UML models [10, 15]. Other types of UML models need to be consistent with UML class models. Using the UML class model to define both the structure and the behavior of the system eliminates the need to check the consistency between

[5]SPIN model checker was the winner of the 2001 ACM Software System Award [28]

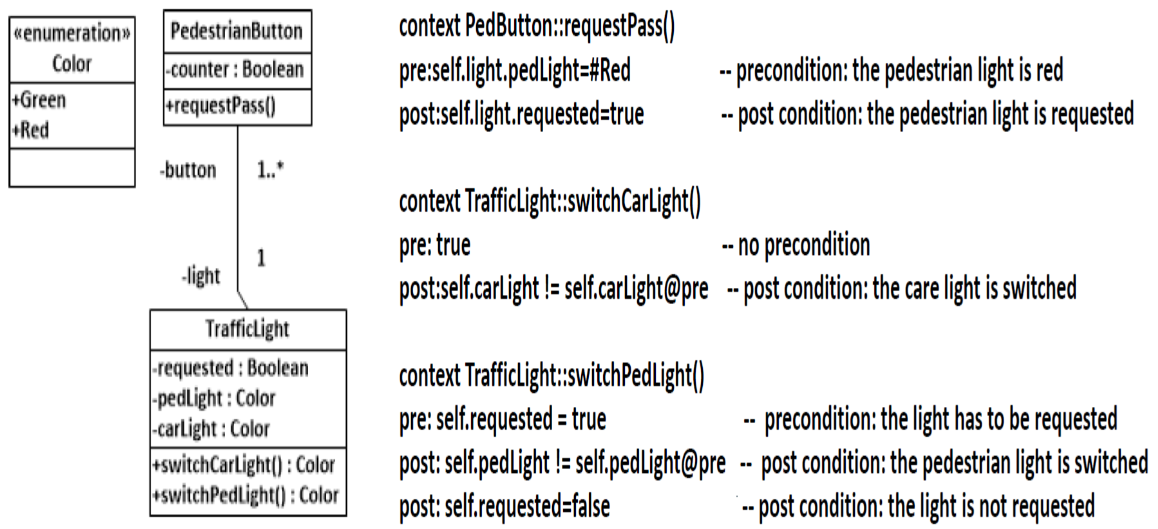
different types of models. Lastly, many mature UML tools, such as USE Model Validator [60] and OCLE [88], are built based on UML class models . Therefore, these tools and similar ones can take advantage of the techniques developed in this dissertation.

Two issues arise concerning the use of OCL to represent behavior. Different model types have different expressive power; therefore, certain models can not express certain properties, but others can and vice versa. The first issue is concerning the expressiveness power of OCL to describe the behavior of a system and whether OCL is as expressive as UML state machines or activity models. Researchers have shown that UML class models augmented with OCL operation specifications can represent the same behavior of a system as state machines [89, 90]. Consequently, UML designers can use class models, augmented with OCL operations specification, without sacrificing the expressiveness of UML state machines.

The second issue is that OCL is a textual language, and might not be as intuitive as the graphical representations of behavior such as UML state machines or activity models. For some people, representing the operations conditions in OCL textually is more involved than drawing some state machines visually. Choosing the model type to represent behavior is a preference matter as some designers like visual representation and others prefer textual form. Even for designers who like to represent behavior using state machines, OCL is still needed to describe other aspects of the design such as class invariants and state machines conditions. Therefore, one can argue that the use of OCL is unavoidable.

3.1.2 Motivating Example

To put the two design decisions into context, consider the UML design class model in Figure 3.2. The figure presents a model of a simple traffic light controller. The behavior of the system is specified using operation specifications that are expressed in OCL. The system consists of a set of traffic lights and pedestrian buttons. Each of the lights is connected to some buttons. Three operations change the state of the system, i.e., requestPass(), switchCarLight(), and switchPedLight(). Intuitively, a pedestrian requests to cross the street by pressing a pedestrian button and waits for



```
context PedButton::requestPass()
pre:self.light.pedLight=#Red             -- precondition: the pedestrian light is red
post:self.light.requested=true           -- post condition: the pedestrian light is requested

context TrafficLight::switchCarLight()
pre: true                                -- no precondition
post:self.carLight != self.carLight@pre  -- post condition: the care light is switched

context TrafficLight::switchPedLight()
pre: self.requested = true                   -- precondition: the light has to be requested
post: self.pedLight != self.pedLight@pre     -- post condition: the pedestrian light is switched
post: self.requested=false                   -- post condition: the light is not requested
```

Figure 3.2: A design class model of simple traffic light system

the pedestrian light to become green. Cars can pass when no pedestrian requests to cross the street. By pressing the button, the requestPass() operation is invoked causing the light connected to the button to be claimed. The switchCarLight() and switchPedLight() are invoked to change the color of the car light and pedestrian light from green to red and vice versa, respectively. The pre- and postconditions of the operations are expressed as OCL specifications in Figure 3.2.

Suppose that a UML designer needs to check if the system satisfies the following temporal property "after requesting to cross the street, a pedestrian should eventually be able to cross the street." To check the property, the UML designer first needs to formally specify it. Traditional OCL, being a first-order logic, cannot specify this temporal property as it specifies a constraint on a sequence of states, which is a requirement that is hard to express in first-order logic. That is, OCL is only capable of expressing state invariants restricting permissible object configurations in one state whereas a temporal property restricts the allowable sequence of states. To define the property, we can use a temporal extension to OCL (e.g., any of the specification approaches described in Chapter 2). Our proposed framework uses the TOCL language to specify the above temporal property. The TOCL property is given below. (Section 3.3 describes the specification technique used to obtain such expression.)

```
context TrafficLight inv Pedestrians_Can_Cross_Street:
```

```
(self.requested = true  implies sometime   self.pedLight= # Green) and
(self.pedLight= # Green implies sometimePast self.requested = true)
```

There are few points that are worth noting regarding the above TOCL expression. First, the expression is a temporal property, not a state invariant although it uses the *inv* OCL keyword. Second, the notations used to specify the property are similar to OCL. A UML designer who is trained to use OCL will find TOCL more familiar than notations such as LTL or CTL. Lastly, *sometime* and *sometimePast* are temporal operators . These operators provide the required constructs to specify the requirement that something eventually becomes true or has become true in the past. TOCL provides other operators such as *next*, *always*, and *until* to specify many kinds of temporal properties.

Given the system and the TOCL specification, we must answer the question: *Does the system behavior specified by the operation specifications allow a system execution that violates the temporal property?*

3.2 The Analysis Technique

We answer the question above with a lightweight analysis technique. Figure 3.3 presents a general overview of the technique's steps and Figure 3.4 provides details to applying the steps to the simple traffic light system. At the front-end of Figure 3.3, a UML designer is responsible for 1) creating an *Application Design Class Model (ADCM)* and 2) defining a temporal property in TOCL. An $ADCM$ has structural and behavioral aspects similar to the traffic light class model shown in Figure 3.2. A TOCL property is an instance of a property specification pattern (section 3.3). Conceptually, at the back-end, the USE model validator generates a set of *scenarios* against which the temporal property is checked. A scenario is a sequence of state transitions that describes a particular system execution. If the $ADCM$ model allows scenarios that violate the property, the tool returns one such scenario as a counterexample.

As depicted in Figure 3.3, the technique involves four main steps that are explained in more details using the traffic light example in Figure 3.4.

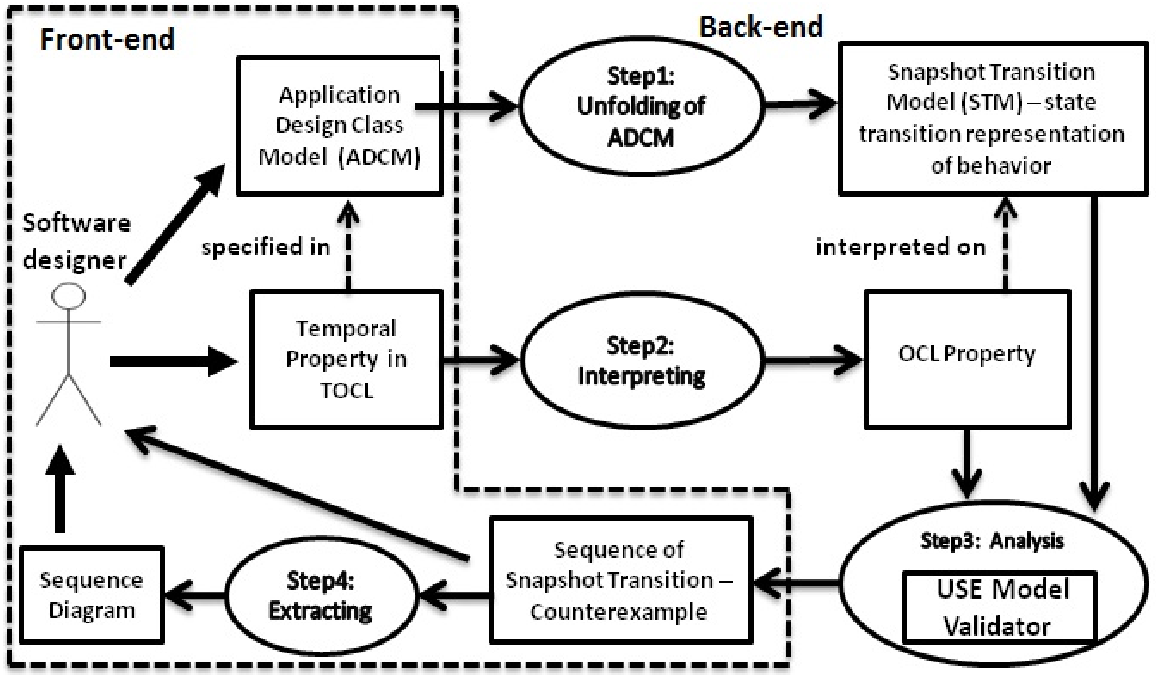
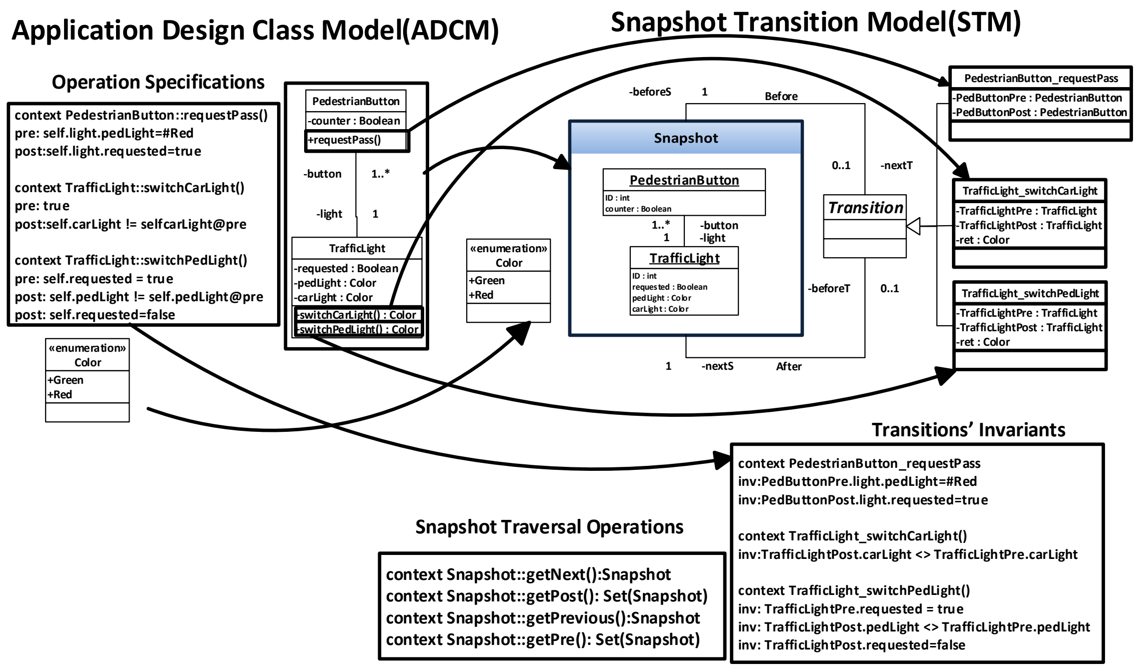

Front-end
Back-end
Application Design Class Model (ADCM)
Step1: Unfolding of ADCM
Snapshot Transition Model (STM) – state transition representation of behavior
Software designer
specified in
interpreted on
Temporal Property in TOCL
Step2: Interpreting
OCL Property
Step3: Analysis
USE Model Validator
Sequence of Snapshot Transition – Counterexample
Step4: Extracting
Sequence Diagram

Figure 3.3: An Overview of the Analysis Approach, extended from our previous work (Al-Lail et al. [73]).

3.2.1 Step1: Unfolding of the ADCM's Behavior Figure. 3.4(a)

An $ADCM$ has structural and behavioral components. Step 1 integrates these two aspects in a form amenable for temporal analysis. Towards this goal, Step 1 systematically converts an $ADCM$ to another class model called *Snapshot Transition Model* (STM). An STM represents all valid sequences of state transitions (i.e., S_0, t_0, S_1, t_1, etc.). Like any transition model, an STM has states and transitions (refer to 3.4(a)) for the visual representation). A state in an STM is called a $snapshot$ and it is a UML structured class that represents a configuration of objects. In UML, a structured class is a class that contains other classes as its parts. For example, S_0 can be a snapshot in which two objects are linked and their attribute values are provided. In the context of $STMs$, transitions are objects of the *Transition* class. These transition objects represent operation calls and have an association with the before and the after snapshots. For example, T_0 is an object that could represent a call for an operation that changes an attribute's value or delete a link between two objects. This operation call, along with its attributes, is represented as an object of the $Transition$ class. Query operations of an *ADCM* do not have side-effects on the system state; therefore, they are not transitions and do not need to be represented in an STM. As such, an STM captures all



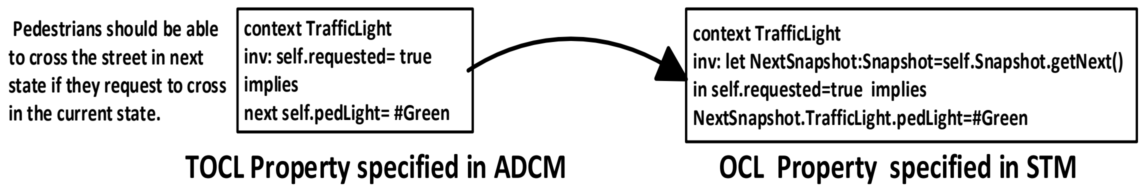
Application Design Class Model(ADCM)
Snapshot Transition Model(STM)
Operation Specifications
PedestrianButton
-counter : Boolean
+requestPass()
-button
1..*
-light
1
TrafficLight
-requested : Boolean
-pedLight : Color
-carLight : Color
+switchCarLight() : Color
+switchPedLight() : Color
«enumeration» Color
+Green
+Red
-beforeS
1
Before
Snapshot
PedestrianButton
ID : int
counter : Boolean
TrafficLight
ID : int
requested : Boolean
pedLight : Color
carLight : Color
0..1
-nextT
Transition
-beforeT
0..1
1
-nextS
After
PedestrianButton_requestPass
-PedButtonPre : PedestrianButton
-PedButtonPost : PedestrianButton
TrafficLight_switchCarLight
-TrafficLightPre : TrafficLight
-TrafficLightPost : TrafficLight
-ret : Color
TrafficLight_switchPedLight
-TrafficLightPre : TrafficLight
-TrafficLightPost : TrafficLight
-ret : Color
Transitions' Invariants
Snapshot Traversal Operations

```
context PedestrianButton::requestPass()
pre: self.light.pedLight=#Red
post:self.light.requested=true

context TrafficLight::switchCarLight()
pre: true
post:self.carLight != selfcarLight@pre

context TrafficLight::switchPedLight()
pre: self.requested = true
post: self.pedLight != self.pedLight@pre
post: self.requested=false
```

```
context PedestrianButton_requestPass
inv:PedButtonPre.light.pedLight=#Red
inv:PedButtonPost.light.requested=true

context TrafficLight_switchCarLight()
inv:TrafficLightPost.carLight <> TrafficLightPre.carLight

context TrafficLight_switchPedLight()
inv: TrafficLightPre.requested = true
inv: TrafficLightPost.pedLight <> TrafficLightPre.pedLight
inv: TrafficLightPost.requested=false
```

```
context Snapshot::getNext():Snapshot
context Snapshot::getPost(): Set(Snapshot)
context Snapshot::getPrevious():Snapshot
context Snapshot::getPre(): Set(Snapshot)
```

(a) Step1: Unfolding of the $ADCM$'s behavior as an STM

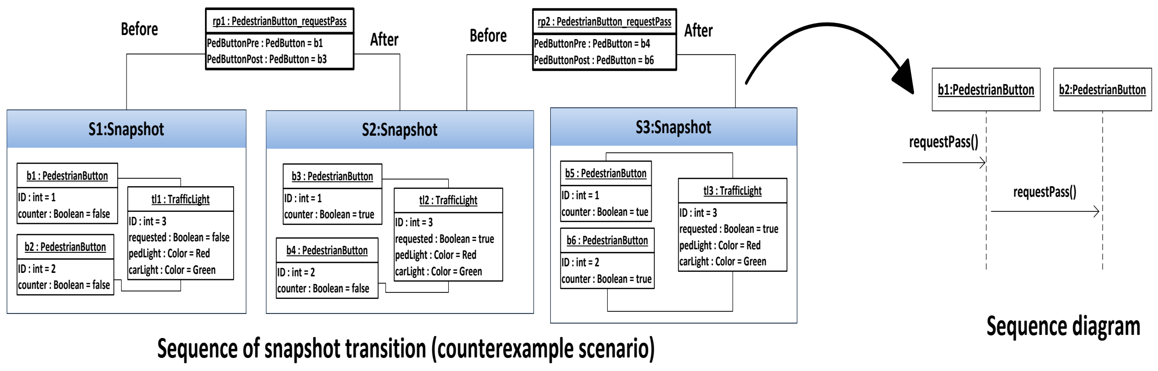


Pedestrians should be able to cross the street in next state if they request to cross in the current state.
TOCL Property specified in ADCM
OCL Property specified in STM

```
context TrafficLight
inv: self.requested= true
implies
next self.pedLight= #Green
```

```
context TrafficLight
inv: let NextSnapshot:Snapshot=self.Snapshot.getNext()
in self.requested=true implies
NextSnapshot.TrafficLight.pedLight=#Green
```

(b) Step2: Interpreting TOCL as OCL, note Step 3, analysis is carried out by USE Model Validator and is not shown in this diagram



(c) Step4: Extracting a sequence diagram from a sequence of snapshot transition counterexample scenario

Figure 3.4: An application of the analysis technique to the traffic light system (the conceptual diagram was initially presented in Al-Lail [91]).

possible states of an $ADCM$ (using the Snapshot class) and it keeps information about successive operation calls that change the states, using objects of the $Transition$ class. This step is called unfolding of an $ADCM$ behavior.

Step 1, the unfolding of the $ADCM's$ behavior step uses the definition of $STMs$ as provided by members of the Software Engineering group at Colorado State University [61]. In our research, Step 1 builds on the algorithm for generating $STMs$ from an $ADCMs$. Ordinary $STMs$, as described by Yu et al. [61], do not support the specification and analysis of temporal properties. Towards this goal, the STM generation process is improved by defining the $SnapshotTraversalQueryOperations$ to add the needed support for temporal properties. In addition, we extended the generation process to provide support for other UML concepts such as inheritance.

An STM is formed by:

1. creating the Snapshot class whose instances represent states,
2. creating a hierarchy of transition classes in which each concrete subclass represents calls of an operation specified in the $ADCM$,
3. converting operation specifications specified in an $ADCM$ to invariants on the transitions representing operation calls in the STM,
4. defining the $SnapshotTraversalQueryOperations$, and
5. adding invariants to ensure well-formed STMs.

The first three steps above are taken from the algorithm presented by Yu et al. [61]. However, extensions were added to enhance the algorithm to support various UML concepts such as inheritance. Steps four and five are additions to the algorithms. The fourth step provides support for temporal property specification and analysis. The fifth step ensures that the STM is well-formed.

Concerning the first step (creating the Snapshot class) the concrete classes of an $ADCM$ are included as parts of the Snapshot structured class in an STM. For example, as shown in Figure 3.4(a), the PedestrianButton and TrafficLight classes in the $ADCM$ are included as parts of the Snapshot class in the STM. The attributes, associations, and class constraints in the $ADCM$ classes are transformed to the STM counterpart. Recall that an instance of the Snapshot class is an object configuration of the system. For example, a snapshot object can have two traffic lights both of which have a green color for cars to pass and a pedestrian requests one of them.

The original algorithm does not consider cases in which the $ADCM$ has inheritance hierarchies between classes. Because inheritance relations are common in UML class models, this research adds support of inheritance. If an $ADCM$ contains inheritance hierarchies, they are first flattened to produce an $ADCM$ that consists only of concrete classes with no inheritance structures. Then the concrete classes are included as parts of the Snapshot class. Flattening inheritance relations neither affects class functionality, nor affects the behavior of the system.

The second step creates a hierarchy of transition classes. As shown in Figure 3.4(a), in an STM, the abstract class *Transition* represents operation calls that modify the state of a system from one snapshot to another. Therefore, the *Transition* class has two associations with the Snapshot class to represent the before and after snapshots of the operation calls. The operations in $ADCM$ are specified as subclasses of the Transition class. To create the hierarchy of transition classes in an STM, an abstract Transition superclass is first created, then a transition subclass is generated for each operation that has side-effects in an $ADCM$. To illustrate, there are three operations in the traffic light $ADCM$ (requestPass(), switchCarLight(), and switchPedLight()). Therefore, three transition subclasses are created (PedestrianButton_requestPass, TrafficLight_switchCarLight, and TrafficLight_switchPedLight) to represent the operations. Objects of these transition subclasses represent specific calls of the operations that the subclasses are derived from. For example, an object of the $PedestrianButton\_requestPass$ class, on the top right side of Figure 3.4(a), represents a call of the operation requestPass() on a particular object of the PedestrianButton class in the $ADCM$.

Note that each Transition subclass has a number of attributes. These attributes are created to represent: 1) the before and after states of the object on which the operation is called, 2) the parameter values of the operation call, and 3) the return value that the operation returns after the call. The object on which an operation is invoked is modeled by two references (shown as attributes in the Transition subclasses) that point to the object's states before and after the operation call. For example in Figure 3.4 (a), the $PedestrianButton\_requestPass$ transition class has two attributes *PedButtonPre* and *PedButtonPost* that point to the before and after states of the pedestrian button object that is invoked upon.

The return value of an operation in an $ADCM$ is modeled as an attribute in the corresponding transition subclass, in the associated STM. For example, consider the $swithchPedlight() : Color$ operation in the TrafficLight class in the $ADCM$. The operation returns a value of type Color; therefore, the $TrafficLight\_swithchPedLight$ transition class in the STM has an attribute *ret* of type Color.

Operations can have parameters of various types and they need to be represented in the STM. Parameters that have types other than classes of $ADCM$ (i.e., Integer, Double, Boolean, String, etc.) are represented as attributes of the corresponding transition subclasses. This representation is similar to representing the return value of an operation. For example, if the $switchPedlight() : Color$ operation had a parameter *t* of type Boolean, the operation's corresponding transition class in the STM ($TrafficLight\_switchPedLight$) would have an attribute *t* of type Boolean.

A parameter of an operation can be of a class type, such as TrafficLight. Each object parameter of an operation in $ADCM$ is modeled as two attributes representing the before and after states of the parameter, object, before and after the operation call. Note that in the traffic light system, none of the operations have parameters.

Recall that the transition subclasses represent operation calls. The third step is concerned with converting the operations' pre- and postconditions to invariants defined on the transition subclasses. These invariants ensure that the transition subclasses are only instantiated when the operation pre- and postconditions are satisfied. Therefore, the restrictions on the behavior of every operation is captured in the STM.

Table 3.1: An example of converting operation pre- and postconditions to invariants

requestPass() specification in ADCM	invariants in STM
context PedestrianButton::requestPass()	context PedestrianButton_requestPass
pre: self.light.pedLight=#Red	inv: PedButtonPre.light.pedLight=#Red
post: self.light.requested=true	inv: PedButtonPost.light.requested=true

Consider the requestPass() specification and the corresponding invariants in Table 3.1. The invariants in the STM are generated from operation specifications of the $ADCM$ as follows. On the $ADCM$ side, the pre- and postconditions of the operation have the keyword *self* that refers to the before and after states of the object on which the operation is called. These object states are represented by references in the STM (i.e., PedButtonPre and PedButtonPost) and are defined using invariants constraining the two references that represent the object on which the operation is called (i.e., PedButtonPre and PedButtonPost). Similarly, Figure 3.4(a) shows the invariants that are created from the OCL specifications of $switchCarLight()$ and $switchPedLight()$ operations.

Step four is concerned with defining the Snapshot Traversal Query Operations that aid the specification and analysis of temporal properties. Recall that temporal properties are those that involve defining constraints on a sequence of system state. It is necessary to define operations that navigate a sequence of snapshots. In model checking techniques, operators are defined to retrieve direct predecessors and successors of a state. Reachable states, from a specific state, are also defined (see [92, chapter 2]). That is, the $Post(s)$ and $Pre(s)$ operators return the direct successors and predecessors of a state *s*.

The process to generate ordinary $STMs$, as described by Yu et al. [62], however, does not define operators similar to $Post(s)$ and $Pre(s)$ of model checking. We extended the process to use *Snapshot Traversal Query Operations* to address the need for these operators.

Figure 3.4 (a) shows the signatures of the Snapshot Traversal Query Operations. The operation `getNext()` returns the next snapshot (direct successor) and the operation `getPost()` returns the set of all snapshots that come after a snapshot (reachable states). The operations

```
context Snapshot::getNext(): Snapshot
body: self.nextT.nextS

context Snapshot::futureClosure(s:Set(Snapshot)):Set(Snapshot)
body:if if s->includesAll(s.getNext()->asSet()) then s else
   futureClosure(s->union(s.getNext()->asSet()))endif


context Snapshot::getPost(): Set(Snapshot)
body: self.futureClosure(Set{self.getNext()})

context Snapshot::getPrevious(): Snapshot
body: self.beforeT.beforeS

context Snapshot::previousClosure(s:Set(Snapshot)):Set(Snapshot)
body:if s->includesAll(s.getPrevious()->asSet()) then s else
   previousClosure(s->union(s.getPrevious()->asSet()))endif

context Snapshot::getPre(): Set(Snapshot)
body: self.previousClosure(Set{self.getPrevious()})
```

Listing 3.1: The OCL definitions of the Snapshot Traversal Query Operations

`getPrevious()` and `getPre()` are defined similarly. Listing 3.1 provides the OCL definitions of these operations. Note that `getPost()` and `getPre()` define reachable snapshots and require the concept of transitive closure operator to retrieve all the future and previous snapshots. We defined a transitive operation using recursion since the OCL standard lacke a defintion of transitive closure when we began the work. The OCL standard now includes a transitive closure operator and part of our future work is to take advantage of it.

The operations `getPost()` and `getPre()` are defined recursively to specify transitive closure. These operations ($getNext()$, $getPost()$, $getPrevious()$, and $getPre()$) were defined and tested with the OCL evaluator of the USE tool. An example of the expected results for $getPost()$ is shown in Figure 3.6. Appendix A shows the expected results of other operations.

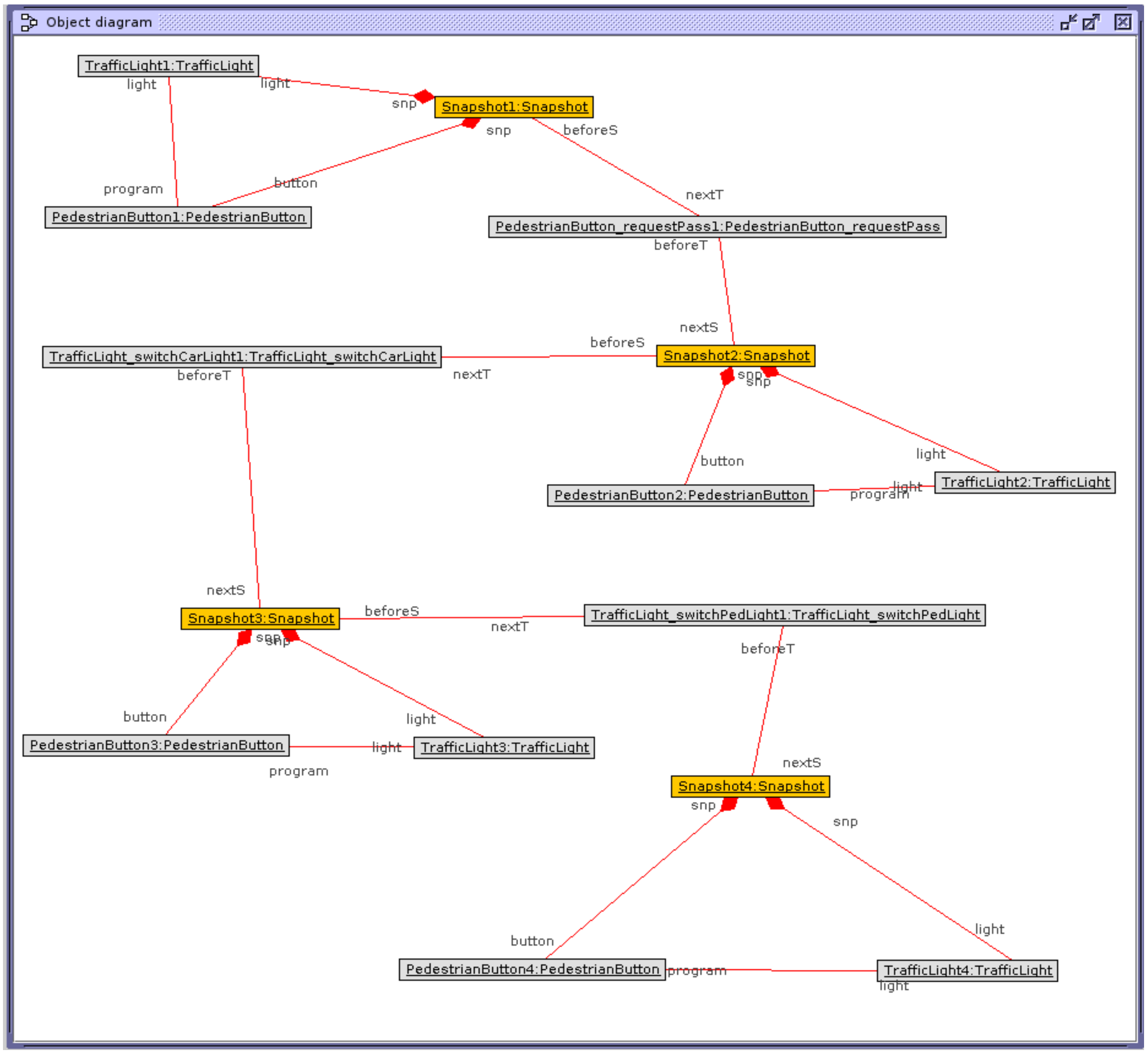
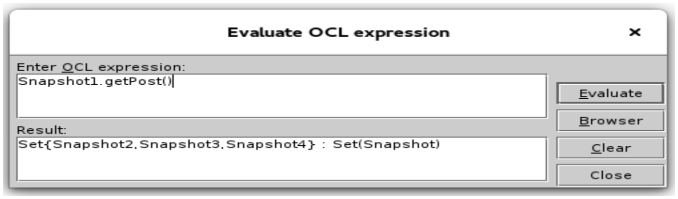

Object diagram
TrafficLight1:TrafficLight
Snapshot1:Snapshot
PedestrianButton1:PedestrianButton
PedestrianButton_requestPass1:PedestrianButton_requestPass
TrafficLight_switchCarLight1:TrafficLight_switchCarLight
Snapshot2:Snapshot
PedestrianButton2:PedestrianButton
TrafficLight2:TrafficLight
Snapshot3:Snapshot
TrafficLight_switchPedLight1:TrafficLight_switchPedLight
PedestrianButton3:PedestrianButton
TrafficLight3:TrafficLight
Snapshot4:Snapshot
PedestrianButton4:PedestrianButton
TrafficLight4:TrafficLight
light
snp
button
program
beforeS
nextT
beforeT
nextS

Figure 3.5: An example scenario, produced using USE Model Validator to show the correct functionality of the Snapshot Traversal Query Operations specified in Listing 3.1.


Evaluate OCL expression

Enter OCL expression:

```
Snapshot1.getPost()
```

Result:

```
Set{Snapshot2,Snapshot3,Snapshot4} : Set(Snapshot)
```

Figure 3.6: The getPost() operation result when invoked on Snapshot1 of the scenario depicted in Figure 3.5

The last step in generating $STMs$ is to add constraints to the STM model to ensure it is well-formed. This step is an addition to the algorithm described by Yu et. al. [61]. Table 3.2 lists the invariants that are added to an STM.

Table 3.2: The STM invariants

No.	OCL Expression
1	context Snapshot inv **AcyclicScenario**: - - prevents loops self.getPost()→excludes(self) and self.getPre()→excludes(self)
2	context Snapshot inv **OneScenario**: - - ensures all scenarios are balanced Snapshot.allInstances()→collect(s:Snapshot | s.getPrevious().oclIsUndefined())→size() = 1 and Snapshot.allInstances()→collect(s:Snapshot | s.getNext().oclIsUndefined())→size() = 1
3	context Transition inv **SameTrans** : - - ensures only one transition connecting two snapshots Transition.allInstances()→forAll(t:Transition | (self.nextS = t.nextS and self.beforeS = t.beforeS) implies self = t)
4	context Snapshot inv **SameSnapshot**: - - ensures two snapshots are the same if - - they are proceeded and lead to same transitions Snapshot.allInstances()→forAll(s:Snapshot | (self.nextT = s.nextT and self.beforeT = s.beforeT) implies self = s)
5	context PedestrianButton_requestPass inv **definedobject**: - - ensures that objects in the before and after snapshots are defined not self.PedButtonPre.oclIsUndefined() and not self.PedButtonPost.oclIsUndefined()
6	context TrafficLight_switchCarLight inv **definedobject**: - - ensures that objects in the before and after snapshots are defined not self.TrafficLightPre.oclIsUndefined() and not self.TrafficLightPost.oclIsUndefined()
7	context TrafficLight_switchPedLight inv **definedobject**: - - ensures that objects in the before and after snapshots are defined not self.TrafficLightPre.oclIsUndefined() and not self.TrafficLightPost.oclIsUndefined()
8	context PedestrianButton_requestPass inv **sameObjectID**: - - ensures that the transitions' before and after object reference are identical self.PedButtonPre.ID = self.PedButtonPost.ID
9	context TrafficLight_switchCarLight inv **sameObjectID**: - - ensures that the transitions' before and after object reference are identical self.TrafficLightPre.ID = self.TrafficLightPost.ID
10	context TrafficLight_switchPedLight inv **sameObjectID**: - - ensures that the transitions' before and after object reference are identical self.TrafficLightPre.ID = self.TrafficLightPost.ID

The first invariant is called **AcyclicScenario** and it ensures that a scenario generated from a STM is free of cycles. Loops are not allowed as every snapshot has different objects even though the objects in two snapshots have the same configurations. Allowing loops, therefore, can cause incorrect results when analyzing temporal properties. The cycles are prevented by producing scenarios in which no snapshot is included in the set of the snapshots that follow it or the set of snapshots that proceed it. The second invariant, **OneScenario**, makes sure that all possible scenarios are balanced, they have one start snapshot and one end snapshot. This constraint prevents creating snapshots that are unreachable or fragments of scenarios that are not well-connected. The **SameTrans** constraint ensures that there is only one possible transition connecting two snapshots. Similarly, the fourth invariant, **SameSnapshot** declares that if two snapshots can be reached by the same transition and can lead to the same transition, the two snapshots must be the same. The **definedobject** constraints (rows 5, 6, and 7) make sure that objects in the before and after snapshots are defined. The **sameObjectID** invariants (rows 8, 9, and 10) ensure that the transitions' objects before and after states are referencing the same objects.

3.2.2 Step2: Interpreting TOCL as OCL Figure 3.7

A temporal property checked by the analysis technique is an instance of a property pattern specified in TOCL (Section 3.3). Traditionally, a temporal logic formula, expressed in LTL, CTL, or TOCL, is interpreted as a first-order logic formula on the traces of a transition system. This is the standard method of defining the semantics of temporal logic languages [92]. In a similar manner, the TOCL property is interpreted as OCL first-order logic expression defined in the context of the STM that describes system traces. OCL can define constraints in UML class models; hence, it can be used to restrict an STM. Step 2 of the analysis technique mechanically generates an OCL expression constraining the STM from the TOCL property.

Consider the TOCL and OCL properties in Figure 3.7. The temporal property expresses the requirement that when a traffic light is requested, pedestrian light becomes green in the next state. To specify this property in TOCL notations, a UML designer needs to specify a context of the prop-

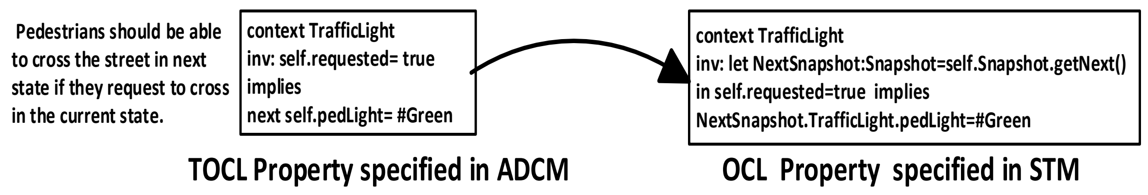

Pedestrians should be able to cross the street in next state if they request to cross in the current state.
context TrafficLight
inv: self.requested= true
implies
next self.pedLight= #Green
TOCL Property specified in ADCM
context TrafficLight
inv: let NextSnapshot:Snapshot=self.Snapshot.getNext()
in self.requested=true implies
NextSnapshot.TrafficLight.pedLight=#Green
OCL Property specified in STM

Figure 3.7: Step two of the analysis technique: Interpreting TOCL as OCL, same as Figure 3.4(b), presented here for ease of access

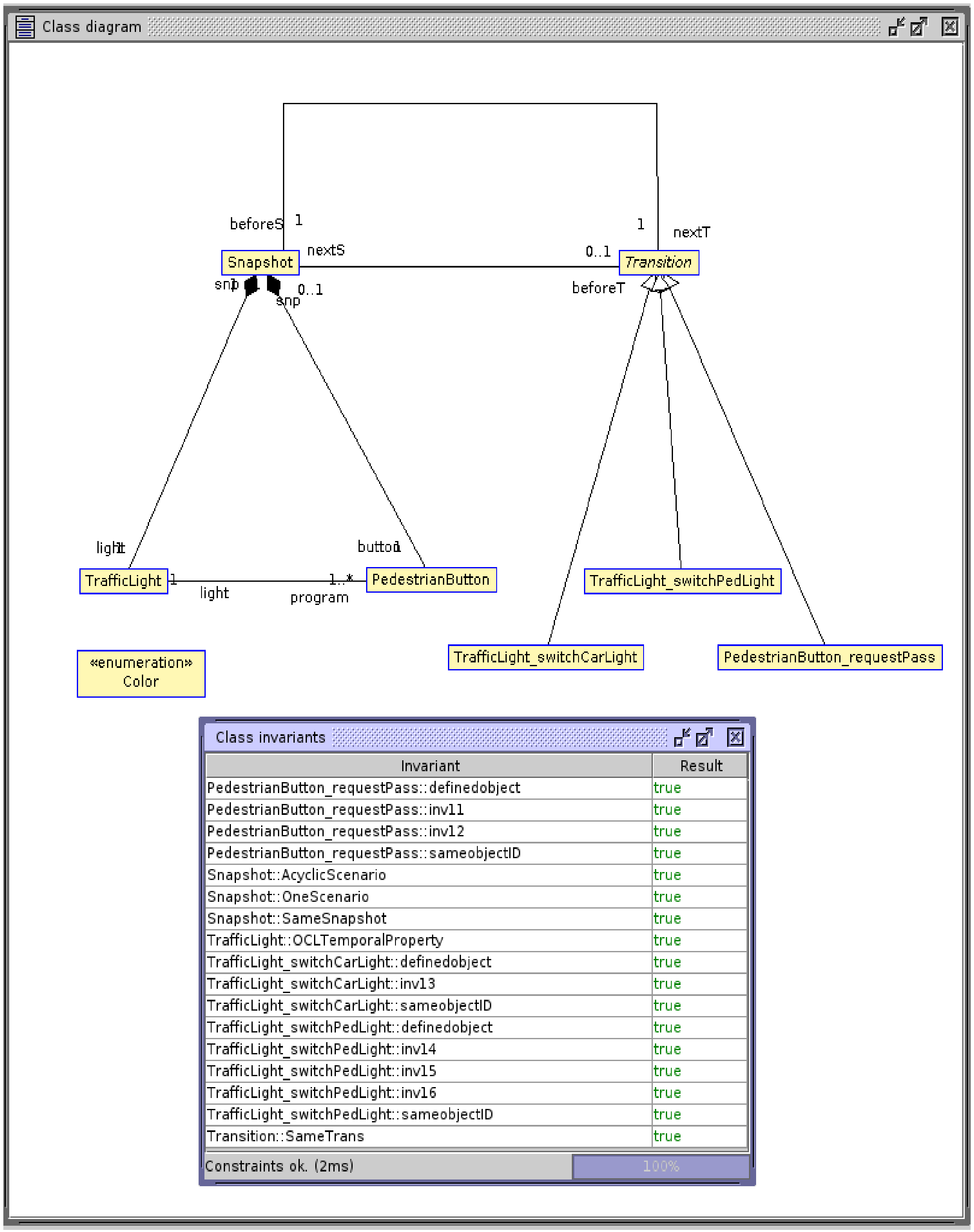
erty. The context is the TrafficLight class. The TOCL expression states that the traffic light being requested (self.requested = true) implies that in the next state (specified by the TOCL temporal operator next) the traffic light for the pedestrian should be green (self.pedLight = #Green).

The TOCL property is interpreted on the STM using OCL expressions as follows. In the OCL property, the auxiliary variable NextSnapshot is defined using the snapshot traversal operation $getNext()$. Having defined what the next state is, the OCL expression then asserts that if a traffic light is requested, the pedestrian light should be green in the NextSnapshot (e.i., NextSnapshot.TrafficLight.pedLight = # Green).

Figure 3.8 provides a graphical representation of the STM of the traffic light system, produced using USE Model Validator. Appendix A provides the complete USE textual specification of this system. The appendix includes the snapshot traversal operations, transitions invariants, the STM invariants, and the OCL expression of the temporal property. We have validated all of these by the USE Model Validator.

3.2.3 Step 3: Analysis.

Step 1 (unfolding the behavior of $ADCM$ as STM) and Step 2 (interpreting TOCL as OCL on STM) pave the way for the analysis task in Step 3. The problem of checking TOCL properties on an $ADCM$ model is reduced to checking OCL on an STM. To check the validity of a TOCL property on an $ADCM$, the USE model validator [93] searches for a counterexample scenario (an instance of STM) that violates the corresponding OCL property.



Invariant	Result
PedestrianButton_requestPass::definedobject	true
PedestrianButton_requestPass::inv11	true
PedestrianButton_requestPass::inv12	true
PedestrianButton_requestPass::sameobjectID	true
Snapshot::AcyclicScenario	true
Snapshot::OneScenario	true
Snapshot::SameSnapshot	true
TrafficLight::OCLTemporalProperty	true
TrafficLight_switchCarLight::definedobject	true
TrafficLight_switchCarLight::inv13	true
TrafficLight_switchCarLight::sameobjectID	true
TrafficLight_switchPedLight::definedobject	true
TrafficLight_switchPedLight::inv14	true
TrafficLight_switchPedLight::inv15	true
TrafficLight_switchPedLight::inv16	true
TrafficLight_switchPedLight::sameobjectID	true
Transition::SameTrans	true

Constraints ok. (2ms) 100%

Figure 3.8: The graphical representation of the STM of the traffic light system, produced using USE Model Validator. The model includes the tested snapshot traversal operations, transitions invariants, the STM invariants, and the OCL representation of the temporal property. This figure shows the correctness of the UML and OCL specification of the model.

Analysis techniques such as model checking that rely on exhaustive search for a counterexample suffer from the state explosion problem. For UML-based systems, the state explosion problem is even worse due to the dynamic allocation and deallocation of objects [94]. Additionally, in these systems, a class can have unlimited number of objects. An STM, therefore, allows the generation of an infinite number of scenarios. However, as computers have finite memory and computation power, the analysis must be restricted to check limited scenarios. Unlike heavyweight analysis techniques, such as traditional model checking in which the analysis exhaustively searches all possible executions of the system, our proposed analysis technique follows a lightweight approach to checking properties. That is, the analysis technique considers only a subset of allowable scenarios by an STM; which in turns makes the analysis computable and feasible.

We use two methods to constrain search space: search-scope and search-depth. The first method is the small-scope hypothesis that has been suggested by Daniel Jackson [63]. A search-scope defines the number of objects to be created for each class in a snapshot. To illustrate, if the search-scope is five, the USE Model Validator creates five objects of each class belonging to a particular Snapshot. The small-scope hypothesis states if a system violates a property then a small counterexample, which shows the violation, is likely to exist.

The second method is the search-depth that researchers in the area of Bounded Model Checking have applied [64]. A search-depth specifies the number of transitions (objects of the Transitions class in our case) to be considered in an analysis task. To illustrate, a designer could set the search-depth to 100, in such a case the tool only creates and checks all the possible scenarios that have up to 100 transitions.

Putting the two methods in the context of the analysis technique, if an STM violates a temporal property, a small counterexample is likely to exist. A UML designer sets the search-scope and search-depth parameters and runs the tool to find a counterexample within the constrained search space. If the analysis fails to find a counterexample, the designer might increase the search-scopes and the search-depth and run the analysis task again to provide higher confidence that the property holds. However, failure to find a counterexample does not guarantee the validity of a temporal

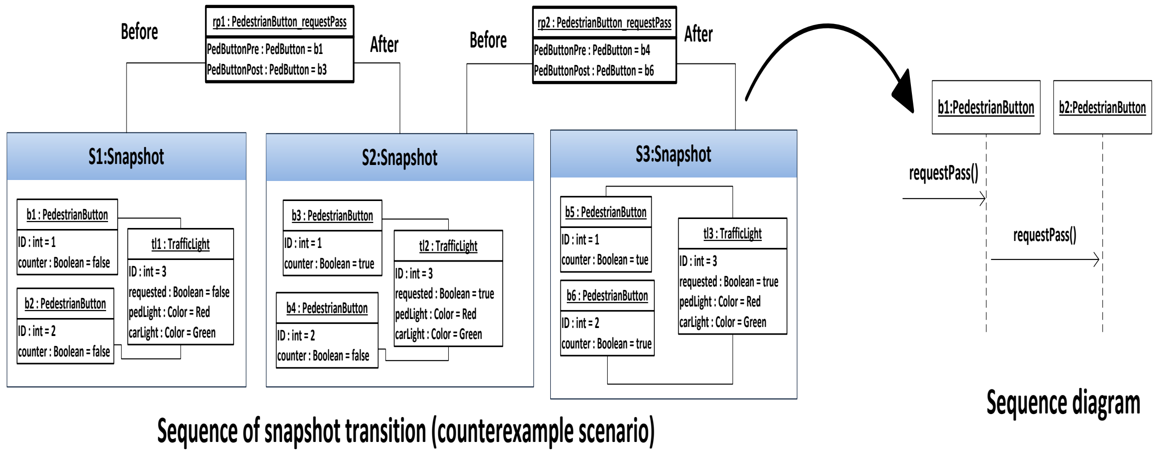


Figure 3.9: Step four of the analysis technique: Sequence diagram extraction, same as Figure 3.4(c), presented here for ease of access.

property because the analysis only covers a subset of the entire search space. In this case, the analysis tool outputs a message indicating that the property is maintained within the given search-scope and search-depth. In the case that the property is violated within the constrained search space, the tool generates a counterexample scenario that shows how the requirement is broken. The counterexample is an instance of the STM which is a UML object diagram. This format should enhance the understandability of the counterexample.

Figure 3.9 shows the counterexample scenario of our example system that violates the TOCL property shown in Figure 3.7. To uncover the fault, the designer must examine the counterexample. Recall that the property states that when the traffic light is requested, the light of pedestrian should be green in the next state. The scenario violates the property because the traffic light is requested in snapshot $S2$ (i.e., requested=true), but the pedestrian light did not turn green in the next snapshot $S3$ (i.e., pedLight=Red).

3.2.4 Step 4: Sequence diagram extraction Figure 3.9.

Step 4 is concerned with interpreting the counterexample. Because of the large number of objects and transitions a counterexample scenario might have, the result of the analysis may be complicated and difficult to examine. A design class model having few classes and few operations

Algorithm 1 Snapshot Transitions to Sequence Diagram

```
Input: Sequence of Snapshot Transition
Output: Sequence diagram
Algorithm Steps: For every object of the Class transition do
Step 1. Get the class name and the operation name that associated with transition.
Step 2. Get the object on which the operation is invoked on.
Step 3. Get the operation parameters from the transition object attributes.
Step 4. Get the return value from the ret attribute of the transition object.
Step 5. Draw a timeline for the object that the operation is invoked on from step 2.
Step 6. Draw an operation call on the object using the name of the operation and its attributes
from steps 1, and 3 above.
Step 7. Draw a return message of the operation with the value from step 4.
```

could make the results of the tool very complicated. We developed an algorithm that aids the examining process Al-Lail et al. [95]. The algorithm extracts a sequence diagram from a scenario. The designer can use the details of the sequence of the snapshot transition counterexample (the output of Step 3) to uncover the fault. The sequence diagram shows the operation calls that were performed on objects that resulted in the violation. Algorithm 1 provides a systematic way to achieve this objective.

3.3 The Property Specification Technique

Recall that specifying temporal properties in formal languages such as LTL and CTL can be challenging, as asserted by model checking experts [48, 49, 66, 78]. Specifying temporal properties may be more challenging for UML designers who might not have strong mathematical training in LTL and CTL languages.

The third objective of this dissertation is to improve the process of specifying temporal properties for UML designers. The technique improves the process by using property specification patterns that ease the effort required to specify properties, and by providing UML-based language representations of the patterns.

Section 3.3.1 provides the background on property specification patterns and section 3.3.2 gives the details of the specification technique.

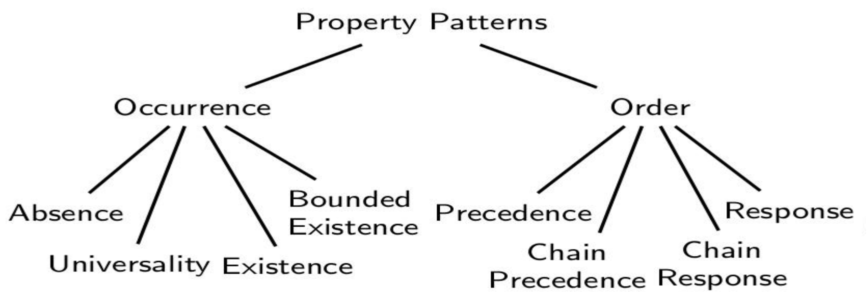


Figure 3.10: Dwyer's patterns classification hierarchy, taken from Kanso's et al. [72]

3.3.1 Dwyer's Property Specification Patterns

The difficulty of specifying temporal logic properties was tackled by Dwyer et al. [49] using patterns. They developed eight property specification patterns in different formal languages such as LTL, CTL, and QRE. The patterns are categorized in two groups, as depicted in Figure 3.10:

— **Occurrence patterns:** Concerned with the occurrence of a given event or state during system execution.

— **Order patterns:** Concerned with relative order in which multiple events or states occur during system execution.

A pattern intends to specify a set of behavioral requirements that a designer wants a system's execution to exhibit or avoid. Table 3.3 provides descriptions of the patterns. Although there are eight patterns, only three patterns (Response, Universality and its dual Absence) account for 80% of the sample specification that included 511 temporal properties Dwyer et al. [66].

A scope of a temporal property defines the portion of a system's execution where the property must hold [6]. A scope of a temporal property is determined by specifying a starting and an ending state/event for the intended behavioral requirement. That is, the scope consists of all states/events beginning with the starting state/event and up to, but not including, the ending state/event. Figure 3.11 depicts the scopes graphically and Table 3.4 specifies the portion of the system execution

[6] Note that a scope of a temporal property is different from the search-scope used in analysis.

Table 3.3: Descriptions of Dwyer's patterns, taken from Dwyer's et al. [49], organized in a table.

Occurrence Patterns	
Pattern	**Intent**
Absence	To describe a portion of a system's execution that is free of certain events or states. Also known as Never
Universality	To describe a portion of a system's execution which contains only states that have a desired property. Also known as Henceforth and Always.
Existence	To describe a portion of a system's execution that contains an instance of certain events or states. Also known as Eventually.
Bounded Existence	To describe a portion of a system's execution that contains at most a specified number of instances of a designated state transition or event.
Order Patterns	
Precedence	To describe relationships between a pair of events or states where the occurrence of the first is a necessary precondition for an occurrence of the second. We say that an occurrence of the second is enabled by an occurrence of the first.
Response	To describe cause-effect relationships between a pair of events/states. An occurrence of the first, the cause, must be followed by an occurrence of the second, the effect.
Response Chains	To describe a relationship between a stimulus event (P) and a sequence of two response events (S,T) in which the occurrence of the stimulus event must be followed by an occurrence of the sequence of response events within the scope.
Precedence Chains	To describe a relationship between an event or state P and a sequence of events or states (S, T) in which the occurrence of S followed by T within the scope must be preceded by an occurrence of the sequence P within the same scope.

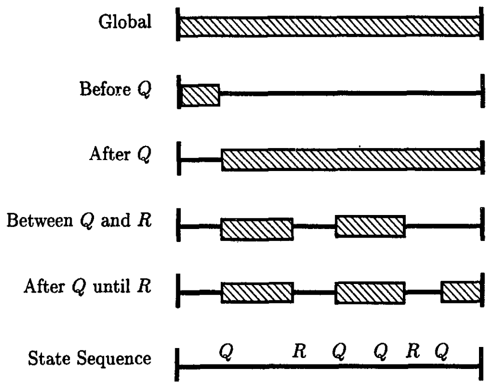


Figure 3.11: Graphical representations of Dwyer's patterns scopes, taken from Dwyer's et al. [49]

for the scopes. For example, to specify the requirement that the system should react to a particular event, a designer can use the response pattern in the global scope.

3.3.2 Dwyer's Patterns in TOCL and OCL

The patterns of Dwyer's et. al. are very useful to specify temporal properties in formal languages such as LTL, CTL, and QRE. We defined the patterns of Dwyer et al. in two UML-related languages to accommodate UML designers:

Table 3.4: Scopes of temporal properties, descriptions are taken from Dwyer's et al. [49], organized in a table here.

Scope	**Portion of the system execution**
Global	The entire system execution
Before Q	The execution up to a given state/event
After Q	The execution after a given state/even
Between Q and R	Any part of the execution from one given state/event to another given state/event
After Q until R	Like the between scope but the designated part of the execution continues even if the second state/event does not occur

1. TOCL that is used for specification of temporal properties on an $ADCM$.

2. OCL interpretation of TOCL on an STM that is used for analysis.

Table 3.5 shows the specification of the response pattern in TOCL and OCL notations. In this section, we also give the specification of the Universality pattern; appendix B provides the specifications of the other six patterns. We developed graphical illustrations of these patterns. For example, the graphical representation of the response pattern is shown in Figure 3.12.

Note that the parts of the patterns that are between square brackets (e.g., *[S $\models$ P]*) must be replaced when instances of the patterns are specified. The expression *[S $\models$ P]* in the OCL patterns means that the property P holds in the snapshot S. *P* is an OCL boolean expression in the context of a class, for example C. When using the patterns to write temporal properties, the OCL expression representing *S* $\models$ *P* is generated by navigating from the Snapshot *S* to C, the context of the OCL expression P.

A user of the specification technique must follow two steps to specify a temporal property: (1) determine the TOCL pattern and the scope that best fit a temporal requirement, and (2) derive a TOCL property by replacing the parameters of the pattern with actual context and conditions. Each TOCL pattern has interpretation rules to automatically generate the OCL counterpart. Figure 3.14 shows the graphical representation of these interpretation rules for the response pattern in the global scope. Similar interpretation rules are defined for all TOCL specification patterns.

Table 3.5: The response pattern specifications in TOCL and OCL

	TOCL in ADCM	OCL in STM
Globally	context *[Class]* inv: *[P]* implies sometime *[S]*	context *[Class]* inv: let CS: Snapshot = self.getCurrentSnapshot(), in let FS: Set(Snapshot) = CS.getPost() in *[P]* implies FS → exists(s:Snapshot | *[s* ⊨ *S]*)
Before R	context *[Class]* inv: *[R]* implies sometime *[S]* since *[P]*	context *[Class]* inv: let CS: Snapshot = self.getCurrentSnapshot() in let BS: Set(Snapshot) = CS.getPre() in let PS: Set(Snapshot) = BS → select(s:Snapshot | *[s* ⊨ *S]*) in let SS: Set(Snapshot)= BS → select(s':Snapshot | *[s'* ⊨ *S]*) in *[R]* implies PS→forAll(ps:Snapshot | ps.getPost()→exists(ss:Snapshot | SS→includes(ss)))
After Q	context *[Class]* inv: *[Q]* implies always (*[P]* implies sometime *[S]*)	context *[Class]* inv: let CS: Snapshot = self.getCurrentSnapshot() in let FS: Set(Snapshot) = CS.getPost() in let PS: Set(Snapshot) = FS → select(s:Snapshot | *[s* ⊨ *S]*) in let SS: Set(Snapshot)= FS → select(s':Snapshot | *[s'* ⊨ *S]*) in *[Q]* implies PS→forAll(ps:Snapshot | ps.getPost()→exists(ss:Snapshot | SS→includes(ss)))
After Q until R	context *[Class]* inv: *[Q]* implies always (*[P]* implies sometime *[S]*)	context *[Class]* inv: let CS: Snapshot = self.getCurrentSnapshot() in let FS: Set(Snapshot) = CS.getPost() in let FSR1: Snapshot= FS→select(s:Snapshot | [s⊨ R]) → asOrderedSet()→ first(), in let PS: Snapshot= FS → any(s:Snapshot | [s ⊨ P]), SS: Snapshot= FS → any(s':Snapshot | [s' ⊨ S]) in *[Q]* implies (PS.getPost() → includes(SS) and FSR1.getPre() → includes(PS) and FSR1.getPre()→ includes(SS)
Between Q and R	context *[Class]* inv: *[Q]* implies (*[P]* implies sometime *[S]*) before *[R]*	context *[Class]* inv: let CS: Snapshot = self.getCurrentSnapshot() in let FS: Set(Snapshot) = CS.getPost() in let FSR1: Snapshot= FS→select(s:Snapshot | [s⊨ R]) → asOrderedSet()→ first(), in let PS: Snapshot= FS → any(s:Snapshot | [s ⊨ P]), SS: Snapshot= FS → any(s':Snapshot | [s' ⊨ S]) in *[Q]* implies (PS.getPost() → includes(SS) and FSR1.getPre() → includes(PS) and FSR1.getPre()→ includes(SS)

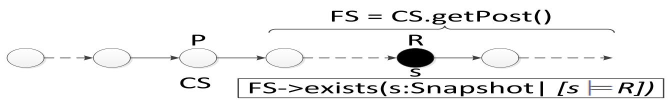


Figure 3.12: Graphical illustration of the response pattern in the global scope (the first row in Table 3.5), taken from our previous work Al-Lail et al. [73].

Table 3.6: The universality pattern specifications in TOCL and OCL

Scope	TOCL in ADCM	OCL in STM
Globally	context *[Class]* inv: always *[P]*	context *[Class]* inv: Snapshot.allInstances → forAll(s:Snapshot | *[s ⊨ P]*)
Before R	context *[Class]* inv: *[R]* implies alwaysPast *[P]*	context *[Class]* inv: let CS: Snapshot = self.getCurrentSnapshot() in let BS: Set(Snapshot) = CS.getPre() in *[R]* implies BS → forAll(s:Snapshot | *[s ⊨ P]*)
After Q	context *[Class]* inv: *[Q]* implies always *[P]*	context *[Class]* inv: let CS: Snapshot = self.getCurrentSnapshot() in let FS: Set(Snapshot) = CS.getPost() in *[Q]* implies FS → forAll(s:Snapshot | *[s ⊨ S]*)
After Q until R	context *[Class]* inv: *[Q]* implies always *[P]* until *[R]*	context *[Class]* inv: let CS: Snapshot = self.getCurrentSnapshot() in let FSR1: Snapshot= CS.getPost()→ select(s:Snapshot | *[s⊨ R]*)→ asOrderedSet()→ first() in let PreFSR1=Set(Snapshot) = FSR1.getPre(), in let CSPre: Set(Snapshot)= CS.getPre()→including(CS) in let BTS: Set(Snapshot)= PreFS1→ reject(s:Snapshot | CSPre→includes(s)) in ((*[Q]* and FSR1.isDefined) implies BTS→forAll(s:Snapshot | *[s ⊨ P]*) or ((*[Q]* and FSR1.oclIsUndefined()) implies FS→forAll(s:Snapshot | *[s ⊨ P]*))
Between Q and R	context *[Class]* inv:*[Q]* implies always *[P]* until *[R]*	context *[Class]* inv: let CS: Snapshot = self.getCurrentSnapshot() in let FSR1: Snapshot= CS.getPost()→ select(s:Snapshot | *[s⊨ R]*)→ asOrderedSet()→ first() in let PreFSR1=Set(Snapshot) = FSR1.getPre(), in let CSPre: Set(Snapshot)= CS.getPre()→including(CS) in let BTS: Set(Snapshot)= PreFSR1→reject(s:Snapshot | CSPre→includes(s)) in *[Q]* implies BTS → forAll(s:Snapshot | *[s ⊨ P]*)

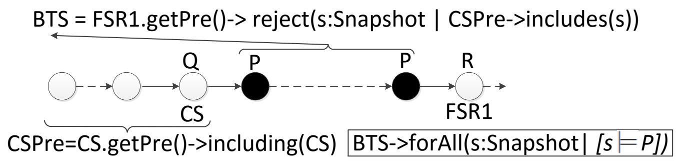
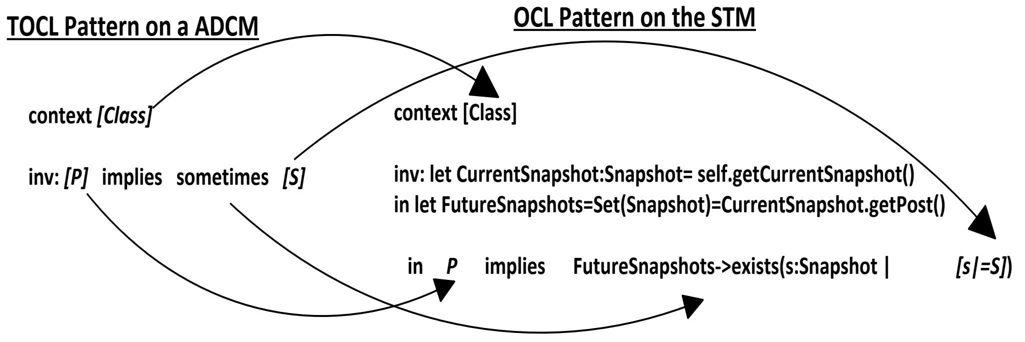

BTS = FSR1.getPre()-> reject(s:Snapshot | CSPre->includes(s))
Q
P
P
R
CS
FSR1
CSPre=CS.getPre()->including(CS)
BTS->forAll(s:Snapshot| [s ⊨ P])

Figure 3.13: Graphical illustration of the universality pattern in the between Q and R scope (the fifth row in Table 3.6) taken from our previous work Al-Lail et al. [73].

For example, to specify a temporal property using the TOCL pattern depicted in Figure 3.14, a UML designer needs to provide the context of the property (*[Class]*), and the two conditions *[P]* and *[S]*. The corresponding OCL pattern and interpretation rules are employed to automatically generate the OCL property to be used for analysis. Note how the *sometimes* temporal operator is interpreted using OCL in an STM. Informally, on the TOCL side, *sometimes S* means that the condition S should hold in one of the succeeding states (snapshots). On the OCL side, succeeding snapshots are defined using an auxiliary variable FutureSnapshots. FutureSnapshots uses getPost() snapshot traversal operation that returns the succeeding snapshots. One of these future snapshots must satisfy the condition *S* (i.e., FutureSnapshots→exists(s:Snapshot | [s⊨S]).



```
context [Class]
inv: [P]  implies  sometimes  [S]
```

```
context [Class]
inv: let CurrentSnapshot:Snapshot= self.getCurrentSnapshot()
in let FutureSnapshots=Set(Snapshot)=CurrentSnapshot.getPost()
  in  P  implies  FutureSnapshots->exists(s:Snapshot |     [s|=S])
```

Figure 3.14: Interpretation rules of the response pattern-globally scope

To illustrate the use of the specification technique, consider Table 3.7 that provides natural language descriptions of eight temporal properties of the traffic light system. A UML designer uses the descriptions to decide the appropriate patterns and scopes of the required temporal requirements (i.e., Pattern - Scope column in Table 3.7). Then the designer uses the corresponding TOCL pattern to obtain a TOCL property, from which the OCL expression is derived. Table 3.8 provides the TOCL and OCL expressions of the eight temporal properties presented in Table 3.7. Figure 3.15 shows a screenshot of the USE Model Validator indicating the correctness of the OCL expressions of these properties.

Table 3.7: Some temporal properties of the traffic light system

No.	Description	Pattern - Scope
TP1	As soon as a traffic light is requested by a pedestrian, its car light turns red.	Response-Globally
TP2	After a pedestrian request to pass, the pedestrian light stays red until the car light turns to red.	Universality-Between Q and R
TP3	Before a pedestrian light turns to green, it must have been requested and the car light is red.	Precedence-Globally
TP4	Between the time when pedestrian request to pass and the time when the car light turns red, the pedestrian light must not be green.	Absence-Between Q and R
TP5	After pedestrian light becomes red, the cars are allowed to pass until a request is made by a pedestrian.	Universality-After Q until R
TP6	The car light and the pedestrian light can not be green simultaneously.	Universality-Globally
TP7	Before pedestrians can cross the street, the car light should become Red after the pedestrians request to pass. The pedestrians must request to pass before they are allowed.	Response-Before R
TP8	After the pedestrians light become red, and before it turns green again, the car light must become Red after pedestrians have to request passing again.	Response-Between Q and R

3.4 The Optimization Technique

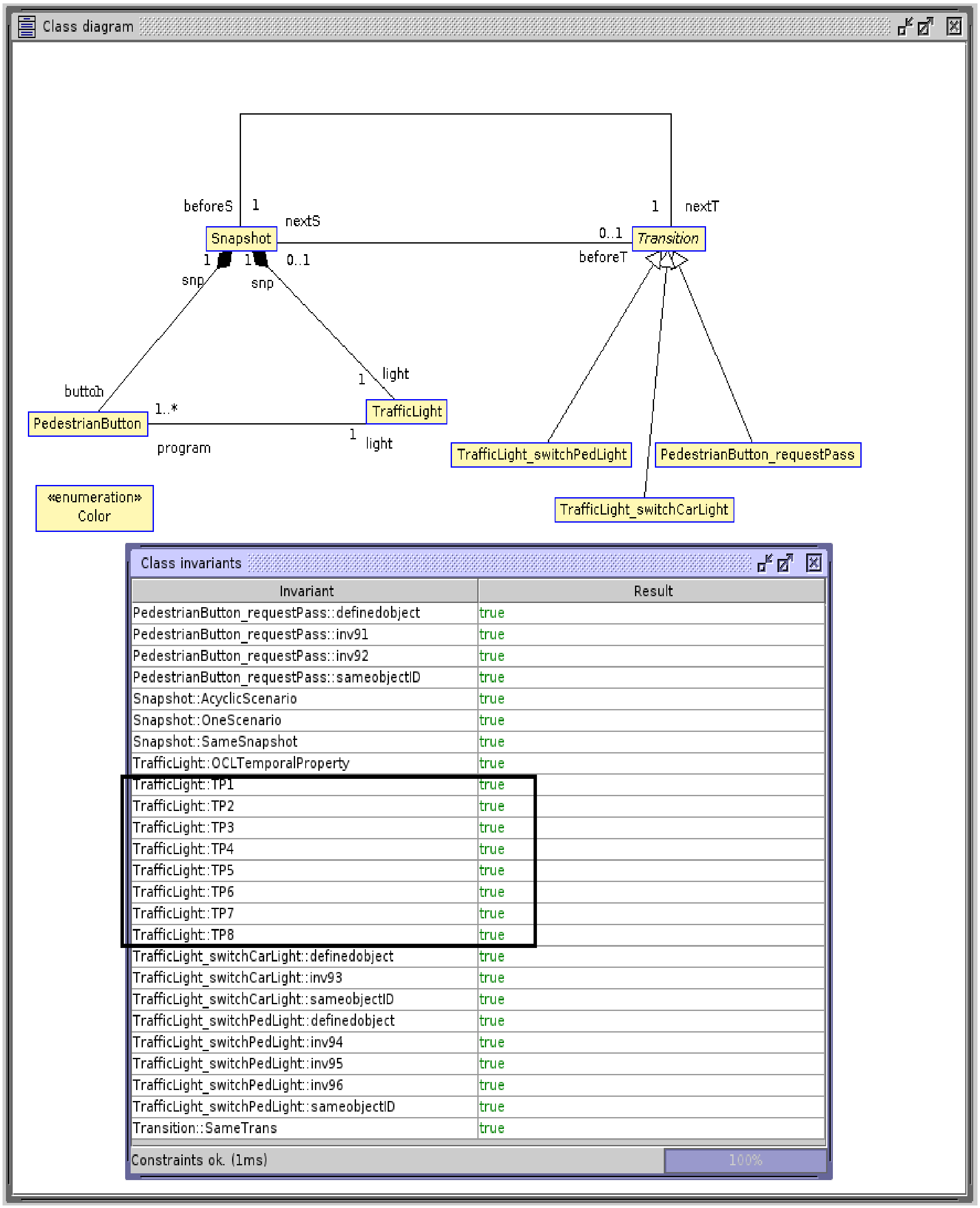
Recall that one advantage of the mainstream model checkers, such as SPIN, is the optimization techniques they use to alleviate the state explosion problem. Losing the power of such optimization techniques is a consequence of developing a "native" UML-based approach for specifying and analyzing temporal properties, as explained in the beginning of this chapter.

Section 3.2 explains the analysis technique that explores the state space of an STM to check the validity of a property. As the analysis technique is based on state exploration, the state explosion problem hinders the technique's applicability of the technique to larger class models. Both the small-scope hypothesis and search-depth can alleviate the problem when a system violates a property, as the analysis task is often able to produce a counterexample with small search-scope and search-depth. However, realistic systems are complex. Applying the technique to analyze complex systems is problematic for two reasons. First, generating and searching a state space even with

Table 3.8: TOCL and OCL specification of the temporal properties described in Table 3.7

No.	Pattern-scope	TOCL Specification on Class Model	OCL Specification on the Snapshot Transition Model
TP1	Response-Globally	context TrafficLight inv: self.requested = true implies next self.carLight=#Red	context TrafficLight inv: let CS: Snapshot= self.getCurrentSnapshot() in let NS: Snapshot= CS.getNext() in self.requested=true implies NS.light.carLight= #Red
TP2	Universality-Between Q and R	context TrafficLight inv TP2:self.requested=true implies always self.pedLight=#Red until self.carLight=#Red	context TrafficLight inv TP2: let CS: Snapshot = self.getCurrentSnapshot() in let FSR1 :Snapshot = CS.getPost()→ select(sr:Snapshot | sr.light.carLight=#Red)→asOrderedSet()→first() in let PreFSR1: Set(Snapshot)= FSR1.getPre() in let CSPre: Set(Snapshot)= CS.getPre()→including(CS) in let BTS: Set(Snapshot)= PreFS1→reject(s:Snapshot | CSPre→includes(s)) in self.requested=true implies BTS→forAll(s:Snapshot | s.light.pedLight=#Red)
TP3	Precedence-Globally	context TrafficLight inv TP3: self.pedLight=#Green implies sometimePast self.requested=true	context TrafficLight inv TP3: let CS: Snapshot = self.getCurrentSnapshot() in let PS: Set(Snapshot) = CS.getPre() in self.pedLight=#Green implies PS → exists (s:Snapshot | s.light.requested=true)
TP4	Absence-Between Q and R	context TrafficLight inv TP4: self.requested=true implies always not self.pedLight=#Green until self.carLight=#Red	context TrafficLight inv TP4: let CS: Snapshot = self.getCurrentSnapshot() in let FSR1 :Snapshot = CS.getPost()→ select(sr:Snapshot | sr.light.carLight=#Red)→asOrderedSet()→first() in let PreFSR1: Set(Snapshot)= FSR1.getPre() in let CSPre: Set(Snapshot)= CS.getPre()→including(CS) in let BTS: Set(Snapshot)= PreFSR1→reject(s:Snapshot | CSPre→includes(s)) in self.requested=true implies BTS→collect(s:Snapshot | s.light.pedLight=#Green) →isEmpty()
TP5	Universality-After Q until R	context TrafficLight inv TP5:self.pedLight=#Red implies always self.carLight=#Green until self.requested=true	context TrafficLight inv TP5: let CS: Snapshot = self.getCurrentSnapshot() in let FS : Set(Snapshot) = CS.getPost() in let FSR1 :Snapshot = CS.getPost()→ select(sr:Snapshot | sr.light.requested=true)→asOrderedSet()→first() in let PreFSR1: Set(Snapshot)= FSR1.getPre() in let CSPre: Set(Snapshot)= CS.getPre()→including(CS) in let BTS: Set(Snapshot)= PreFSR1→reject(s:Snapshot | CSPre→includes(s)) in ((self.pedLight=#Red and FSR1.isDefined) implies BTS→forAll(s:Snapshot | s.light.carLight=#Green) or ((self.pedLight=#Red and FSR1.oclIsUndefined()) implies FS→forAll(s:Snapshot | s.light.carLight=#Green))
TP6	Universality-Globally	context TrafficLight inv TP6: always not (self.pedLigh=#Green and self.carLight=#Green)	context TrafficLight inv TP6:Snapshot.allInstances→forAll(s:Snapshot | not (s.pedLigh=#Green and s.carLight=#Green)
TP7	Response-Before R	context TrafficLight inv TP7: self.carLight=#Green implies sometime self.carLight=#Red since self.requested=true	context TrafficLight inv TP7: let CS: Snapshot = self.getCurrentSnapshot() in let BS: Set(Snapshot) = CS.getPre() let PS: Set(Snapshot) = BS→select(ps:Snapshot | ps.light.requested=true) in let SS: Set(Snapshot) = BS→select(ss:Snapshot | ss.light.carLight=#Red) in self.carLight=#Green implies PS→forAll(ps:Snapshot | ps.getPost()→exists(ss:Snapshot | SS→includes(ss)))
TP8	Response- Between Q and R	context TrafficLight inv TP8: self.pedLight=#Red implies (self.requested=true implies sometime self.carLight=#Red) before self.carLight=#Green	context TrafficLight inv TP8: let CS: Snapshot = self.getCurrentSnapshot() in let FS : Set(Snapshot) = CS.getPost() in let FSR1 :Snapshot = FS→select(sr:Snapshot | sr.light.carLight=#Green) →asOrderedSet()→first() in let PreFS1: Set(Snapshot)= FSR1.getPre() in let PS: Snapshot = FS→any(ps:Snapshot| ps.light.requested=true) in let SS: Snapshot = FS→any(ss:Snapshot | ss.light.carLight=#Red) in self.pedLight=#Red implies (PS.getPost()→includes(SS) and FSR1.getPre()→includes(PS)and FSR1.getPre()→includes(SS))

small search-scope and search-depth can take significant time and memory. For example, creating and analyzing a state space for a system of 15 classes and 20 operations (e.g., [67]) with search-scope of five and search-depth of ten takes over two hours. The analysis task needs to create all



Invariant	Result
PedestrianButton_requestPass::definedobject	true
PedestrianButton_requestPass::inv91	true
PedestrianButton_requestPass::inv92	true
PedestrianButton_requestPass::sameobjectID	true
Snapshot::AcyclicScenario	true
Snapshot::OneScenario	true
Snapshot::SameSnapshot	true
TrafficLight::OCLTemporalProperty	true
TrafficLight::TP1	true
TrafficLight::TP2	true
TrafficLight::TP3	true
TrafficLight::TP4	true
TrafficLight::TP5	true
TrafficLight::TP6	true
TrafficLight::TP7	true
TrafficLight::TP8	true
TrafficLight_switchCarLight::definedobject	true
TrafficLight_switchCarLight::inv93	true
TrafficLight_switchCarLight::sameobjectID	true
TrafficLight_switchPedLight::definedobject	true
TrafficLight_switchPedLight::inv94	true
TrafficLight_switchPedLight::inv95	true
TrafficLight_switchPedLight::inv96	true
TrafficLight_switchPedLight::sameobjectID	true
Transition::SameTrans	true

Constraints ok. (1ms) 100%

Figure 3.15: A new graphical representation of the STM of the traffic light system. This figure is similar to Figure 3.8 but it has been augmented with temporal properties TP1 to TP8, as specified in Table 3.8. Appendix A provides the complete USE textual model of the traffic light system.

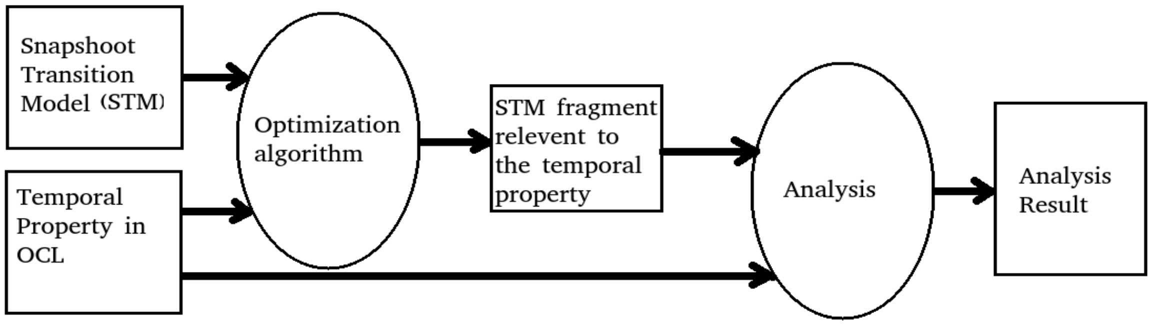

Snapshoot Transition Model (STM)
Temporal Property in OCL
Optimization algorithm
STM fragment relevent to the temporal property
Analysis
Analysis Result

Figure 3.16: Overview of the optimization technique, taken from our previous work Al-Lail et al. [96]

possible configurations of at least 11 snapshots of at least five objects of the 15 classes. Further, the task needs to create the object configurations conforming to all the invariants and constraints. This requires additional time. The second reason is that in complex systems subtle design faults may not reveal themselves in small scope executions. Therefore, searching bigger state space is necessary.

Recall that the fourth objective of this research is to propose a technique that scales the analysis to large UML class models in order to alleviate the state explosion problem. This section describes an optimization technique that can be used to scale the analysis.

Figure 3.16 depicts an overview of the optimization technique. The inputs of the optimization algorithm are an STM and an OCL expression representing a TOCL temporal property. The technique uses the temporal property as criterion to produce a fragment of the STM model that only contains the STM model elements that are relevant to the analysis of the property. If the STM fragment has fewer elements than the original STM, the analysis can handle larger analysis scopes and depths taking less time and memory. Algorithm 2 presents a systematic way to produce STM fragments based on the temporal property being analyzed. In collaboration with the author of the original algorithm Sun et al. [65], we extended it to accomedate temporal properties queries, Al-Lail et al. [96]. Algorithm 2 proceeds as described in the following steps:

1. The algorithm computes *DRMElmts* in lines 4-13, the set of STM model elements that are directly referenced by the OCL property, by traversing the abstract syntax tree of the OCL

temporal property. This set includes a set of classes, attributes, enumerations, references, and Snapshot Traversal Query Operations.

2. The algorithm then computes *IRMElmts* in lines 14-23, the set of STM model elements that are indirectly referenced by the temporal property. This set contains the subclasses of the classes that are directly referenced by the property.

3. The set of all related model elements, denoted *RMElmts* in line 24, then is the union of the directly related and indirectly related model elements.

4. The algorithm then checks if any model invariants are relevant to the analysis of the OCL property. An invariant is relevant to the property if the invariant references at least one STM model element that is related to the property. As such, the algorithm computes *tmpRMElmts* and checks if its intersection with *RMElmts* is not empty (lines 25-33).

5. Line 34 adds the set of elements of relevant invariants *INVRMElmts* to the set of all relevant elements of the OCL property *RMElmts*. The generated STM fragment contains only elements that are in *RMElmts* and the relevant invariants *INVList* Line 35.

To illustrate the optimization algorithm, consider the STM of the traffic light system in Figure. 3.17 and the following OCL representation of a temporal property defined on the STM.

```
context TrafficLight inv Pedestrians_Can_Crooss_Street:
let NextSnapshot:Snapshot = self.snp.getNext() in self.requested = true
implies NextSnapshot.TrafficLight.pedLight= # Green
```

Following the steps in the algorithm, the temporal property is defined in the context of class *TrafficLight*, and thus directly depends on the class *TrafficLight*. The expression *self.snp* is an association call and it returns a snapshot associated with the TrafficLight class (referred to by self). Thus, there is a direct dependency with the reference *snp* and its type class, *Snapshot*. The expression *snp.getNext()* is an operation call expression and it returns a snapshot. There is, thus, a direct dependency with the operation *getNext()*, and its type class, *Snapshot*. In addition, the *getNext()*

Algorithm 2 Extract STM fragment

```
1:  Input: A STM, a list of invariants, INVList, and a temporal property in OCL, TP
2:  Output: A $STM$ fragment, and a subset of INVList that are relevant to the analysis of TP
3:  Algorithm Steps:
4:  Analyze the dependencies between STM and TP, set DRMElmts = a set of model elements that
    are directly referenced by TP;
5:  Set QOpers = a set of query operations from DRMElmts;
6:  for QOpers not empty do
7:      Set QOperDRMElmts = $\{\}$;
8:      for each element, QOper, in QOpers do
9:          QOperDRMElmts = QOperDRMElmts $\cup$ a set of model elements directly referenced by
            QOper;
10:     end for
11:     DRMElmts = DRMElmts $\cup$ QOperDRMElmts;
12:     Set QOpers = a set of query operations from DRMElmts - QOpers;
13: end for
14: Set IRMElmts = $\{\}$;
15: for each element, ME, in DRMElmts do
16:     if ME is a class then
17:         if ME has subclasses then
18:             for each subclass, subME, of ME do
19:                 IRMElmts = IRMElmts $\cup$ subME;
20:             end for
21:         end if
22:     end if
23: end for
24: Set RMElmts = DRMElmts $\cup$ IRMElmts;
25: Set INVRMElmts = $\{\}$
26: for each invariant, INV, in INVList do
27:     Set tmpRMElmts = a set of model elements that are directly and indirectly referenced by
        INV;
28:     if tmpRMElmts $\cap$ RMElmts returns $\emptyset$ then
29:         Remove INV from INVList
30:     else
31:         INVRMElmts = INVRMElmts $\cup$ tmpRMElmts
32:     end if
33: end for
34: RMElmts = RMElmts $\cup$ INVRMElmts
35: Return RMElmts and INVList;
```

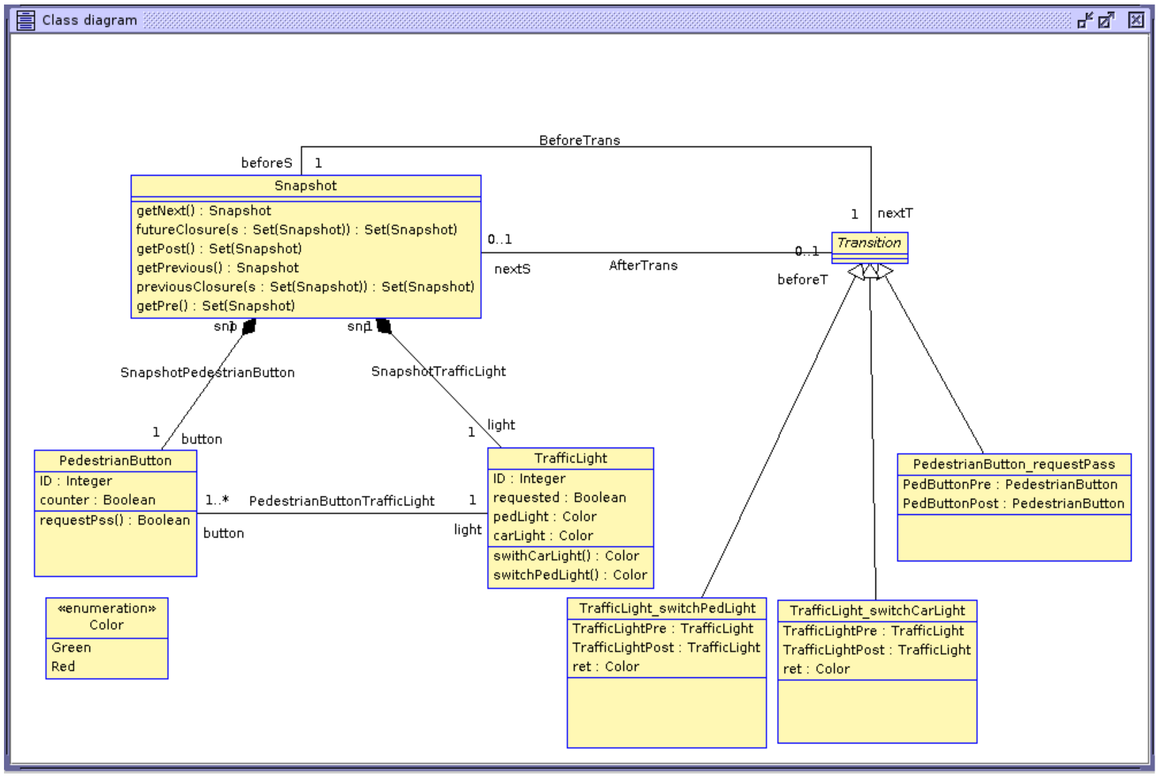


Figure 3.17: The STM of the traffic light system

operation is defined as: *self.nextT.nextS* (Listing 3.1), where *self.nextT* is an association call that returns a transition associated with a snapshot, and thus the temporal property directly depends on the reference *nextT*, and its type class Transition. The expression *self.requested* refers to an attribute defined in class *TrafficLight*, and thus property directly depends on attribute *requested*. Similarly, the expression *NextSnapshot.TrafficLight.pedLight =# Green* refers to an attribute defined in class *TrafficLight*, and thus the property directly depends on the attribute pedLight. Because the attribute pedLight has an enumeration type, *Color*, the property thus directly depends on *Color*. The application of the algorithm reveals that temporal property is directly related to the following STM elements: *DRMElmts* = {*TrafficLight, Snapshot, Transition, snp, nextT, nextS, getNext(), pedLight, requested, Color*}.

Because class Transition has subclasses, the temporal property indirectly references the three subclasses of the class Transition; therefore they belong to the set of the indirectly related model element (*IRMElmts* = { *PedestrianButton_requestPass,* TrafficLight_switchCarLight, *TrafficLight _switchPedLight*}).

The algorithm does a similar analysis for all the invariants that are defined in the STM model in Table 3.2 and checks if the set of invariant-related model elements intersects with the set of the temporal property related model elements. In this case all the STM invariants are related to the property. In particular, the invariants **AcyclicScenario**, **OneScenario**, **SameTrans**, and **SameSnapshot** are directly related to the temporal property as they have Snapshot and Transaction as contexts that are elements of *DRMElmts* set defined above. Similarly, all invariants named **definedobject** and **sameObjectID** defined on the context of subclasses PedestrianButton_requestPass, TrafficLight_switchCarLight, and TrafficLight_switchPedLight are indirectly related to the temporal property. These subclasses are elements of *IRMElmts* and they are all included in the STM fragment.

3.5 Chapter Summary

This chapter described the proposed UML framework for specifying and analyzing temporal properties. Specifically, the following three techniques were discussed:

1. A UML-based analysis technique that exclusively uses UML notations and tools. The technique is lightweight as it neither uses sophisticated mathematical notations nor comprehensively searches the entire search space. The search space is restricted by search-scopes and search-depths. In addition, the analysis technique only uses UML notations to specify and analyze temporal properties. Further, a UML-based tool (USE Model Validator) is employed to find countersexamples that are also presented in two UML notations: object diagrams representing a scenario that violates a property and a sequence diagram highlighting the scenario operations calls. The analysis technique is not dependent on the USE Model Validator. Provided the right tool, the analysis could be performed by more powerful tools such as Satisfiability Modulo Theories (SMT) solvers.

2. A property specification technique that improves the process of specifying temporal properties for UML designers. The technique uses property specification patterns that ease the effort required to specify properties and provides UML-based language representations of the patterns. To achieve this objective, the patterns of Dwyer et al. are defined in two UML-related languages: TOCL to specify temporal properties on an $ADCM$ and an OCL interpretation of TOCL on an STM that is used for analysis. Furthermore, interpretation rules, used to automatically generate OCL expressions, are defined for each pattern.

3. An optimization technique that reduces the time needed for analysis, allowing the analysis to be scaled to larger UML models. The technique employs an algorithm, which given a temporal property, reduces a class model to the model elements that are relevant to the analysis of the property. Removing irrelevant elements reduces the search space to be explored by the analysis tool and provides a way for alleviating the state explosion problem.

We evaluate the techniques in the following chapter.

Chapter 4

Evaluation

As outlined in Chapter 1, this dissertation aims to achieve the following six objectives:

1. To explore the state-of-the-art techniques and tools in order to identify research gaps and challenges in the field of Model Checking UML models.

2. To develop a UML-based analysis technique that exclusively uses UML notations and tools.

3. To streamline the process of specifying temporal properties for UML designers by developing a specification technique that uses UML notations.

4. To develop an optimization technique that reduces the time needed for analysis, allowing the analysis to be scaled to larger UML models.

5. To provide a proof-of-concept tool by implementing the specification, analysis, and optimization techniques.

6. To evaluate the framework through an actual software specification and analysis projects.

In Chapter 2, we conducted a Systematic Literature Review (SLR) that addressed the first objective. Chapter 3 discussed the details of the UML framework consisting of the analysis, the specification, and the optimization techniques that aimed to achieve the second, third, and fourth objectives of the dissertation, respectively. This chapter aims to address the fifth and the sixth objectives of the dissertation. It also provides the details of our Generalized Spatio-Temporal Role-Based Access Control Model(GSTRBAC) model [67], as it is one of the contributions of our research.

We developed a proof-of-concept research prototype to investigate the effectiveness of the techniques. The effectiveness of the three techniques is determined as follows:

– The analysis technique is judged based on the ability of the technique to find design faults.

– The specification technique is judged based on its ability to specify temporal properties of different kinds.

– The optimization technique is judged based on how much time is is saved as compared with the time needed for analysis without optimization.

Using the proof-of-concept tool, we evaluated the three techniques by applying them to the specification and the analysis of the following case studies:

1. The Generalized Spatio-Temporal Role-Based Access Control Model(GSTRBAC). The development of GSTRBAC focused on addressing the many application requirements of wireless and mobile devices that make use of the spatio-temporal information of a user to provide better functionality. Such applications necessitate authorization models where access to a resource depends on the credentials of the user and also on the location and time of access. The analysis of GSTRBAC is published in our previous work Al-Lail et al. [73], and the formal specification, validation, and enforcement of the model is published in Abdunabi et al. [67].

2. The Steam Boiler Control System (SBCS) specification problem Abrial et al. [68]. The SBCS specification problem has been used to assess the effectiveness of many software specification and verification approaches such as the Spin model checker, Petri Nets, and the PVS automated theorem prover. Using this case study, therefore, provided a benchmark that can be used to compare our framework with other methods. The analysis of the SBCS system is published in our previous work Al-Lail et al. [95].

The evaluation was done on a laptop computer with 2.2 GHz Intel(R) Core(TM) i7-3632QM CPU, 8 GB RAM and Windows 8. As we will show in this chapter, the results of the studies show that all the temporal properties of the two systems can be expressed by the specification technique, and that the analysis technique is capable of uncovering design faults. Moreover, the optimization technique can significantly reduce the time needed for analyzing large models.

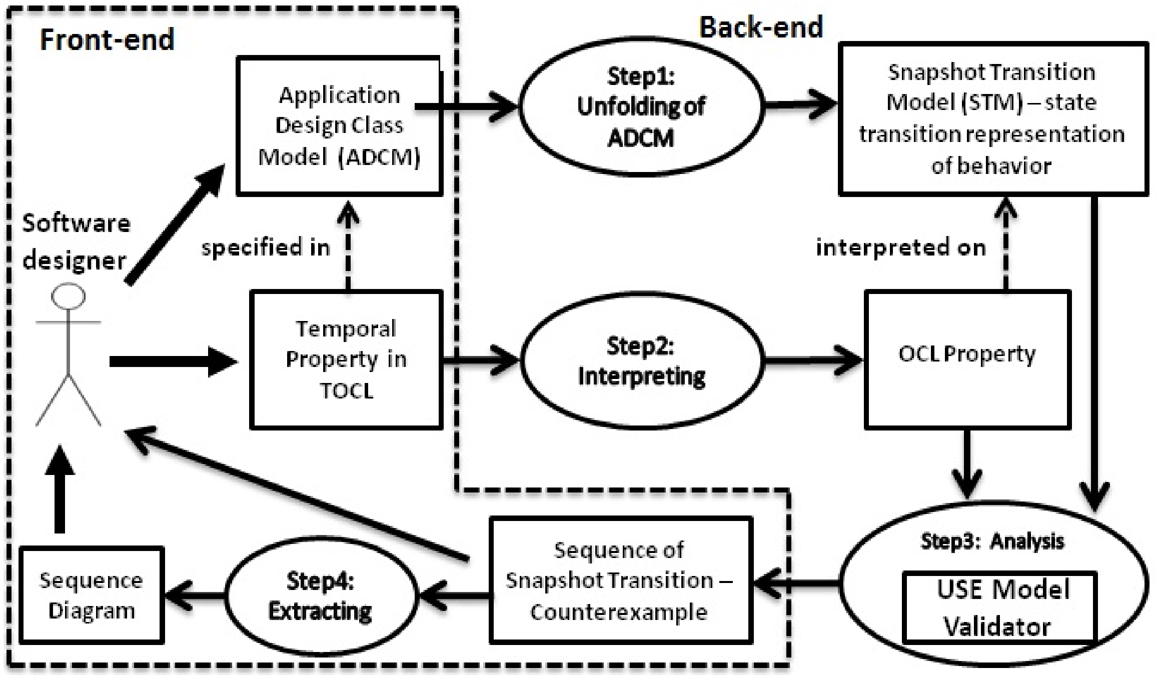

Front-end
Back-end
Application Design Class Model (ADCM)
Step1: Unfolding of ADCM
Snapshot Transition Model (STM) – state transition representation of behavior
Software designer
specified in
interpreted on
Temporal Property in TOCL
Step2: Interpreting
OCL Property
Step3: Analysis
USE Model Validator
Sequence of Snapshot Transition – Counterexample
Step4: Extracting
Sequence Diagram

Figure 4.1: An Overview of the Analysis Approach, extended from our previous work (Al-Lail et al. [73])

This chapter is organized as follows. We describe our prototype tool and its use in Section 4.2. Section 4.2 provides the details of the GSTRBAC model and Section 4.3 shows how our analysis and the specification techniques were used to obtain the final GSTRBAC model. In Section 4.4, we provide the results of the SBCS case study and in Section 4.5, we summarize and discuss the results of the evaluation .

4.1 Research Prototype Tool

Figure. 4.2 depicts the architecture of our proof-of-concept tool that provides implementation of the steps of our analysis technique, depicted in Figure 4.1. The tool, named the Temporal Analysis of UML Class Models (TAUCM), is composed of five implementation packages:

- Class Model to STM: This package generates STM models from design class models. Further, it includes an Eclipse plugin that transforms OCL operation specifications to invariants of the STM. This package implements the first step of the analysis techniques, as shown in Figure 4.1. .

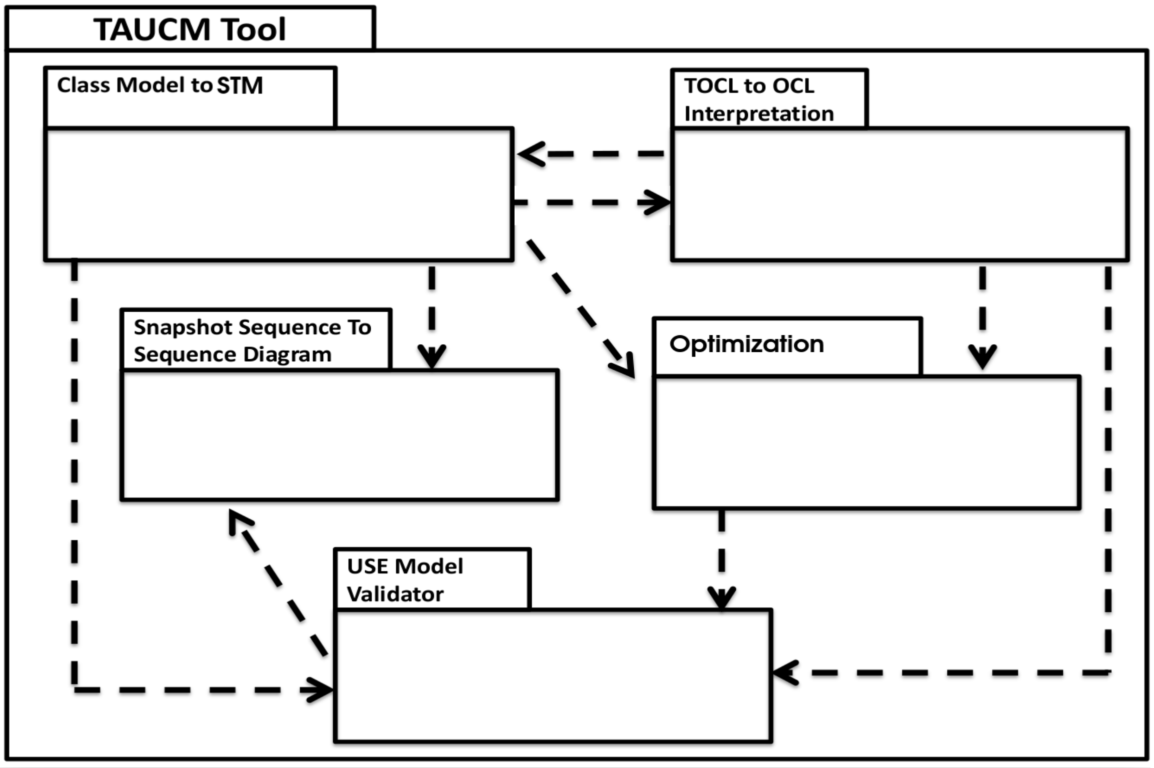

TAUCM Tool
Class Model to STM
TOCL to OCL Interpretation
Snapshot Sequence To Sequence Diagram
Optimization
USE Model Validator

Figure 4.2: Tool Architecture

- TOCL to OCL Interpretation: This package implements the interpretation rules that generate OCL expressions from TOCL properties. In addition, it has a plugin that transforms STM models and their OCL expressions to USE specifications. This package provides partial implementation of the second step of the analysis techniques, as shown in Figure 4.1.

- Optimization: This package has the implementation of the optimization technique.

- USE Model Validator: This package is used for the anlaysis step. The package is out-of-the-shelf tool developed by the Database Systems Group at Bremen University [60].

- Snapshot Sequence To Sequence Diagram: This package produces a sequence diagram representation of the output produced by USE Model Validator.

A substantial portion of the `Class Model to STM` package was implemented by Kayle Hoehn and Wuliang Sun for the Scenario-Based Analysis Technique, which is the main contribution of Lijun Yu's dissertation [97]. Yu's dissertation had a totally different objective from ours.

Nevertheless, Yu's work on generating STMs is applicable to our objectives. In our research, we extended the implementation to add the necessary code to address temporal properties. Specifically, we augmented the implementation by adding code for creating the Snapshot Traversal Query Operations and code for adding invariants to ensure the creation of well-formed STMs, the detains of which are discussed in Chapter 3.

For the `TOCL to OCL Interpretation` package, we developed a partial implementation of the specification technique and its interpretation rules. Specifically, we only implemented the widely used patterns, namely the Response and the Universality patterns. However, these patterns cover 80% of properties used in industry according to a study done by Dwyer et al. [66]. Therefore, we believe this partial implementation is adequate for our proof-of-concept tool.

The `Optimization` package was initially implemented as part of Wuliang Sun's slicing technique [98]. Sun's approach checks invariants of large class models by slicing the model to submodels that are relevant to the invariants. In a joint work with Sun (i.e., Al-Lail et al. [96]), we modified the algorithm and the code to accommodate temporal properties. Specifically, we incorporated our version of the STMs' generation process, and the partial implementation of the TOCL interpretation to generate OCL constraint representations of temporal properties.

The tool was built upon a number of existing technologies. In particular, we used Java language, Eclipse Modeling Framework (EMF) [99], and the Eclipse OCL API. EMF is used to create, manipulate and validate UML models. The Eclipse OCL API is an implementation of the OCL OMG standard for EMF-based models.

Figure 4.3, Figure 4.4, and Figure 4.5 provide visualizations of the different inputs of the TAUCM tool. The inputs of the tool are (1) an EMF Ecore file: that describes a UML design class model, i.e., ADCM, (2) an OCL file: a textual file that contains the ADCM's OCL invariants and operations specifications, (3) a TOCL file: a textual file that has an TOCL specification of a temporal property (4) a properties file: textual file that defines: a search_scope, a value K that represents a search_depth of the analysis, and ranges of values to be assigned to numerical parameters and at-

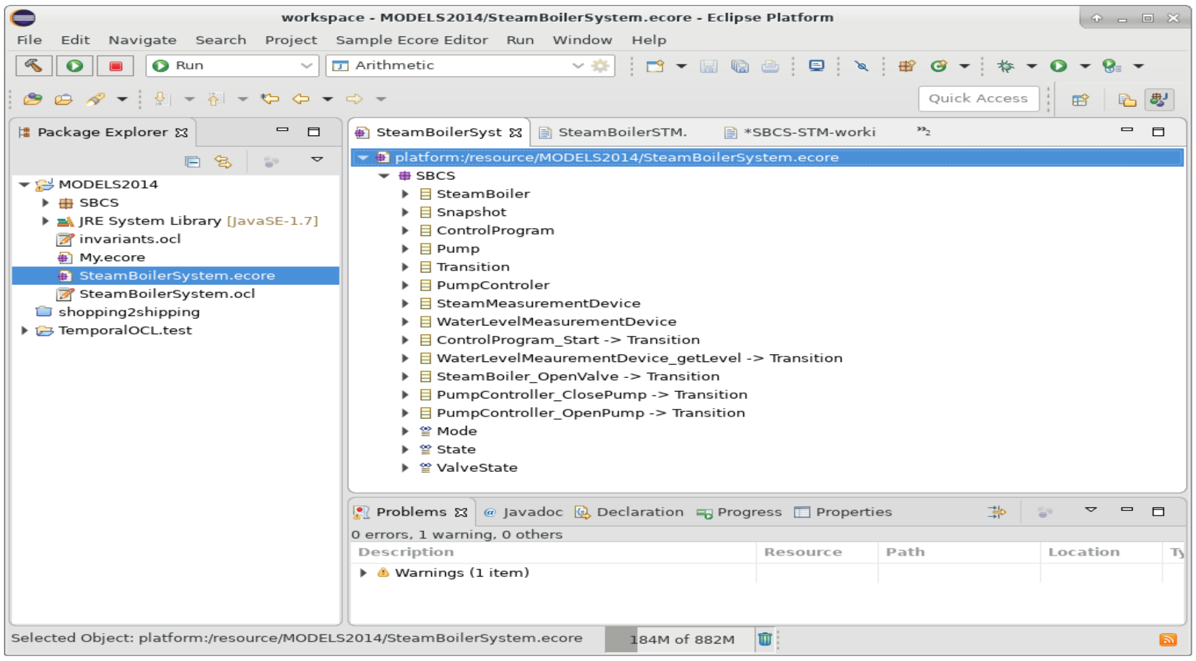
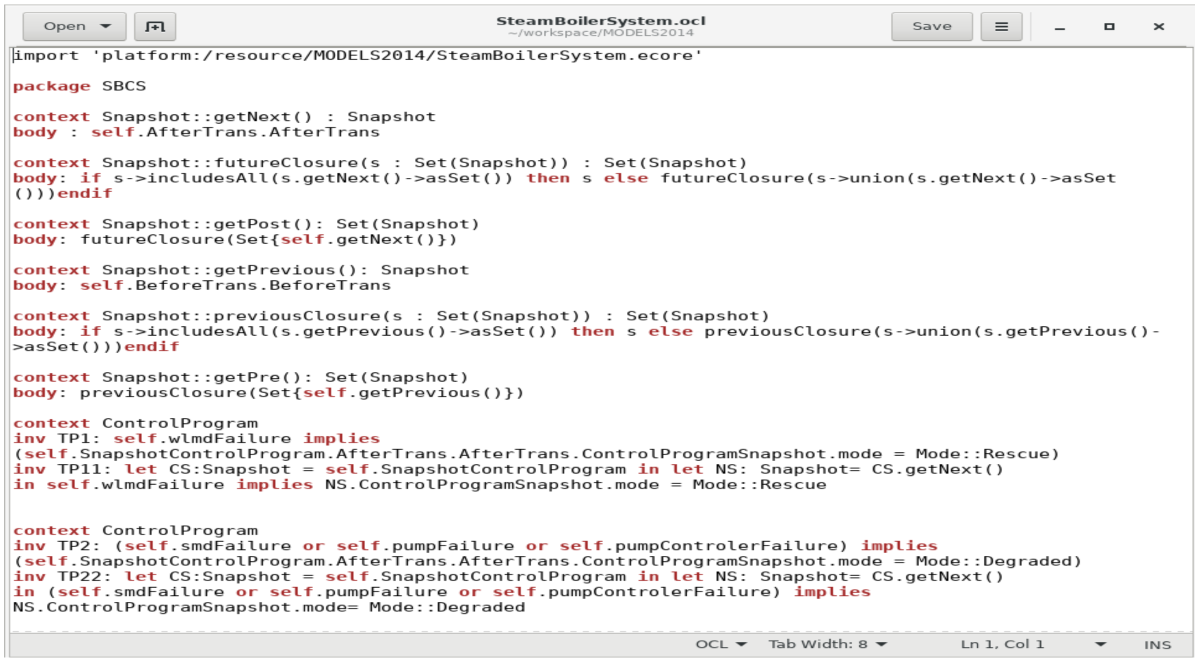

workspace - MODELS2014/SteamBoilerSystem.ecore - Eclipse Platform
File Edit Navigate Search Project Sample Ecore Editor Run Window Help
Run
Arithmetic
Quick Access
Package Explorer
MODELS2014
SBCS
JRE System Library [JavaSE-1.7]
invariants.ocl
My.ecore
SteamBoilerSystem.ecore
SteamBoilerSystem.ocl
shopping2shipping
TemporalOCL.test
SteamBoilerSyst
SteamBoilerSTM.
*SBCS-STM-worki
platform:/resource/MODELS2014/SteamBoilerSystem.ecore
SBCS
SteamBoiler
Snapshot
ControlProgram
Pump
Transition
PumpControler
SteamMeasurementDevice
WaterLevelMeasurementDevice
ControlProgram_Start -> Transition
WaterLevelMeaurementDevice_getLevel -> Transition
SteamBoiler_OpenValve -> Transition
PumpController_ClosePump -> Transition
PumpController_OpenPump -> Transition
Mode
State
ValveState
Problems
Javadoc
Declaration
Progress
Properties
0 errors, 1 warning, 0 others
Description
Resource
Path
Location
Warnings (1 item)
Selected Object: platform:/resource/MODELS2014/SteamBoilerSystem.ecore
184M of 882M

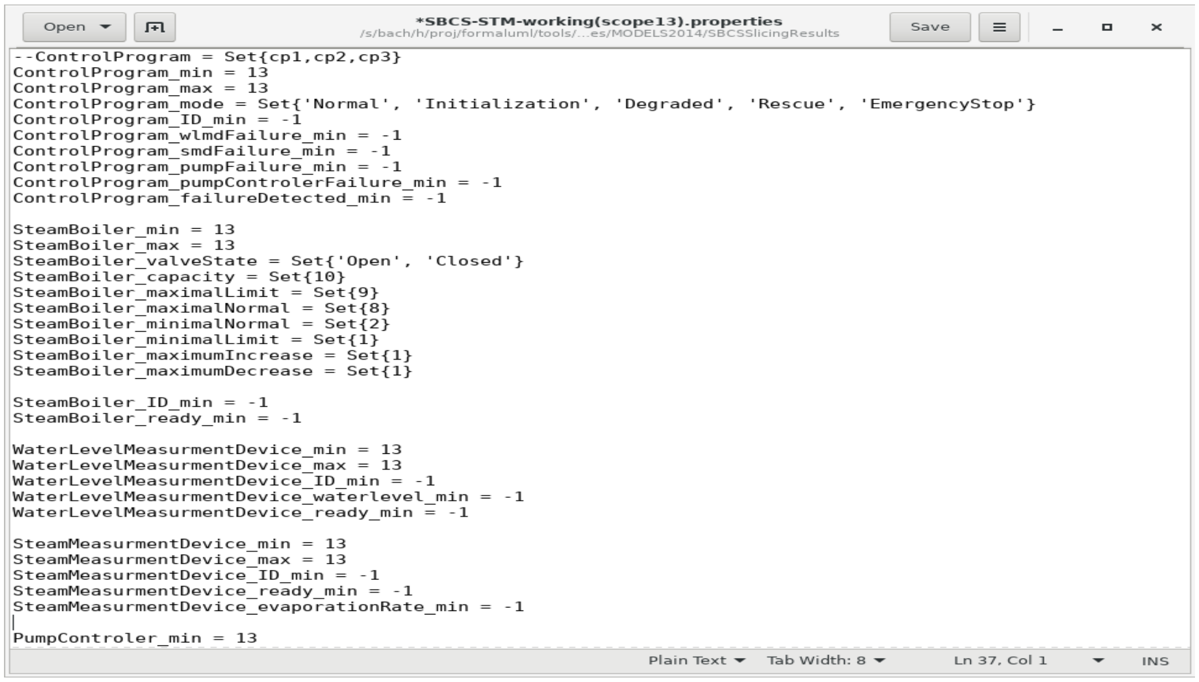
Figure 4.3: An example of TAUCM tool's input 1: An Eclipse ecore model describing a UML design class model.

tributes to constraint the search space (e.g., $0 \leq \text{Age} \leq 100$). Note that after the TOCL expression is interpreted as OCL constraint, it gets added to the OCL file, as shown in Figure 4.4.



```
import 'platform:/resource/MODELS2014/SteamBoilerSystem.ecore'

package SBCS

context Snapshot::getNext() : Snapshot
body : self.AfterTrans.AfterTrans

context Snapshot::futureClosure(s : Set(Snapshot)) : Set(Snapshot)
body: if s->includesAll(s.getNext()->asSet()) then s else futureClosure(s->union(s.getNext()->asSet
()))endif

context Snapshot::getPost(): Set(Snapshot)
body: futureClosure(Set{self.getNext()})

context Snapshot::getPrevious(): Snapshot
body: self.BeforeTrans.BeforeTrans

context Snapshot::previousClosure(s : Set(Snapshot)) : Set(Snapshot)
body: if s->includesAll(s.getPrevious()->asSet()) then s else previousClosure(s->union(s.getPrevious()-
>asSet()))endif

context Snapshot::getPre(): Set(Snapshot)
body: previousClosure(Set{self.getPrevious()})

context ControlProgram
inv TP1: self.wlmdFailure implies
(self.SnapshotControlProgram.AfterTrans.AfterTrans.ControlProgramSnapshot.mode = Mode::Rescue)
inv TP11: let CS:Snapshot = self.SnapshotControlProgram in let NS: Snapshot= CS.getNext()
in self.wlmdFailure implies NS.ControlProgramSnapshot.mode = Mode::Rescue


context ControlProgram
inv TP2: (self.smdFailure or self.pumpFailure or self.pumpControlerFailure) implies
(self.SnapshotControlProgram.AfterTrans.AfterTrans.ControlProgramSnapshot.mode = Mode::Degraded)
inv TP22: let CS:Snapshot = self.SnapshotControlProgram in let NS: Snapshot= CS.getNext()
in (self.smdFailure or self.pumpFailure or self.pumpControlerFailure) implies
NS.ControlProgramSnapshot.mode= Mode::Degraded
```

Figure 4.4: An example of TAUCM tool's input 2: A textual file, called OCL file that contains OCL invariants and operation specification and an OCL representation of a TOCL property.


Open
*SBCS-STM-working(scope13).properties
/s/bach/h/proj/formaluml/tools/...es/MODELS2014/SBCSSlicingResults
Save
Plain Text
Tab Width: 8
Ln 37, Col 1
INS

```
--ControlProgram = Set{cp1,cp2,cp3}
ControlProgram_min = 13
ControlProgram_max = 13
ControlProgram_mode = Set{'Normal', 'Initialization', 'Degraded', 'Rescue', 'EmergencyStop'}
ControlProgram_ID_min = -1
ControlProgram_wlmdFailure_min = -1
ControlProgram_smdFailure_min = -1
ControlProgram_pumpFailure_min = -1
ControlProgram_pumpControlerFailure_min = -1
ControlProgram_failureDetected_min = -1

SteamBoiler_min = 13
SteamBoiler_max = 13
SteamBoiler_valveState = Set{'Open', 'Closed'}
SteamBoiler_capacity = Set{10}
SteamBoiler_maximalLimit = Set{9}
SteamBoiler_maximalNormal = Set{8}
SteamBoiler_minimalNormal = Set{2}
SteamBoiler_minimalLimit = Set{1}
SteamBoiler_maximumIncrease = Set{1}
SteamBoiler_maximumDecrease = Set{1}

SteamBoiler_ID_min = -1
SteamBoiler_ready_min = -1

WaterLevelMeasurmentDevice_min = 13
WaterLevelMeasurmentDevice_max = 13
WaterLevelMeasurmentDevice_ID_min = -1
WaterLevelMeasurmentDevice_waterlevel_min = -1
WaterLevelMeasurmentDevice_ready_min = -1

SteamMeasurmentDevice_min = 13
SteamMeasurmentDevice_max = 13
SteamMeasurmentDevice_ID_min = -1
SteamMeasurmentDevice_ready_min = -1
SteamMeasurmentDevice_evaporationRate_min = -1

PumpControler_min = 13
```

Figure 4.5: An example of TAUCM tool's input 3: A textual file, called properties file, which defines scopes and a depth of the search.

A UML designer uses the tool as follows. The designer creates an Ecore ACDM using `Ecore Model Editor`. The OCL operations specifications are created using a text editor and saved as OCL.ocl (i.e., Figure 4.4). The user also specifies a temporal property in TOCL and saves it in a file with tocl extension. The Ecore ACDM is transformed to an Ecore STM using the `Class Model to STM` package. Furthermore, using the same package, the OCL operation specifications are transformed to invariants of the Ecore STM. The `TOCL to OCL Interpretation` package generates an OCL expression from the TOCL property and includes it in the OCL file. If optimization is required, the designer, at this stage, selects the option for the optimized model to be generated. The Ecore file and the OCL file are then fed into the `Optimization` package and the results of the optimization is saved as another Ecore and OCL file representing the optimized model. The `Ecore to USE`, a subpackage in the TOCL to OCL Interpretation not shown in the figure, reads the optimized Ecore STM, and the corresponding OCL file containing the temporal property, and transforms the files to an USE specification. The `USE Model Validator` loads the USE specification, and the textual file that defines the search parameters. The designer then runs the analysis tasks and waits for the output. In case that the property is violated, the prototype produces a counterexample, otherwise, the prototype indicates that no instance exists that violates the property within given search-scope and search-depth.

4.2 The Formal Specification of Generalized Spatio-Temporal Role-Based Access Control Model

4.2.1 Overview the GSTRBAC Model

In collaboration with Ramadan Abdunabi, we used an earlier version of the framework to specify and analyze a novel access control model [67, 73]. The development of GSTRBAC focused on addressing the many application requirements of wireless and mobile devices that make use of the spatio-temporal information of a user to provide better access control functionality. GSTRBAC is defined on top of the role-based access control RBAC model [100] to support access control for mobile applications. RBAC is the *de-facto* access control model used in the commercial sector. RBAC is policy-neutral and can be used to express different types of policies to simplify security management.

Mobile applications typically have new authorization requirements where environmental conditions, such as location and time, are used together with the credentials of the user to determine access. For example, a mobile application policy may be that a user should not use his mobile device to terminate the home motion-detector system from arbitrary out-of-home locations and after midnight. Such services should be terminated at the time that a user leaves his smart home. GSTRBAC can be used to specify this policy.

Figure 4.6 shows the UML class model of GSTRBAC. As we will show in Section 4.3, our specification and analysis techniques were employed when designing the GSTRBAC model. The model has 19 classes, 25 associations, 13 invariants, 37 operations, and 71 pre- and postconditions. Appendix C provides the complete UML specification of the model in the USE textual representation.

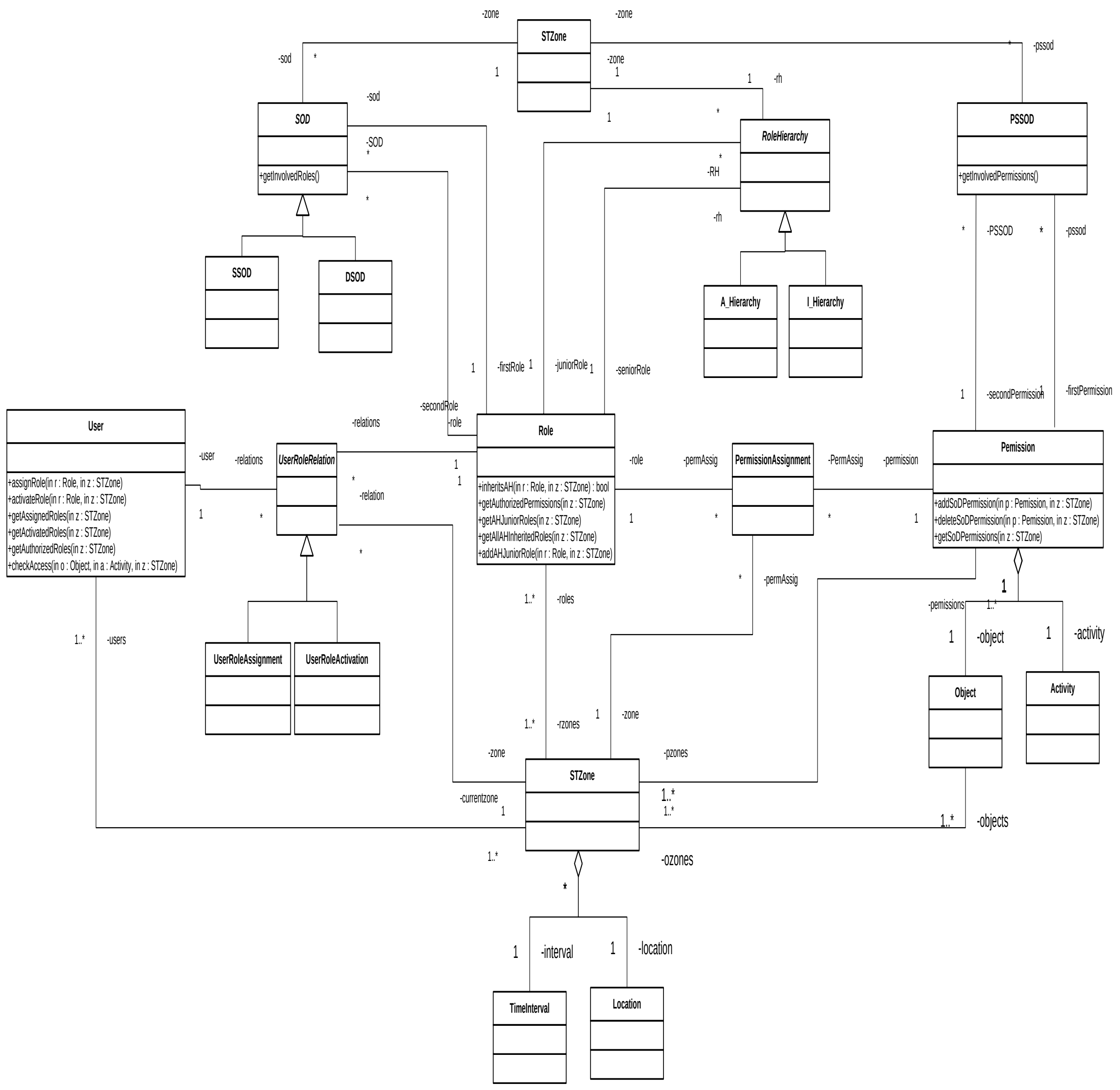

STZone
-zone
-sod
SOD
+getInvolvedRoles()
SSOD
DSOD
RoleHierarchy
-RH
-rh
A_Hierarchy
I_Hierarchy
PSSOD
+getInvolvedPermissions()
-pssod
-PSSOD
User
+assignRole(in r : Role, in z : STZone)
+activateRole(in r : Role, in z : STZone)
+getAssignedRoles(in z : STZone)
+getActivatedRoles(in z : STZone)
+getAuthorizedRoles(in z : STZone)
+checkAccess(in o : Object, in a : Activity, in z : STZone)
UserRoleRelation
UserRoleAssignment
UserRoleActivation
Role
+inheritsAH(in r : Role, in z : STZone) : bool
+getAuthorizedPermissions(in z : STZone)
+getAHJuniorRoles(in z : STZone)
+getAllAHInheritedRoles(in z : STZone)
+addAHJuniorRole(in r : Role, in z : STZone)
PermissionAssignment
Pemission
+addSoDPermission(in p : Pemission, in z : STZone)
+deleteSoDPermission(in p : Pemission, in z : STZone)
+getSoDPermissions(in z : STZone)
-firstRole
-juniorRole
-seniorRole
-secondRole
-secondPermission
-firstPermission
-relations
-user
-relation
-role
-permAssig
-PermAssig
-permission
-roles
-rzones
-zone
-pzones
-permissions
-object
-activity
Object
Activity
-currentzone
-users
-objects
-ozones
-interval
-location
TimeInterval
Location

Figure 4.6: UML Class Model for GSTRBAC, taken from our previous work (i.e., Abdunabi et al. [67]).

The core component of the GSTRBAC model is a spatio-temporal zone that is referred to as STZone. STZone is a logical and abstract entity encapsulating the particularities of location and time. The STZone class is associated with RBAC entities (e.g., User, Role, and Permission) and relationships (e.g., UserRoleRelation, PermissionAssignment, RoleHierarchy, and Seperation of Duties (SOD)), in order to define where and when these entities are accessible.

The primary GSTRBAC sets of Users, Roles, Permissions, and STZones, are represented by concrete classes. GSTRBAC relationships (e.g., UserRoleRelation, PermissionAssignment, Role-

Hierarchy, and SOD) are represented by association classes that have been transformed to normal classes in Figure 4.6, following the modeling guidelines in Booch et al. [18] and [101]. For example, the class PermissionAssignment in Figure 4.6 represents the assignment of a role to permission at particular STZone.

The model supports spatio-temporal constraints on user-role assignment, permission-role assignments, user-role activation, role hierarchy (RH), and separation of duties (SOD). Further, in GSTRBAC, a user might move from a valid zone to an invalid zone after access is authorized. Therefore, the system should revoke access when a user leaves the valid spatio-temporal zone. The GSTRBAC objects might interact, in a subtle way, by invoking operations and exchanging messages leading to violation of some of the model constraints. In such scenarios a user may access some roles from undesirable zones, or a mobile user suddenly moves to an invalid zone while exercising some rights. Therefore, it is important to ensure the soundness of the GSTRBAC model by doing analysis and ensuring that there does not exist a scenario of operations that lead to the violation of any of the predefined model constraints.

4.2.2 Location and Time Representation

As outlined above, each GSTRBAC set (e.g., Users, Roles, Permissions, and STZones) and relation (e.g., user-role assignment, permission-role assignment, and role hierarchy) is associated with spatio-temporal information. Before describing these sets and relations in details, we show how spatio-temporal information is represented in the GSTRBAC model.

Location Representation

There are two types of locations: *physical* and *logical*. All users and objects are associated with locations that correspond to the physical world. These are referred to as the physical locations. A physical location is formally defined by a set of points in a three-dimensional geometric space. A *physical location* $PLoc_i$ is a non-empty set of points $\{p_i, p_j, \ldots, p_n\}$ where a point p_k is represented by three co-ordinates. The granularity of each coordinate is dependent upon the application.

Physical locations are grouped into symbolic representations that will be used by applications. We refer to these symbolic representations as logical locations. Examples of logical locations are Fort Collins, Colorado etc. A *logical location* is an abstract notion for one or more physical locations. We assume the existence of a mapping function m that converts a logical location to a physical one.

[Mapping Function m] m is a total function that converts a logical location into a physical one. Formally, $m : L \rightarrow P$, where P is the set of all possible physical locations and L is the set of all logical locations.

Different kinds of operations can be performed on location data. We define two binary operators, namely, *containment* $\subseteq$, and *equality* $=$. A physical location $ploc_j$ is said to be *contained in* another physical location $ploc_k$, denoted as, $ploc_j \subseteq ploc_k$, if the following condition holds: $\forall p_i \in ploc_j, p_i \in ploc_k$. The location $ploc_j$ is called the contained location and $ploc_k$ is referred to as the containing or the enclosing location. Intuitively, a physical location $ploc_j$ is contained in another physical location $ploc_k$, if all points in $ploc_j$ also belong to $ploc_k$. Two physical locations $ploc_r$ and $ploc_s$ are *equal* if $ploc_r \subseteq ploc_s$ and $ploc_s \subseteq ploc_r$. Note that these operators are defined on physical locations. Thus, logical locations must be transformed into physical locations (using mapping function m defined above) before we can apply these operators. We define a logical location called *anywhere* that contains all other locations. Each application can describe logical locations at different granularity levels. For example, some permissions may be applicable on the entire state whereas other permissions are only applicable to people in the city.

Time Representation

The GSTRBAC model uses two kinds of temporal information. The first is known as time instant and the other is time interval. A *time instant* is one discrete point on the time line. The exact granularity of a time instant is application dependent. For instance, in some application a time instant may be measured at the nanosecond level and in another one it may be specified at the millisecond level. A *time interval* is a set of time instants. We use the notation $t_i \in d$ to mean that t_i is a time instant in the time interval d. We also define operators containment $\subseteq$ and equality $=$

for operating on time intervals. A time interval d_j is said to be *contained in* another time interval d_k, denoted as, $d_j \subseteq d_k$, if the following condition holds: $\forall t_i \in d_j, t_i \in d_k$. The interval d_j is called the contained interval and d_k is referred to as the containing or the enclosing interval. Two time intervals d_s and d_r are said to be equal if $d_r \subseteq d_s$ and $d_s \subseteq d_r$. We define a time interval called *always* that includes all other time intervals. Each application should be able to express different types of temporal intervals.

Spatio-Temporal Zone

A spatio-temporal zone abstracts location and time representations into a single entity.

Spatio-temporal zone depends of the formalization of a spatio-temporal point.

[Spatio-Temporal Point] A spatio-temporal point is a pair of the form (d, l) where d and l represent time interval and location respectively. An example of a spatio-temporal point is: (Home Office, [6 p.m. - 8 a.m.]).

[Spatio-Temporal Zone] A spatio-temporal zone is a set of spatio-temporal points.

An example of a spatio-temporal zone is: $\{$(HomeOffice, [6 p.m. - 8 a.m.]), (DeptOffice, [8 a.m. - 6 p.m.])$\}$.

[Spatio-Temporal Zone Containment] A spatio-temporal zone P is contained in another spatio-temporal zone Q, denoted by $P \subseteq Q$, if for every spatio-temporal point $(d', l') \in P$, there exists a spatio-temporal point $(d, l) \in Q$, such that $(d', l') \subseteq (d, l)$.

4.2.3 Effect of Spatio-Temporal Constraints on RBAC Entities

Each RBAC entity, namely, users, roles, permissions, and objects are associated with spatio-temporal zones.

Users

We assume that each valid user carries a locating device that is able to track his location. The location of a user changes with time. The spatio-temporal zone associated with a user gives the user's current location and time. The user's current location and time information will be used for making access decisions. Consequently, we require the minimal temporal and location interval be used to express the spatio-temporal zone associated with a user. We define a function $currentzone$ that returns the minimal spatio-temporal zone associated with user u.

Objects

Objects may also be mobile like the user. Hence, we have locating devices that track the location of an object. Moreover, an object may not be accessible everywhere. $ozones$ represent the spatio-temporal zone where the object is available.

Roles

Role can be assigned or activated only in specific locations and time. The role of on-campus student can only be assigned/activated inside the campus during the semester. The spatio-temporal zone associated with a role gives the location and time from which roles can be assigned or activated. The function $rzones$ gives the set of spatio-temporal zones associated with a given role.

Permissions

Permissions are also associated with a spatio-temporal zone that indicate where and when a permission can be invoked. For example, a permission to perform backup of servers can be executed only from the department after 10 p.m. on Friday nights. The function $pzones$ gives the zones from which a specific permission can be activated.

4.2.4 Effect of Spatio-Temporal Constraints on RBAC Operations

In this section we discuss the effect of the time and location on the different RBAC relations, namely, user role assignment, user role activation, and permission role assignment.

User Role Assignment

A user role assignment is location and time dependent. That is, a user can be assigned to a role only if the user is in specific locations. For example, a person can be assigned the on-campus student role only when he is in the campus during the semester. This relation is depicted in the GSTRBAC class model as the subclass *UserRoleAssignment* of the class *UserRoleRelation* (see Figure 4.6). The superclass *UserRoleRelation* is linked to *STZone* class to specify spatio-temporal constraints on the user role assignment relationship. The assignment of user u to a role r in spatio-temporal zone z is done through the class *UserRoleAssignment*.

The following OCL operation describes the pre and post conditions of the user-role assignment operation, *assignRole*. The pre-conditions respectively specify that the role r is available in zone z, the current user u is in zone z, and the role r is not already assigned to user u in zone z. Note that, the association role *rzones* gives the set of zones where a role can be assigned or activated. The association role *currentzones* determines the current *STZone* of a user. The OCL query operation *containedZones()* in the *STZone* class returns the set of zones that are contained the current zone. The query operation *getAssignedRoles(z)* determines the set of roles assigned to user in *STZone* z. The post-condition asserts that a user-role assignment relationship instance has been created between user u and role r in zone z, which is reflected by the fact that the role is included in the set of the assigned roles in the STZone z. The complete specifications of all GSTRBAC operations and constraints are in Appendix C.

```
context User::assignRole(r: Role, z:STZone): UserRoleAssignment
pre:  r.rzones->includes(z)
pre:  z.containedZones()->includes(self.currentzone)
pre:  self.getAssignedRoles(z)->excludes(r)
post: self.getAssignedRoles(z)->includes(r)
```

User Role Activation

A user can activate a role if the role is available in the specific zone and it is already assigned to that user. For example, the role of doctor trainee can only be activated in a hospital during the training period. Role assignment and activation should only take place in a specific *STZone* set. In Figure 4.6 of the GSTRBAC class model, the binary association between the class Role and the class *STZone* defines the set of *STZone* of each role.

Similar to *assignRole* The *activateRole* operation in class *User* is specified in OCL as follows. The preconditions are added to the *activateRole* operation to spatio-temporally restrict the execution of the operation by users. The first precondition checks that user u is currently in zone z, the second one ensures that the role r is spatio-temporally available in zone z, the user u assignment to role r in zone z is checked in the third precondition, and the last one verifies that the role r is not already in active state by user u in zone z. The new OCL query operation *getAssignedRoles(z)* used here gives the set of activated roles in *STZone* z in the user context. The post condition ensures that the role is activated in the zone.

```
context User::activateRole(r: Role, z:STZone): UserRoleActivation
pre: z.containedZones()->includes(self.currentzones)
pre: r.rzones->includes(z)
pre: self.getAssignedRoles(z)->includes(r)
pre: self.getActivatedRoles(z)->excludes(r)
post: self.getActivatedRoles(z)->includes(r)
```

Check Access

This operation checks whether a user is authorized to perform some activity on an object during a certain time and when the user is in a certain location. For instance, a user is allowed to fire a missile only if he is assigned the role of top secret commander and he is in the controller room of the missile during a severe crisis period. For a user to perform activity a on an object o in zone z, there must be some role activated in z which has a permission p that can be invoked at z and the zone of the object is in z. The OCL specification is given below.

```
context User::checkAccess(o:Object,a: Activity,z:STZone):Boolean
post: result = getActivatedRoles(z)-> collect( r | r.getAuthorizedPermissions(z))->asSet()
-> exists( p | p.object=o and p.activity=a and o.ozones->includes(z))
```

Permission Role Assignment

Some permissions may be assigned to a role during specific time and locations. For example, the permission of opening a cashier drawer in a store should be only assigned to a salesman role during the day time. A permission p can be assigned to a role in zone z only if z is in the allowedzones for both the permission and the role. The OCL specification is given below.

```
context Role::assignPermission(p:Permission, z:STZone): PermissionAssignment
pre:   p.pzones->includes(z) and self.rzones-> includes(z)
pre:   self.getAssignedPermissions(z)->excludes(p)
post:  self.getAssignedPermissions(z)->includes(p)
```

4.2.5 Spatio-Temporal Role Hierarchy

The structure of an organization in terms of lines of authority can be modeled as a hierarchy. This organization structure is reflected in RBAC in the form of a role hierarchy [100]. Role hierarchy is a transitive and anti-symmetric relation among roles. Roles higher up in the hierarchy are referred to as senior roles and those lower down are junior roles. The major motivation for adding role hierarchy to RBAC was to simplify role management. Senior roles can inherit the permissions of junior roles, or a senior role can activate a junior role, or do both depending on the nature of the hierarchy. This obviates the need for separately assigning the same permissions to all members belonging to a hierarchy. Joshi et al. [102] identify two basic types of hierarchy. The first is the permission inheritance hierarchy where a senior role inherits the permission of junior roles. The second is the role activation hierarchy where a user assigned to a senior role can activate junior roles.

In the GSTRBAC class model (Figure 4.6), permission inheritance hierarchy and role activation hierarchy are represented by the classes *I-Hierarchy* and *A-Hierarchy*, respectively. These hierarchies are specializations of the abstract class *RoleHierarchy*. *RoleHierarchy* is associated with *STZone* that describe the zones location and time as to where and when these hierarchical relationships are enabled.

Permission Inheritance Hierarchy

In our GSTRBAC model, the permission inheritance hierarchy is associated with spatio-temporal constraints. For example, a project manager inherits the permissions of a developer when he is at the customer site giving a demo. In permission inheritance hierarchy, a senior role can be added to the junior role r in spatio-temporal zone z, if both roles are allowed in zone z and if r is not already a junior role in this hierarchy. The following OCL expression expresses the pre and post conditions that represent the spatio-temporal constraints for adding a new junior role.

```
context Role::addIHJuniorRole(r:Role,z:STZone): I_Hierarchy
pre: self.rzones->includes(z) and r.rzones-> includes(z)
pre:  self.getIHJuniorRoles(z)->excludes(r)
post: self.getIHJuniorRoles(z)->includes(r)
```

The delete operation of a junior role from permission-inheritance hierarchy in particular zone can be defined in the similar manner. The following OCL operation represent the deletion of permissions-inheritance hierarchy relation instance *I-Hierarchy* between senior role and junior role r in the *STZone* z.

```
context Role::deleteIHJuniorRole(r:Role, z:STZone)
pre: self.getIHJuniorRoles(z)->includes(r)
post: self.getIHJuniorRoles(z)->excludes(r)
```

The *I-Hierarchy* relationship must be acyclic as a junior role must not be able to inherit permissions from a senior role through the role hierarchy. The following OCL constraint specifies this condition.

```
context r1,r2: Role inv IHierarchy_Cycle_Constraint:
not  STZone.allInstances-> exists(z|r1.inheritsIH(r2,z) and r2.inheritsIH(r1,z)and r1<>r2)
```

The OCL boolean operation *inheritsIH(r,z)* returns true if a role is a junior role directly or indirectly of the context role in particular zone, otherwise it returns false. This boolean operation evaluates the inheritance relation between roles in question either directly or indirectly through multiple levels of the permissions-inheritance hierarchy. Note that the definition of this operation uses recursion to get the transitive closure of the inherit relation, as the OCL did not have definition of the transitive closure operator at the time this research was initially done.

```
inheritsIH(r:Role,z:STZone): Boolean =
if (self.getIHJuniorRoles(z)->includes(r)) then true
else self.getIHJuniorRoles(z)->exists(j | j.inheritsIH(r,z))
endif
```

We define the OCL query operation *getAuthorizedPermissions(z)* to get the authorized permissions for a given role at zone z. This includes the permissions directly assigned to the role (given by *getAssignedPermissions(z)*) and also the permissions inherited through role hierarchy. This OCL query operation returns a set of permissions that can be accessed by the context role at zone z.

```
context Role::getAuthorizedPermissions(z:STZone): Set(Permission)
Post: result= self.getAssignedPermissions(z)-> union(self.getAllIHInheritedRoles(z)->
collect(r | r.getAssignedPermissions(z)))->asSet()
```

The query *getAllIHInheritedRoles(z)* operation uses the *inheritsIH(z)* operation in order to get the set of all junior roles that are inherited directly or indirectly by context role through permissions-inheritance hierarchy in *STZone z*.

```
getAllIHInheritedRoles(z:STZone): Set(Role)=
Role.allInstances-> select(r | self.inheritsIH(r,z))-> asSet()
```

Role Activation Hierarchy

Restricted spatio-temporal role activation hierarchy allows members of senior roles to activate junior roles in predefined spatio-temporal zones only when both roles are spatio-temporally available. For example, department chair can activate staff during the semester inside the department building. The OCL operations of adding and deleting junior roles to the *A-Hierarchy* are defined in the same way of the *I-Hierarchy*. Further, the OCL query operation *getAHJuniorRoles(z)* returns all the junior role in *A-Hierarchy* of the context role in particular zone. The OCL addition operation of junior role *A-Hierarchy* is defined as following:

```
context Role::addAHJuniorRole(r:Role,z:STZone): A_Hierarchy
pre: self.allowedzones->includes(z) and r.allowedzones->includes(z)
pre:  self.getAHJuniorRoles(z)-> excludes(r)
post: self.getAHJuniorRoles(z)-> includes(r)
```

The OCL deletion operation of a junior role from *A-Hierarchy* is specified as following:

```
context Role::deleteAHJuniorRole(r:Role, z:STZone)
pre: self.getAHJuniorRoles(z) -> includes(r)
post: self.getAHJuniorRoles(z)-> excludes(r)
```

The cyclic constraints on the role activation hierarchy, *A-Hierarchy*, is defined in the same way as the permission inheritance hierarchy, *I-Hierarchy*.

```
context r1,r2: Role inv AHierarchy_Cycle_Constraint:
not STZone.allInstances-> exists(z| r1.inheritsAH(r2,z) and r2.inheritsAH(r1,z) and r1<>r2)

inheritsAH(r:Role,z:STZone): Boolean =
if (self.getHHJuniorRoles(z)->includes(r)) then true else self.getAHJuniorRoles(z)->
 exists(j | j.inheritsAH(r,z))
endif
```

The OCL query operation *getAuthorizedRoles(z)* is defined to get the authorized roles for the context user that are explicitly assigned, or implicitly obtained through, *A-Hierarchy* in the *STZone* z. The OCL query operation *getAllAHInheritedRoles(z)* defined in *getAuthorizedRoles(z)* returns a set of junior roles inherited by context role in all paths of the *A-Hierarchy* for the zone z. The result of this OCL query is a set of roles authorized to the user.

```
context User:: getAuthorizedRoles(z:STZone): Set(Role)
post: result= self.getAssignedRoles(z)-> union(self.getAssignedRoles(z)->
collect(r| r.getAllAHInheritedRoles(z))-> asSet())
```

4.2.6 Spatio-Temporal Separation of Duty

Separation of Duty (SOD) aims to prevent fraud and errors committed purposely or inadvertently by users. The main idea is to separate the responsibility of multiple individuals, such that no single user or role will have enough authority to commit a fraud. An example is that the same individual cannot be a member of purchasing-manger and accounts-manger roles because this creates possibility of fraud. The purchasing-manger and accounts-manger roles are called mutually exclusive roles or conflicting roles. This statement requires that exclusive roles should not be assigned to the same individual.

The same security principle can also be applied to permissions to provide additional assurance for separation of duties, in case of errors at higher level of mutually exclusive roles. The SoD between permissions constraint states that conflict permissions can not be assigned to the same role. For example, the permissions of preparing and approving purchase orders should not be assigned to the purchasing-manger role. Intuitively, this security principle limits the distribution of powerful permissions to roles.

Two classes of SoD constraints are formally defined in the classic RBAC known as static and dynamic SoD constraints, are respectively termed as SSoD and DSoD [103]. SSoD prevents the assignment of conflicting roles to the same user or assignment of conflicting permissions to the same role. DSoD prevents the simultaneous activation of conflicting roles.

Role SSoD

Sometimes an organizational security policy requires that the same individual should not be assigned to some roles in specific locations for some duration. For example, the same user should not be assigned to billing clerk and accounts receivable clerk roles in the same time at specific trade corporations. Therefore, we define the following OCL invariant to forbid the assignment of two conflicting roles in particular zones to the same user.

```
context User inv SSOD_Constaint:
STZone.allInstances->forAll( z | not self.getAssignedRoles(z)-> exists(r1,r2 |
r1.getSSoDRoles(z)->includes(r2)))
```

The SSOD constraint might be violated through role hierarchy relation. A senior role can inherit permissions from two junior roles that are conflicting by SoD. For example, the billing supervisor role inherits permissions of the billing clerk role when the billing supervisor acknowledges the correction of the bills issued by Billing Clerk, and Billing Clerk has SSoD relation with the accounts clerk in an accounting department, then the billing supervisor should also have SSoD relation with accounts clerk in the same department. This constraint on role hierarchy can be specified using the OCL expression as follows:

```
context User  inv SSOD_RH_Constraint:
STZone.allInstances->  forAll( z | not self.getAuthorizedRoles(z)->  exists(r1,r2 |
r1.getSSoDRoles(z)->includes(r2)))
```

The above OCL invariant restricts a user from having some conflicting roles through role hierarchy. The OCL query *getAuthorizedRoles(z)* determines all roles authorized by a user through role hierarchy, neither of these roles should conflict in zone z.

Permissions SSoD

Sometimes permissions are also related by SSoD relation; a role cannot acquire permissions that are conflicting. In our model, permission SSoD can also be constrained by spatio-temporal constraints. For example, a loan officer is not permitted to issue a loan request and approve it in the bank building during the day-time. The following OCL invariant expresses the PSSoD requirement:

```
context Role inv PSSOD_Constaint:
 STZone.allInstances-> forAll( z | not self.getAssignedPermissions(z)-> exists(p1,p2 |
 p1.getPSSoDPermissions(z)-> includes(p2)))
```

The above OCL invariant states that there are no permissions-role assignment relations for the same role and two different permissions in a specific zone, and these permissions have $PSSoD$ relation in that zone. The query *getPSSoDPermissions(z)* specifies the conflicting permissions in zone z based on *PSSoD* relation.

The above *PSSOD_Constraint* protects against the direct assignment of conflicting permissions to roles, however, it can be violated through permissions-inheritance hierarchy. Such a situation occurs if the senior role inherits some junior roles that have been mutually assigned conflicting permissions. Thus, we need to check that permissions authorized directly (e.g assignment) or indirectly (e.g hierarchy) for a role in some spatio-temporal zones do not have conflicting permissions. This constraint is specified in the context of the role using the following OCL invariant to prevent the violation of SSoD on permissions via role hierarchy.

```
 context Role  inv PSSOD_RH_Constraint:
 STZone.allInstances->  forAll( z | not self.getAuthorizedPermissions(z)->  exists(p1,p2 |
 p1.getPSSoDPermissions(z)-> includes(p2)))
```

The above OCL constraint states that no conflicting permissions are authorized to a role in a specific zone. The query *getAuthorizedPermissions(z)* returns all permissions authorized for a role via assignment or hierarchy relationships in *GSTRBAC*.

DSoD

Dynamic SoD is another form of mutual exclusion between roles considered at run-time. It states that the mutually exclusive roles cannot be activated simultaneously by the same individual. In our model, two conflicting roles cannot be activated in some spatio-temporal zone by the same user. For example, simultaneous activation of Cashier and Cashier Supervisor is forbidden during the working hours in the same store to deter such user from committing fraud. The following OCL invariant specify this constraint:

```
context User inv DSOD_Constaint:
 STZone.allInstances-> forAll( z | not self.getActivatedRoles(z)-> exists(r1,r2 |
 r1.getDSoDRoles(z)->includes(r2)))
```

The above invariant states that if two roles are conflicting activation roles related by $DSoD$ in specific zone, then no single individual can have two role activation relation in the same critical zone. The operation *getDSoDRoles(z)* returns the conflicting activation roles in a specific zone.

Note that this constraint restricts the explicit activation of conflicting roles in specific zone, however, this constraint can still be violated through implicit role activation in role-activation hierarchy. This might authorize a user to activate some conflicting junior roles. This constraint is specified in GSTRBAC model using OCL expressions as following:

```
 context User  inv DSOD_RH_Constraint:
 STZone.allInstances-> forAll( z | not self.getAuthorizedActiveRoles(z)-> exists(r1,r2 |
 r1.getDSoDRoles(z)->includes(r2)))
```

This OCL constraint asserts that roles activated by a user in specific zone have no conflict. The query operation *getAuthorizedActiveRoles(z)* returns the set of roles that can be activated by a user through role-activation hierarchy in zone z.

4.2.7 Spatio-Temporal Prerequisite Constraints

Some organizations require some prerequisite conditions to be satisfied before an operation such as user role assignment or user role activation can take place. This kind of constraint in RBAC is termed as *prerequisite constraints* [104]. For example, a user should be assigned role r_1 to be authorized for role r_2.

Prerequisite Constraints on User-Role Assignment

In our model, we augment the prerequisite constraints on user-role assignment by restricting the fact that the user must be assigned to some less critical role in a given spatio-temporal zone before being assigned a more critical role in the same zone. For example, the role of emergency-nurse can be assigned to John in the urgent care unit from 12:00 a.m. to 5:00 a.m. only if he is assigned the role of nurse-on-night-duty at the hospital during those hours.

The following OCL constraint is defined to restrict the creation of user-role assignment instances *UserRoleAssignment* on the condition of satisfaction of spatio-temporal prerequisite constraints. The OCL constraint states that for all zones if a user is assigned to a role *r1* in particular STZone *z*, this implies that the user has been assigned to all the prerequisite roles of *r1*. We use the query operation *getPrerequisiteRoles()* which returns all the prerequisite roles for the context role. The constraints prohibit the assignment of the critical role r_1 to the context user in zone *z* unless all the prerequisite roles are also assigned in the same zone.

```
context User inv Prerequiste_URAssign:
STZone.allInstances->forAll(z | Role.allInstances-> forAll(r1|(self.getAssignedRoles(z)->
includes(r1)) implies (self.getAssignedRoles(z)->includesAll(r1.getPrerequisiteRoles()))))
```

Prerequisite Constraints on Permission-Role Assignment

Sometimes prerequisite constraints are imposed on permission role assignment. A role can be assigned a permission provided it has other prerequisite permissions already assigned to it. In our model, some critical permissions can be assigned to roles in specific zones only if some prerequisite permissions are already assigned to the same role in the same zone. For example, a bank teller must have the permission of reading an account during working hours before he can be given the permission to update that account. The prerequisite constraint on permission-role assignment can be specified using OCL expression as follows.

```
context Role  inv Prerequist_PRAssign:
STZone.allInstances-> forAll(z | Permission.allInstances-> forAll(p1 |
(self.getAssignedPermissions(z)->   includes(p1)) implies
(self.getAssignedPermissions(z) -> includesAll(p1.getPrerequisitePermissions()))))
```

Prerequisite User-Role Activation

Sometimes roles can be activated only if some prerequisite role can be activated. For example, in a university the teaching assistant role can be activated during a semester in a department only if the student role can be activated during the same time.

The following OCL invariant restricts the generation of user-role activation for critical roles based on spatio-temporal prerequisite role activation constraint. This OCL invariant states that every time a user *u* tries to activate role *r1* in zone *z*, this constraints checks whether that user has already activated prerequisite roles in that zone. The query operation *getPrerequisiteRoles()* returns a set of prerequisite roles needed to activate role *r1*.

```
context User inv Prerequist_URActiv:
STZone.allInstances-> forAll(z | Role.allInstances-> forAll(r1 |
(self.getActivatedRoles(z)->   includes(r1)) implies
(self.getActivatedRoles(z)-> includesAll(r1.getPrerequisiteRoles()))))
```

4.3 Case Study 1: Specification and Analysis of Generalized Spatio-Temporal Role-Based Access Control Model

We used the specification and the analysis techniques to specify and analyze the GSTRBAC model that is discussed in previous sections. In addition, we followed an incremental approach as we designed the model. That is, we specified and analyzed partial models of the GSTRBAC model. During this process, we uncovered more than fifty design faults that we fixed before obtaining the final design.

In this section, we present an example of using the analysis technique to find a design fault in one of the partial models we created. Moreover, we present some of the temporal properties we defined using the specification technique. However, we did not evaluate the optimization technique using the GSTRBAC model as it had been designed before the incorporation of the technique in our framework.

4.3.1 Analyzing the GSTRBAC system

During the design phase of the GSTRBAC model, we used a partial model of GSTRBAC in which we only considered the parts of the model that are related to user-role interactions (see Figure 4.7). The property that we were interested in checking was the requirement that when a user changes a spatio-temporal zone, all the activated roles get deactivated in the next state. Such property is referred to as persistence checking of spatio-temporal constraints. To check the property, we used an earlier version of the analysis technique Al-Lail et al. [73]. As described in Chapter 3, the analysis technique has the following four steps:

1. Unfolding the behavior of the model as an STM.
2. Interpreting a TOCL property as OCL expression.
3. Analyzing the model.
4. Extracting a sequence diagram outcome of the counterexample.

We show how we applied the above steps to the specification and analysis of the above temporal property on the partial model.

Step1: Unfolding the Behavior

In this step, we converted the partial class model in Figure 4.7 to its corresponding STM. The STM is formed by: (1) creating the Snapshot class, (2) creating a hierarchy of transition classes representing operation invocation, (3) converting operation specifications to invariants of the transition classes, (4) defining the Snapshot Query Operations, and (5) adding invariants to ensure that the obtained STM is well-formed.

Figure 4.8 depicts the result of the unfolding of the partial GSTRBAC class diagram. The Snapshot class is a class that defines object configurations in a particular state. Note, as of April 2018, the USE Model Validator does not have support for structured classes, as discussed in Chapter 3. Consequently, we had to represent the $Snapshot$ class as a class that has composition relationships with the different classes that constitute the parts of the class $Snapshot$. This representation is similar to the Structured Classefier, as defined by the OMG UML standard [105]. As shown in Figure 4.8, the Snapshot class has composition relationships to User, STZones, Roles classes, and their two types or relations, RoleAssignment and RoleActivation classes. Note that the original partial model in Figure 4.7 has an inheritance hierarchy that is represented by the abstract

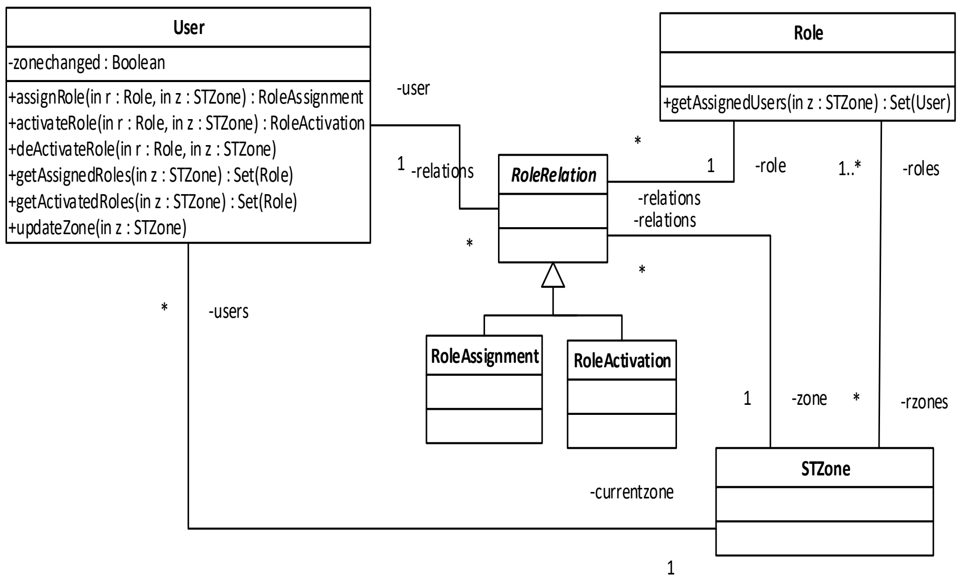

User
-zonechanged : Boolean
+assignRole(in r : Role, in z : STZone) : RoleAssignment
+activateRole(in r : Role, in z : STZone) : RoleActivation
+deActivateRole(in r : Role, in z : STZone)
+getAssignedRoles(in z : STZone) : Set(Role)
+getActivatedRoles(in z : STZone) : Set(Role)
+updateZone(in z : STZone)
Role
+getAssignedUsers(in z : STZone) : Set(User)
RoleRelation
RoleAssignment
RoleActivation
STZone
-user
-relations
-role
-roles
-users
-zone
-rzones
-currentzone

Figure 4.7: Partial GSTRBAC model, taken from our previous work Al-Lail et al. [73]

class RoleRelation. Following the guidelines in Chapter 3, abstract classes are not represented in $STMs$ as they do not have instances (objects) in the states of a system. In addition, the inheritance hierarchy is flattened first, and then the concrete classes (i.e., RoleAssignment and RoleActivation) are mapped as parts of the Snapshot class in the STM model (see Figure 4.8).

The second step of the unfolding step is creating a hierarchy of transition classes that represents the operations. In the partial GSTRBAC model, we considered all seven of the operations shown in Figure 4.7 as they relate to the persistence checking property. Please note that four of the operations have side-effects and three are query operations. In particular, the operations that have side-effects are assignRole(), updateZone(), activateRole(), and deActivateRole(). The query operations are getAssignedRoles(), getActivatedRoles(), and getAssignedUsers().

As described in Chapter 3, we only need to consider the operations with side-effects, namely, updateZone(), activateRole(), and deActivateRole(). Therefore, we defined these operations as subclasses of the abstract Transition class in order to create the hierarchy. For example, the operation assignRole(r: Role, z:STZone) in the user class is mapped to the transition subclass User_ AssignRole. The query operations are needed to define invariants and temporal properties. These

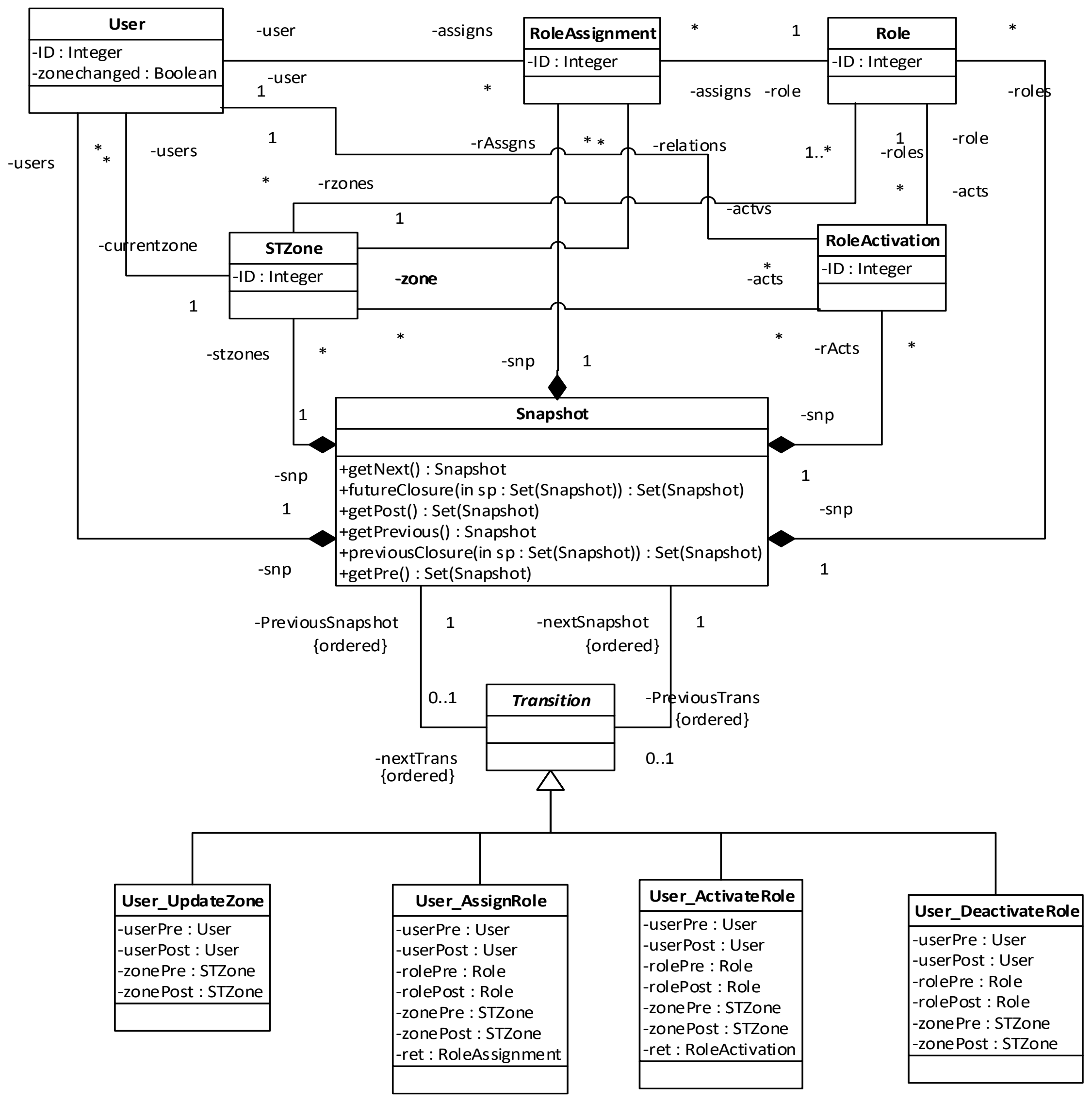


Figure 4.8: The STM model of the Partial GSTRBAC model in Figure 4.7, taken from our previous work Al-Lail et al. [73]

operations are still included in the STM model, but they are not shown in the Figure 4.8 to simplify the presentation. Further, no transition subclasses are created for the query operations.

The parameters of operations are mapped into attributes of the subclasses. If a parameter has a class type, it is mapped into two attributes. For example, the parameters r of assignRole(r: Role, z:STZone) are represented by $rolePre$, and $rolePost$, each of which specifies the parameters' state before and after the operation's call. In addition, two attributes ($userPre$ and $userPost$ of type User) are created for the object that the operation is invoked on. Because the assignRole(r:Role, z:STZone) has a return value of type RoleAssignment, a ret attribute is created to represent the return object.

The third step is concerned with converting the operations' pre- and postconditions into invariants defined in the context of the transition subclasses. The pre- and postconditions of the operation assignRole(r:Role, z:STZone) are mapped to invariants on the transition subclass User_AssignRole as follows.

```
context User::assignRole(r: Role, z:STZone): UserRoleAssignment
pre: self.currentzone->includes(z)
pre: self.getAssignedRoles(z)-> excludes(r)
post: self.getAssignedRoles(z)-> includes(r)
```

The above pre- and postconditions are converted to the following invariants.

```
context User_AssignRole
inv: userPre.currentzone->includes(zonePre)
inv: userPre.getAssignedRoles(zonePre)-> excludes(rolePre)
inv: userPost.getAssignedRoles(zonePost)-> includes(rolePost)
```

As described in Chapter 3, the above invariants ensure that the User_AssignRole transition subclass is only instantiated when the pre- and post-conditions of the assignRole() operation are satisfied. To elaborate, consider the first pre condition and its corresponding invariant. The precondition states that the current zone of the user before the call of the operation must include the zone where the role r is to be assigned. Recall from above that object in which the operation is called and the parameters r and z of the operation assignRole() are represented by two attributes of the same type in the STM model, namely $userPre$, $userPost$, $rolePre$, $rolePost$, $zonePre$, and $zonePost$, respectively. Consequently, these attributes are used when defining the invariant on the STM side. For the first invariant, the status of the user before the call ($userPre$) is used to define the constraint. The invariant ensures that the current zone of the user of the before call state includes the zone $zonePre$ where the role $rolePre$ is to be assigned. In our technique, we guarantee that objects in the before and after snapshot are the same by making sure they have the same ID. To achieve this, we define constraints, called frame constraints Gogolla et al. [106], that specify objects and links are not affected by the operation are the same in the before and after snapshots. The other invariants of the subclass User_AssignRole are defined in a similar manner.

These mapping rules are similarly applied on the other three operations that have side-effects, namely, updateZone(), activateRole(), and deActivateRole().

For Step 4 (i.e., defining the Snapshot Query Operations), the query operations are added to the STM, as shown in Figure 4.8. For Step 5 of the unfolding step (i.e., adding invariants to ensure that the obtained STM is well-formed), the STM constraints are added to the STM. In particular, the STM invariants in Table 3.2, on page 59, are directly created without any change.

Step2: Interpreting TOCL as OCL

We are interested in making sure that the partial GSTRBAC model does not violate the persistence checking requirement. In our framework, a software designer uses TOCL to specify such requirement. The property is specified in TOCL as follows:

```
context u:User inv Persistence_Check: u.zonechanged implies next
self.getActivatedRoles(currentzone)-> isEmpty()
```

The above TOCL property is specified on the class model in Figure 4.7. It states when a user changes his zone, then the set of activated roles in the new zone is empty. The following constraint is the corresponding OCL expression that is obtained by applying the interpretation rules from TOCL and OCL, as described in Chapter 3.

```
context u:User inv Persistence_Check:
let CS: Snapshot = u.getCurrentSnapshot(),
in let NS: Snapshot = CS.getNext() in u.zonechanged
implies NS.u.getActivatedRoles(u.currentzone)-> isEmpty()
```

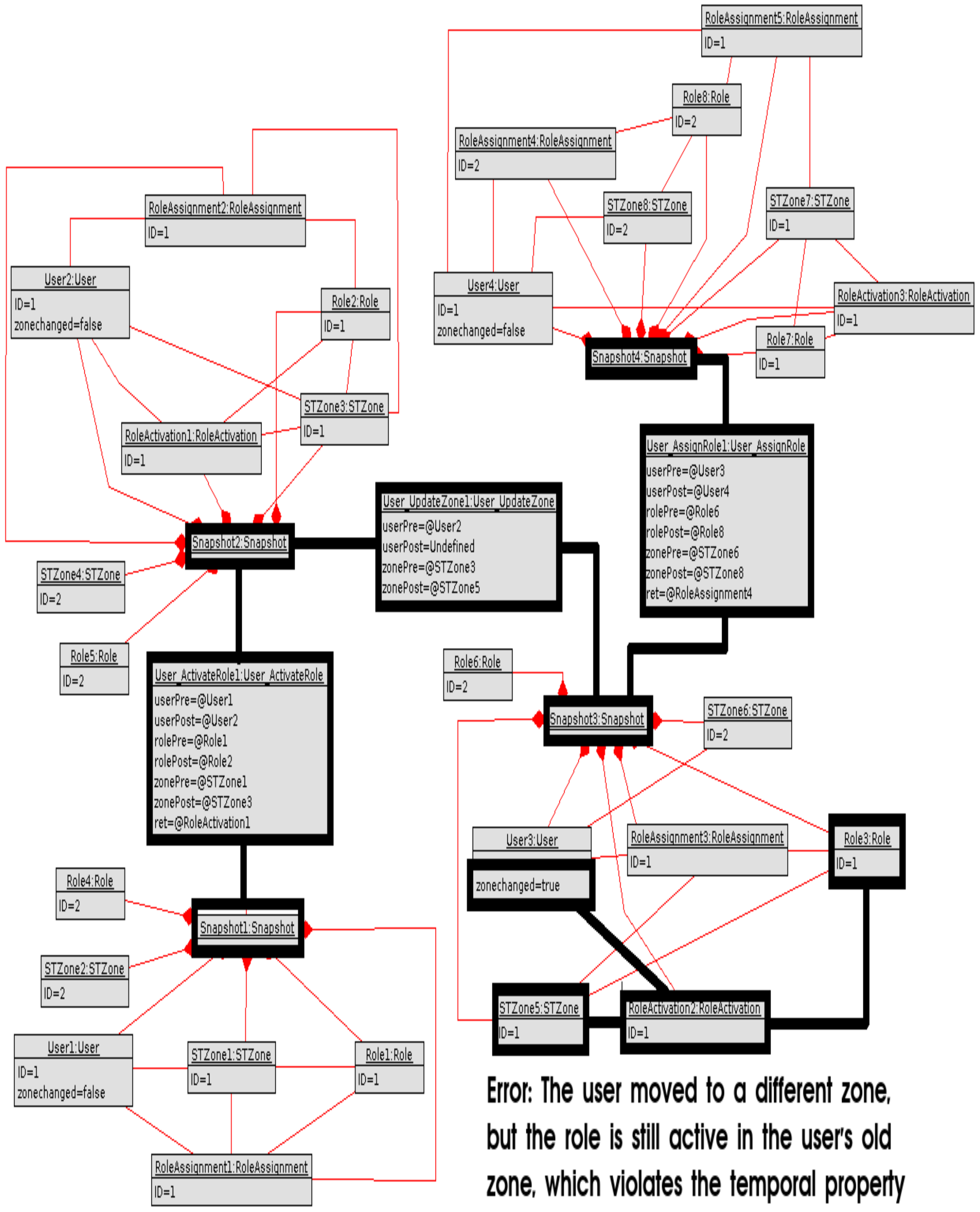


Figure 4.9: Counterexample: Scenario violating the Persistence-Check temporal property, taken from our previous work Al-Lail et al. [73]

Step 3: Analysis

At the back end of the prototype tool, the analysis task is performed by the USE Model Validator to check the persistence property. Given all the required inputs (i.e., the STM model, search scopes, search_depth, and the OCL property), the USE Model Validator tool attempts to produce a scenario that violates the property. Figure 4.9 shows the counterexample that leads to the temporal property violation.

As explained in Chapter 3, the analysis takes advantage of the small scope-hypothesis and the search-depth to constrain the search space and make the analysis feasible. We instructed the tool to produce a counterexample with search-depth of 3 and scopes of 1 user, 2 roles, and 2 zones in each snapshot. Because the search-depth (number of transitions) is 3 the counterexample has four snapshots as each transition connects two snapshots (see Figure 4.9). Snapshot1 shows that the User1 (ID=1) was assigned to Role1(ID=1) in STZone1(ID=1). In the next state (Snapshot2), the role got activated in the same zone through the invocation of the operation User_ActivateRole1. In Snapshot3, the user moved to a different zone (User_UpdateZone1 transition), (STZone6 ID=2). In addition, in Snapshot3, the user's zonechanged attribute was changed to true. The next state of the user with ID=1 should not have any roles activated. However, the role (Role with ID=1) was still active.

To uncover the fault, we debugged the counterexample. The design fault we found is that the UpdateZone() operation was missing a precondition that ensures that there is no activated roles before the operation is called. We added the needed precondition to address this design fault.

Step 4: Sequence diagram extraction

Figure 4.9 shows the counterexample which is an object diagram representing a sequence of snapshot transitions violating the temporal property. Such results might be complicated and difficult to debug in the case that the design class model is complex. Figure 4.10 shows the sequence diagram of the counterexample. As future work, we intend to future develop the use of the sequence diagram to facilitate uncovering of design faults. That is, a user could click on a message

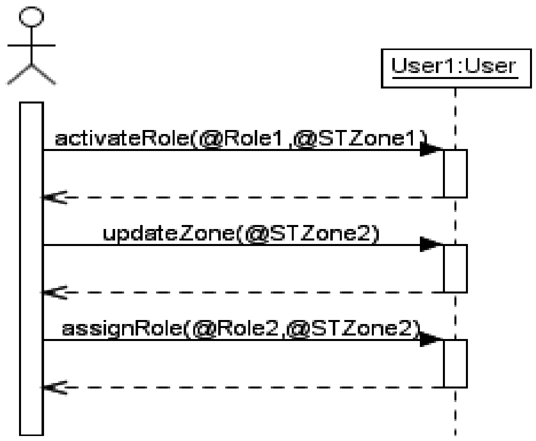


Figure 4.10: Sequence diagram counterexample, taken from our previous work Al-Lail et al. [73]

in a sequence diagram and could see how the state changed (the changed elements could be highlighted).

4.3.2 Specifying temporal properties of the GSTRBAC system

To specify temporal properties of a system in UML notations, a UML designer uses our specification technique to obtain the TOCL and OCL properties. As described in Chapter 3, a user of the technique follows the following steps to specify a property:

1. Determine the TOCL pattern and the scope that best fit a temporal requirement

2. Obtain the TOCL property (i.e., instance of the pattern) by replacing the parameters of the pattern by appropriate context and conditions

The interpretation rules are then employed to generate the OCL counterpart expression for the analysis purpose.

We have specified 29 GSTRBAC temporal properties using our TOCL and OCL patterns. In this section, we only describe two of these properties in Table 4.1 how the above steps are applied to formally specify them. Appendix C provides the description and specifications of the rest of properties.

Table 4.1: Two temporal properties of the GSTRBAC model

No.	Description	Pattern - Scope
GSTRBAC-TP1	If a role is available in a particular zone, the role should eventually be assigned to a user in that zone.	Response-Globally
GSTRBAC-TP2	When a user activates a role in a zone, the role remains active until the user moves to a different zone.	Universality-Between Q and R

Table 4.2: An Example of Using The Specification Technique to obtain an Instance of The Response Pattern for GSTRBAC-TP1, taken from our previous work Al-Lail et al. [73]

Scope	TOCL	OCL
Scope: Globally	**context** *[Class]* **inv:** *[P] implies sometime [S]*	**context** *[Class]* **inv:** *let CS: Snapshot = self.getCurrentSnapshot(),* in let FS: Set(Snapshot) = CS.$getPost()$ **in** *[P] implies FS → exists(s:Snapshot ∣ [s ⊨ S])*
Example: GSTRBAC-TP1	context r:Role inv GSTRBAC-TP1: r.getAvailableZones()→ includes(z:STZone) implies sometime r.getAssignedUsers(z) → notEmpty()	context r:Role inv GSTRBAC-TP1: let CS: Snapshot= self.getCurrentSnapshot() in let FS: Set(Snapshot)= CS.gePost() in r.getAvailableZones()→ includes(z:STZone) implies FS → exists (s:Snapshot ∣ s.r.getAssignedUsers(z) → notEmpty())

The first property (GSTRBAC-TP1) specifies the requirement that if a role is available in a particular zone, the role should be eventually assigned to a user in that zone. The description indicates a cause/effect relationship between two events, i.e., the availability of a role in a zone and the assignment of a role to a user in that zone. Therefore, the property is an instance of the response pattern in the global scope. After that, we used the corresponding TOCL pattern to obtain a TOCL property, from which the OCL expression is derived. For this we inserted the proper context and conditions of the designated places indicated by square brackets. Table 4.2 provides the TOCL and OCL expressions of GSTRBAC-TP1.

It is worth noting that the Persistence_Check property we specified and checked in Section 4.3.1 is a special case of the response pattern in which the effect occurs in the next state of the cause.

Table 4.3: An Example of Using The Specification Technique to obtain an Instance of The Universality Pattern for GSTRBAC-TP2, taken from our previous work Al-Lail et al. [73]

	TOCL	**OCL**
Scope:Between Q and R	context *[Class]* inv:*[Q]* implies always *[P]* until *[R]*	context *[Class]* inv: let CS: Snapshot = self.getCurrentSnapshot() in let FSR1: Snapshot= CS.getPost()→ select(s:Snapshot | *[s⊨ R]*)→ asOrderedSet()→ first() in let PreFSR1=Set(Snapshot) = FSR1.getPre(), in let CSPre: Set(Snapshot)= CS.getPre()→including(CS) in let BTS: Set(Snapshot)= PreFSR1→reject(s:Snapshot | CSPre→includes(s)) in *[Q]* implies BTS → forAll(s:Snapshot | *[s ⊨ P]*)
Example: GSTRBAC-TP2	context u:User inv GSTRBAC-TP2: u.getActivatedRoles(u.currentzone)→ includes(r:Role) implies always u.getActivatedRoles(u.currentzone) → includes(r) until u.zonechanged= True	context u:User inv GSTRBAC-TP2: let CS: Snapshot = self.getCurrentSnapshot() in let FSR1 :Snapshot = CS.getPost()→ select(sr:Snapshot | sr.u.zonechanged= True)→asOrderedSet()→first() in let PreFSR1: Set(Snapshot)= FSR1.getPre() in let CSPre: Set(Snapshot)= CS.getPre()→including(CS) in let BTS: Set(Snapshot)= PreFSR1→reject(s:Snapshot | CSPre→includes(s)) in u.getActivatedRoles(u.currentzone)→ includes(r:Role) implies BTS→forAll(s:Snapshot | s.u.getActivatedRoles(u.currentzone) → includes(r))

The second GSTRBAC property (GSTRBAC-TP2) captures the requirement that when a user activates a role in a zone then the role remains active until the user changes zone. Based on this description, we decided that the property is an instance of a more complex pattern (the universality pattern in between Q and R scope). Again, we defined the appropriate context and conditions to obtain the TOCL property base on the University pattern in between Q and R. The TOCL and OCL specifications of this property are given in Table 4.3.

4.4 Case Study 2: Specification and Analysis of The Steam Boiler Control System

The second case study is based on the Steam Boiler Control System (SBCS) specification problem Abrial et al. [68]. The SBCS problem has been used extensively to assess the effectiveness of many software specification and verification approaches. Ziemann and Gogolla [59] have defined

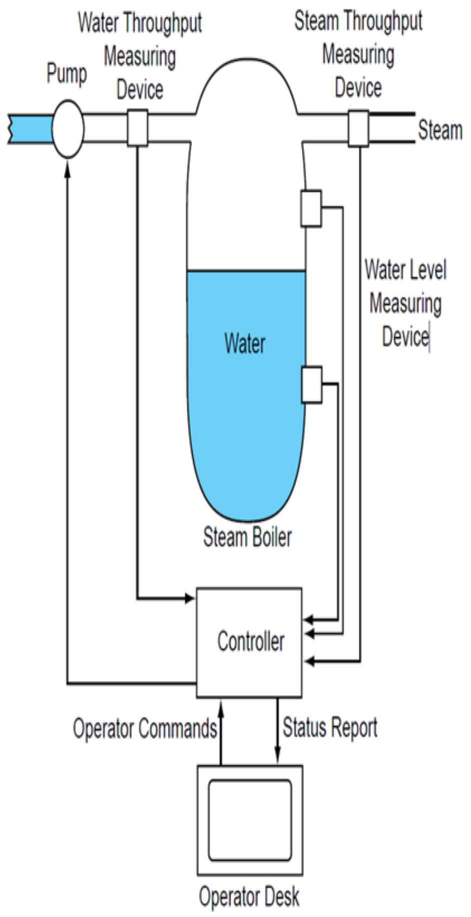

Water Throughput Measuring Device
Steam Throughput Measuring Device
Pump
Steam
Water Level Measuring Device
Water
Steam Boiler
Controller
Operator Commands
Status Report
Operator Desk

Figure 4.11: A Visualization of the Steam Boiler Control System

few temporal properties of the SBCS system, but to the best of our knowledge, we are the only group that has applied a UML approach to designing the SBCS system. Similar to the GSTRBAC case study, we followed an incremental approach as we designed the SBCS system. During the design process, we uncovered 16 design faults.

In this section, we first give an overview of the SBCS problem. Then, we present an example of using the analysis technique to find a design fault. Further, we present some of the temporal properties we defined using the specification technique. We also show the evaluation of the optimization technique using the SBCS system.

4.4.1 The Steam Boiler Control System Problem

Figure 4.11 depicts a simple visualization of the system. We provide an informal specification of the system's program that controls the level of water in the steam boiler. The system must work correctly and the water level must neither be too low, nor too high. The water level has to be within two normal limits (`minimalNormal` and `maximalNormal`) and can not pass over two critical limits (`minimalLimit` and `maximalLimit`); otherwise the steam boiler can be seriously damaged.

The system uses different devices to maintain the level of the water in a steam boiler. The physical system is composed of the following devices:

- The steam-boiler
- A device that measures the water in the steam-boiler
- A device that measures the quantity of steam coming out of the boiler
- Four pumps that supply the steam-boiler with water
- Four pump-controllers to supervise the pumps
- A valve that evacuates water from the steam-boiler
- An operator desk that manually stops the system

The system takes actions when any of the devices fail or when the water level exceeds one of the defined limits. To illustrate, when the level of the water goes bellow the `minimalNormal` value, the program sends a signal to the pump-controller to supply the boiler with water. The physical units interact with each other via exchanging messages. There are four pump controllers whose behavior is identical; therefore, we only specify one pump to simplify the discussion.

Figure 4.12 shows a design class model of the SBCS system. The class model is enriched with a number of OCL invariants and operations' pre and post conditions. We only present some of the important OCL constraints. One of the important invariants states that the water level should not exceed the capacity of the boiler. This invariant is specified in OCL as follows:

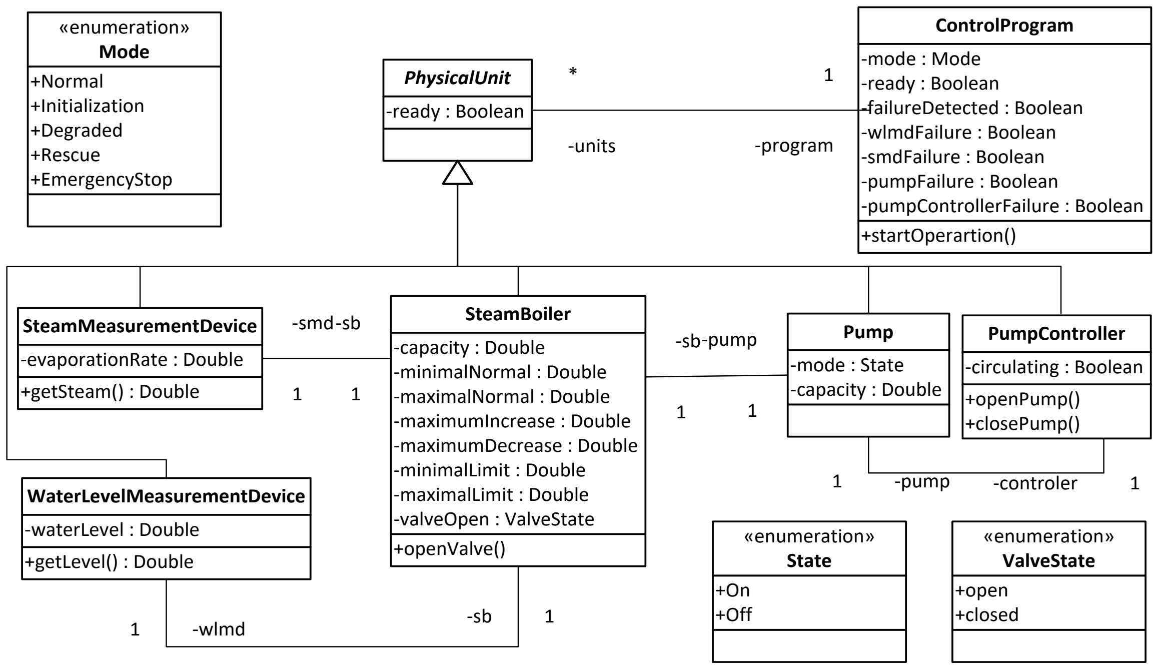


Figure 4.12: The Design Class Model for the Steam Boiler Control System, taken from our previous work Al-Lail et al. [95]

```
context WaterLevelMeasurementDevice
inv: self.waterLevel < self.sb.capacity
```

Note that the level of the water could exceed the `maximalLimit` value. In such a case, the system goes into the emergency stop mode. Another invariant states that the boiler valve can only be opened during the initialization of the system. This invariant is given below.

```
context SteamBoiler
inv: self.valveOpen=#open implies self.program.mode =#Initialization
```

The class model has five operations that change the state of the system. The `getLevel()` operation obtains the water level from the water measuring device and stores it in the variable `waterLevel`. Note that the `getLevel()` does not read and then return the value of the variable `waterLevel`, but it regularly updates it based on the measurement taken by the device. This is reflected in the postcondition of the operation, as shown below. Similar to `getLevel()`, the `getSteam()` operation gets the evaporation steam rate from the steam measuring device and writes it in the `evaporationRate` variable. The `openPump()`, `closePump()`, and `openValve()` operations open the pump, close the pump, and open the boiler valve, respectively. The OCL specifications of `getLevel()` and `openPump()` are defined below. The complete specification of the SBCS class model is provided in Appendix D.

```
context WaterLevelMeasurementDevice::getLevel(): Double
pre: self.program.mode= #Normal
post: self.waterLevel = result
context Pumpcontroller::openPump()
pre: self.pump.mode = # Off
post: self.pump.mode = # On
```

4.4.2 Analayzing the steam boiler system

We analyzed the design class model of the SBCS system to check if it satisfies the requirement that as soon as the program recognizes a failure of the water measuring device it goes into the `Rescue` mode. Ensuring that the system satisfies this property is critical to the safety of the boiler. The following sections discuss the application of the steps of our analysis technique to this analysis task.

Step1: Unfolding the behavior

In this step, we converted the class model in Figure 4.12 to its corresponding STM (see Figure 4.13). Following the guidelines in Chapter 3, a snapshot defines configurations of objects of each of the concrete classes in Figure 4.12. Note that a SBCS system consists of a single device of each of the types. That is, a SBCS system cannot have two steam boilers. Consequently, an instance of the Snapshot class can only have one object of each of the classes. Because the USE Model Validator does not have support for structured classes, we represented the $Snapshot$ class as a class that has composition relationships with the different classes that constitute the parts. As shown in Figure 4.13, the $Snapshot$ class has composition relationships to SteamBoiler, Pump,

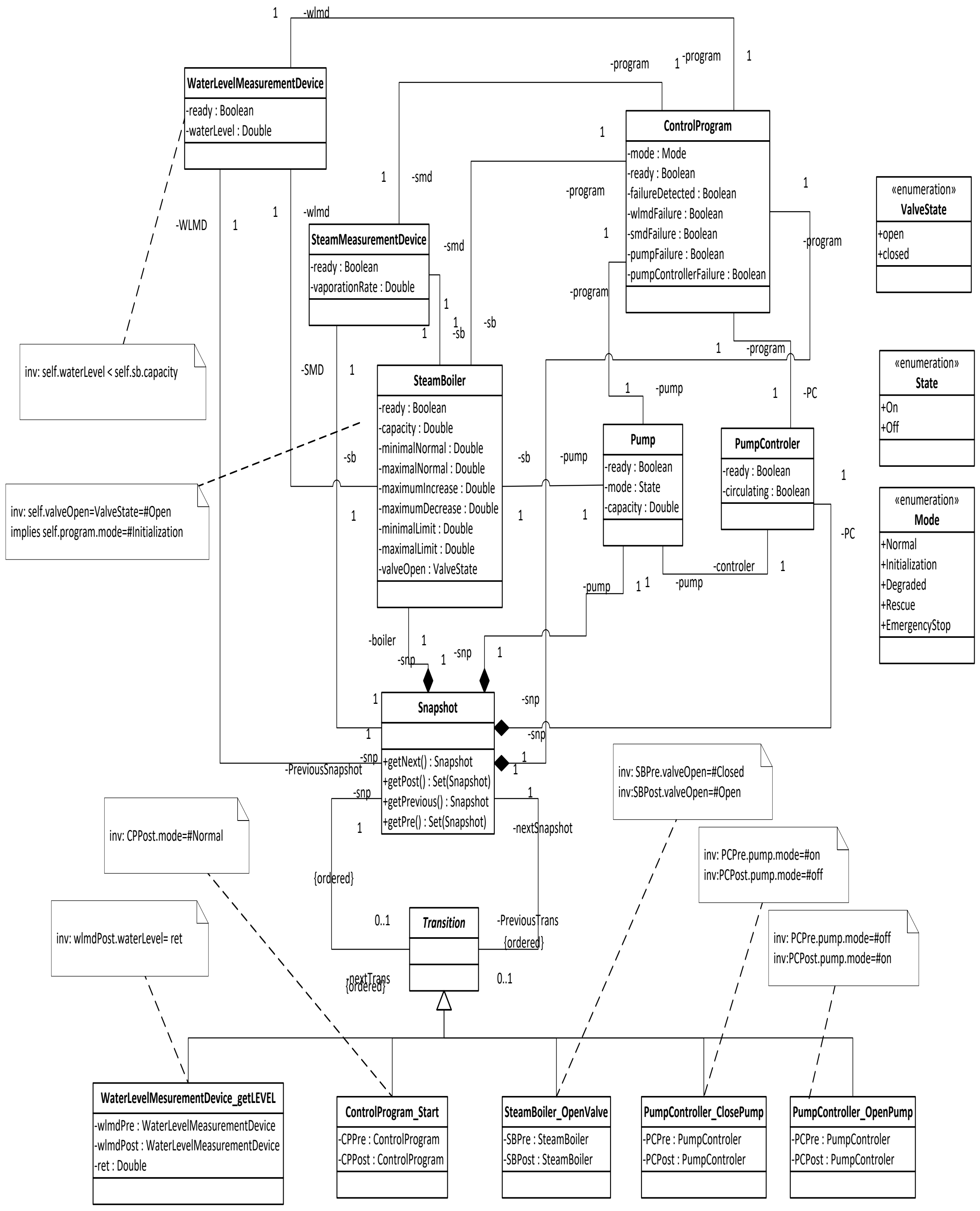


Figure 4.13: The STM of The SBCS Design Class Model, taken from our previous work Al-Lail et al. [95] with modification

PumpController, ControlProgram, SteamMeasurementDevice, and WaterLevelMeasurementDevice.

To create the hierarchy of transition classes, we generated a subclass of the abstract $Transition$ class for each operation. In the SBCS class model Figure 4.12, we only considered five modifier operations and thereby we created five subclasses of the class $Transition$ (i.e., getLevel(), startOperation(), openValve(), openPump(), and closePump()). Note these operations in Figure 4.12 and their representations in Figure 4.13.

For each parameter of an operation, we should generate two attributes that represent the value of the parameter before and after the call of the operation. However, none of the operations has a parameter. On the other hand, we defined two attributes that point to the object's states before and after an operation call (e.g., wlmdPre and wlmdPost). An attribute is also created for the return values of operation (e.g., ret).

As explained in Chapter 3, we need to represent the pre and post conditions of the operations as invariants in the STM. Figure 4.13 shows all of these invariants. We demonstrate how the getLevel() operation's pre and post conditions were defined in the STM model. The above pre and post conditions of getLevel() operation are converted to invariants in STM as follows:

```
context WaterLevelMeasurmentDevice_getLevel
inv: wlmdPre.program.mode=# Normal
inv: wlmdPost.waterLevel= ret
```

Similarly, we created transition subclasses and invariants for the other modifier operations. Figure 4.13 shows these subclasses and invariants.

Step2: Interpreting TOCL as OCL

The property that we want to check is: as soon as the program recognizes a failure of the water measuring device unit it goes into the rescue mode. This property is an instance of the response pattern; therefore, we used our specification technique to define a TOCL expression for this property as follows:

```
context ControlProgram
inv: self.wlmdFailure=true implies next self.mode= #Rescue
```

The TOCL states that if the water measuring device fails *(self.wlmdFailure=true)* then the program goes into the rescue mode ($next self.mode = \#Rescue$). Using the interpretation rules we defined for the response pattern in Chapter 3, we obtained the following OCL expression:

```
context ControlProgram
inv: let CS: Snapshot= self.snp
in NS: Snapshot= CS.getNext()
in self.wlmdFailure=true implies NS.program.mode= # Rescue
```

In the above OCL expression, the next state (NS) is returned by first getting the current snapshot, i.e., CS, and navigating to the next state by the operation *getNext()*. Then the OCL asserts that if the water measuring device fails (self.wlmdFailure=true), then the program in the next state is in the `Rescue` mode.

Step 3: Analysis

The USE Model Validator takes the STM model and the OCL property and checks if there exists an instance of the STM model that violates the OCL expression. We analyzed the model with a search_scope of one object of each class and a search_depth of 10 transitions. Figure 4.14 shows the counterexample that leads to the temporal property's violation. Note that, the actual counterexample has a total of 11 snapshots though we only show the first three snapshots that are related to the property and the design fault.

In Figure 4.14, Snapshot1 shows that none of the units were failing, and all of them were ready for operation. After starting the operation, the water level measuring device had a failure in the next state (Snapshot2). Recall that the temporal property states that after a failure in the water level measuring device, the system should go into the *Rescue* mode in the next state. However, the system remained in the normal mode in Snapshot3. This sequence of transitions violates the property.

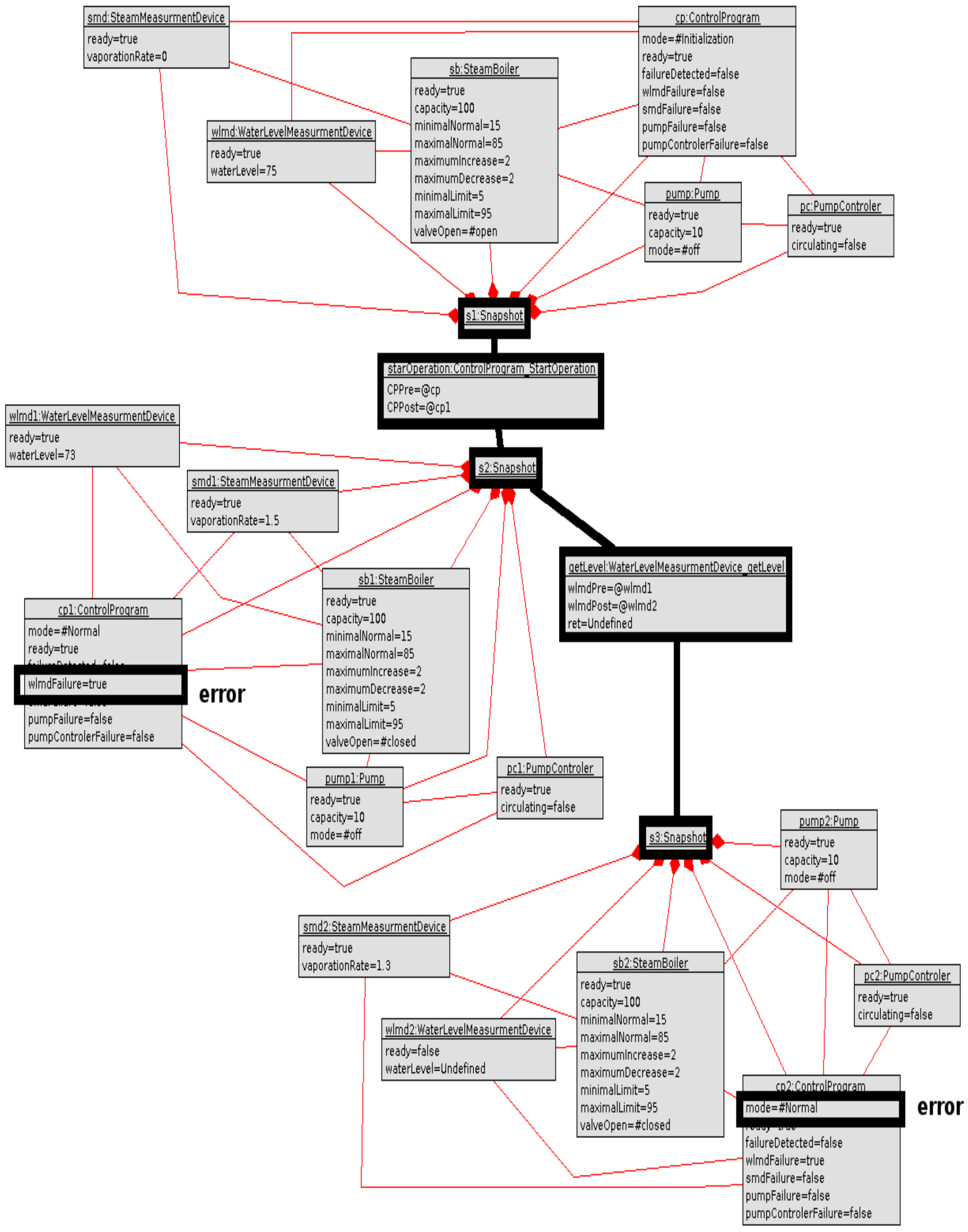


Figure 4.14: Counterexample: Scenario violating the temporal property, taken from our previous work Al-Lail et al. [95]

We examined the counterexample to find the design fault. We noted that the counterexample has an object of the WaterLevelMeasurmentDevice_getLevel class. This should not have been allowed after the failure of the water level measuring device. This situation indicated that the getLevel() operation missed a precondition to check for this condition. We also discovered that the other operations missed this precondition as well. In this case, one analysis task uncovered multiple design faults. We added the necessary preconditions to all operations' specifications to satisfy the requirement. By adding these pre conditions, we guarantee that none of the operation's calls will proceed if the water measuring device is failing. By doing a follow up analysis after the fix, we confirmed that the property was satisfied by the modified model.

4.4.3 Specifying temporal properties of the SBCS system

The SBCS system has many temporal requirements that need to be analyzed. We have specified 10 temporal properties using our specification technique. Table D.1 describes some of these properties in English. Similar to our explanation in previous section, we used the descriptions to determine the patterns and scopes. After that, we defined the TOCL properties as instances of the approperiate patterns. The interpretation rules were then employed to generate the OCL counterpart expressions to be used for the analysis. Table D.3 provides the TOCL and OCL specifications of these properties. Appendix D provides the description and specifications for the rest of the properties.

Table 4.4: Some temporal properties of the SBCS system, taken from our previous work Al-Lail et al. [95]

No.	Description	Pattern - Scope
SBCS-TP1	As soon as the program recognizes a failure of the water measuring device unit it goes into the rescue mode.	Response-Globally
SBCS-TP2	Failure of any physical units except the water measuring device puts the program into degraded mode.	Response-Globally
SBCS-TP3	If the water level is close to reaching the maximalLimit or minimalLimit values (i.e., greater than maximalNormal or less than minimalNormal) the program enters the mode emergency stop.	Response-Globally
SBCS-TP4	When the valve of the steam boiler is open, then eventually the water level will be lower or equal to the maximal normal level.	Response-Globally

Table 4.5: TOCL and OCL specification of the SBCS temporal properties described in Table D.1, taken from our previous work Al-Lail et al. [95]

No.	TOCL Specification on Class Model	OCL Specification on the Snapshot Transition Model
SBCS-TP1	**context** *ControlProgram* **inv:** *self.wlmdFailure implies* *next self.mode=# Rescue*	**context** *ControlProgram* **inv:** *inv: let CS: Snapshot= self.snp* *in NS: Snapshot= CS.getNext()* **in** *self.wlmdFailure implies NS.program.mode= # Rescue*
SBCS-TP2	**context** *ControlProgram* **inv:** *(smdFailure or pumpFailure* *or pumpcontrollerFailure) implies* *next self.mode=# Degraded*	**context** *ControlProgram* **inv:** *let CS: Snapshot= self.getCurrentSnapshot()* in let NS: Snapshot = CS.$getNext()$ **in** *(self.pumpcontrollerFailure or self.pumpFailure or* *self.smdFailure) implies NS.program.mode =# Degraded*
SBCS-TP3	**context** *SteamBoiler* **inv:** *(self.wlmd.waterLevel >* *self.maximalNormal or self.wlmd.waterLevel* *< self.minimalNormal) implies next* *self.program.mode = # EmergencyStop*	**context** *SteamBoiler* **inv:** *let CS: Snapshot = self.snp* in let NS: Snapshot = CS.$getNext()$ **in** *(self.wlmd.waterLevel > self.maximalNormal or* *self.wlmd.waterLevel < self.minimalNormal) implies* *NS.program.mode = # EmergencyStop*
SBCS-TP4	**context** *SteamBoiler* **inv:** *self.valveOpen = # open implies* *sometime* *(self.wlmd.waterLevel < = maximalNormal)*	**context** *SteamBoiler* **inv:** *let CS: Snapshot = self.snp* in let FS: Set(Snapshot) = CS.$getPost()$ **in** *self.valveOpen = # open implies FS → exists* *(s:Snapshot ∣ s.WLMD.waterLevel < = maximalNormal)*

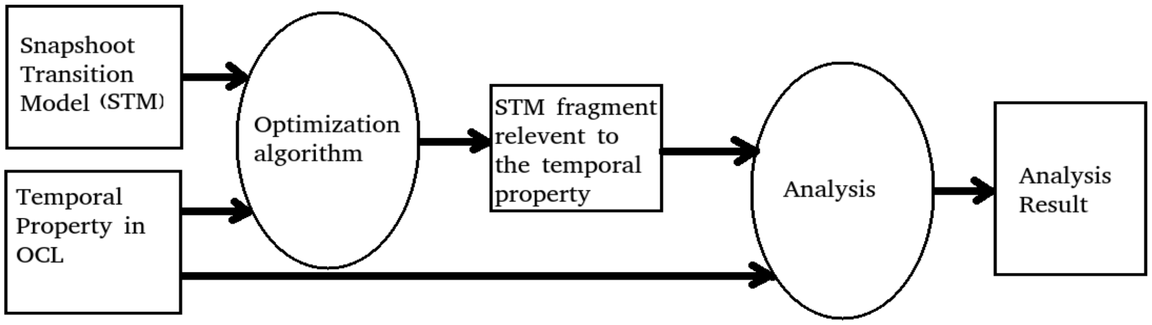

Snapshoot Transition Model (STM)
Temporal Property in OCL
Optimization algorithm
STM fragment relevent to the temporal property
Analysis
Analysis Result

Figure 4.15: Overview of the optimization technique, taken from our previous work Al-Lail et al. [96]

4.4.4 Using The Optimization Technique

Our optimization technique intends to reduce the time needed to analyze large class models. Figure 4.15 depicts an overview of the optimization technique. As described in Chapter 3, the inputs of the optimization algorithm are an STM and an OCL expression representing a TOCL temporal property. The technique uses the temporal property as a criterion to produce a fragment of the STM model that only contains the STM model elements that are relevant to the analysis of the property. If the STM fragment has fewer elements than the original STM, the analysis can handle larger analysis scopes and depths and it can take less time and memory.

In this section, we show the application of the optimization technique to the SBCS system. We also present the result of our evaluation of the technique.

Applying the optimization technique

Consider the STM of the Steam Boiler Control System in Figure 4.13. This model is denoted as SBCS-STM in this section. SBCS-STM has five transition subclasses that correspond to the operations from the original class model of the system. The SBCS-STM model has a number of OCL invariants that are: (1) transferred from the the original class model, (2) generated from operations' specifications, and (3) the STM invariants.

We applied the optimization technique to optimize the SBCS-STM with respect to the four temporal properties in Table D.3. Figure 4.16 shows the SBCM-STM fragment that is produced by

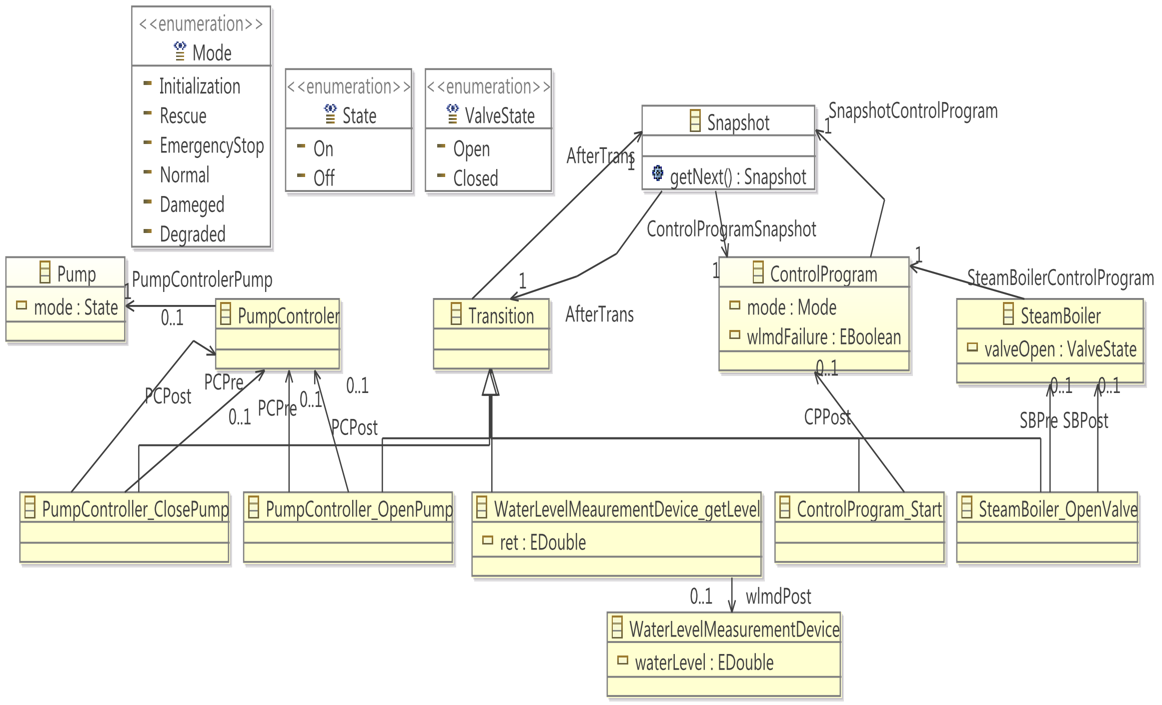


Figure 4.16: The STM slice with respect to SBCS-TP1, STM-TP1-Slice

our proof-of-concept tool for SBCS-TP1. We only show how optimization was applied to produce the fragment in Figure 4.16. Appendix D provides the fragments for the other three properties.

Algorithm 2 in Chapter 3 describes the following steps. The first step is concerned with finding the direct model elements relevant to a temporal property. SBCS-TP1 is defined in the context of class *ControlProgram*, and thus directly depends on the class *ControlProgram*. The expression *self.snp* is an association call that returns a snapshot associated with the control program (referred to by self). Thus, there is a direct dependency with reference *snp* and its type class, *Snapshot*. The expression *CS.getNext()* is an operation call expression and it returns a snapshot. There is thus a direct dependency with the operation *getNext()*, and its type class, *Snapshot*. In addition, the *getNext()* operation is defined as: *self.nextTrans.nextSnapshot*, where *self.nextTrans* is an association call and it returns a transition associated with the snapshot, and thus SBCS-TP1 directly depends on the reference *nextTrans*, and its type class Transition. The expression *self.wlmdFailure* refers to an attribute defined in class *ControlProgram*, and thus SBCS-TP1 directly depends on attribute *wlmdFailure*. Similarly, the expression *NS.program.mode* refers to an attribute defined in class *ControlProgram*, and thus SBCS-TP1 directly depends on the attribute mode. Because the attribute mode has an enumeration type, Mode, SBCS-TP1 thus directly depends on Mode. Therefore, SBCS-TP1 directly references the following STM elements *DRMElmts* = {*ControlProgram, Snapshot, Transition, snp, nextTrans, nextSnapshot, program , getNext(), wlmdFailure, Mode*}.

The second step of the algorithm is concerned with computing the set of the indirect model elements. Because the class Transition is in the direct related model element of SBCS-TP1 and the class Transition has subclasses, SBCS-TP1 indirectly references the subclasses. As a result, the set of the indirect model elements is the following: *IRMElmts* = { *WaterLevelMeasurementDevice_getLEVEL, SteamMeasurementDevice_getSTEAM, SteamBoiler_OpenValve, Pumpcontroller_ ClosePump, Pumpcontroller_OpenPump*}. The third step is combining the sets of direct and indirect model elements to produce a set of the related model elements, *RMElmts* = *DRMElmts* ∪ *IRMElmts*.

As SBCS-STM model has many invariants, and the forth step finds which of the invariants are relevant to SBCS-TP1. As described in Chapter 3, an invariant is relevant to a property if any of the invariant's related model elements is in the set of the SBCS-TP1's *RMElmts*. To find the set of an invariant's related model elements, the algorithm finds the set in the same manner of finding it for temporal properties expression, i.e., finding the direct and indirect related elements. Therefore, the tool did a similar analysis to find the *RMElmts* set for all the invariants that are defined in the SBCS-STM, and checked if this set intersects with the set of SBCS-TP1's *RMElmts*. If the intersection of the two sets is not empty, the invariant and its related model elements are included in the fragment. Figure 4.16 shows the returned ecore model of the related elements by the tool. The related invariants are returned in an .ocl file.

Evaluation of the optimization technique

We performed an experiment to evaluate the optimization technique. The experiment involved analysis of the four temporal properties in Table D.3. For each of the properties, two sets of analyses were performed: (1) the analyses performed on the unsliced model (SBCS-STM), and (2) the analyses performed on the model fragments that are relevant to the considered temporal property (e.g., STM-TP1-Slice). We collected two types of time (in milliseconds) for each set: the optimization time (OT) to obtain an optimized model (slice) for each of the properties, and the analysis time (AT) to find a counterexample, if any. We also calculated the speedup percentages.

The hypotheses were: (1) the optimization technique is capable of significantly reduce the analysis time, and (2) the analysis results are preserved. The result is preserved when the following two conditions hold:

1. The two analyses either both reveal a counterexample or not
2. In case a counterexamples are found, the design faults uncovered by debugging the counterexamples are the same

The case that any of the above conditions is not met indicates that incorrectness of the optimization algorithm.

Table 4.6 presents the results of the experiment for the four properties. Each sub-table has a column `depth` that represents the the number of transitions considered in an analysis task. The sub-table labeled SBCS-TP1 Analysis, for example, shows the times used to analyze the SBCS-STM model and STM-TP1-Slice with respect to SBCS-TP1. Note that the optimization time (`OT`) is constant for every property (e.g., 411 for SBCS-TP1). The reason is that we only need to optimize SBCS-STM once for each of the properties. After that, the optimized model (e.g., STM-TP1-Slice) is analyzed with different search_depths.

As indicated by the speedup column, the optimization technique slows down the overall analysis process for depths less than 5. This is because the time needed to slice SBCS-TP1 is longer than the time needed to analyze the SBCS-STM with depth less than 5. However, for depths greater than 5, the time for optimizing SBCS-STM with respect to SBCS-TP1 becomes relatively small compared to the analysis time. This is particularly true for depths equal or greater than 10, as indicated by the speedup column and the time needed for the analysis. Although the optimization slows down the analysis for depths less than 5, it is less than a second, which can be insignificant as compared to the analysis time of complex class models.

Table 4.6: Results of the optimization technique's experiment (OT: Optimization Time, AT: Analysis Time, Total= OT + AT, SBCS-STM: Unsliced STM mode for SBCS)

	SBCS-TP1 Analysis					**SBCS-TP2 Analysis**				
	STM-TP1-Slice			SBCS-STM		STM-TP2-Slice			SBCS-STM	
depth	OT	AT	Total	AT	Speedup	OT	AT	total	AT	Speedup
1	441	46	487	47	-936%	451	31	482	31	-1454%
2	441	94	535	94	-469%5	451	62	513	79	-549%
3	441	125	566	234	-141%	451	78	529	203	-160%
4	441	187	628	200	-214%	451	172	623	375	-66%
5	441	422	863	2046	57%	451	343	794	1672	52%
6	441	937	1378	5531	75%	451	1015	1466	3031	51%
7	441	2438	2879	5828	50%	451	2031	2482	5750	56%
8	441	2157	2598	9313	72%	451	3312	3763	12172	69%
9	441	5126	5567	75254	92%	451	5656	6107	38283	84%
10	441	11532	11973	30845	61%	451	10391	10842	24923	56%
11	441	15032	15473	190572	91%	451	13938	14389	172415	91%
12	441	24470	24911	207245	87%	451	27627	28078	372503	92%
13	441	36268	36709	391801	90%	451	33267	33718	608140	94%
	TP3 Analysis					**SBCS-TP4 Analysis**				
	STM-TP3-Slice			SBCS-STM		STM-TP4-Slice			SBCS-STM	
depth	OT	AT	Total	AT	Speedup	OT	AT	total	AT	Speedup
1	455	24	475	234	-102%	538	32	570	47	-1112%
2	455	49	500	156	-220%	538	47	585	78	-650%
3	455	86	537	234	-129%	538	156	694	171	-305%
4	455	125	576	328	75%	538	188	726	391	-85%
5	455	391	842	2375	64%	538	547	1085	1078	-0.6%
6	455	922	1373	8907	84%	538	1157	1695	2016	15%
7	455	2063	2514	4610	45%	538	2328	2866	5407	46%
8	455	3750	4201	9110	53%	538	3609	4147	15688	73%
9	455	6563	7014	61096	88.5%	538	7469	8007	73316	89%
10	455	10407	10858	81566	86%	538	9626	10164	21204	52%
11	455	16766	17217	419393	95%	538	11282	11820	512242	97%
12	455	26563	27014	151711	82%	538	16626	17164	135283	87%
13	441	32627	33078	477947	93%	538	34123	34661	511478	93%

The experiment also showed that the analysis results are preserved by the optimization technique. That is, the analysis performed on the unsliced model and on the slices are consistent in checking the validity of a property. For example, the analysis of TP1 on SBCS-STM revealed a counterexample; this is consistent with the analysis on STM-TP1-Slice, which revealed a counterexample as well. The design fault found in both cases is the same, which is the missing preconditions of the operations.

4.5 Chapter Summary

In this chapter, we applied the techniques in the context of two case studies: the GSTRBAC model and the SBCS system. The first objective of the case studies was to evaluate the expressiveness of the specification technique in specifying a variety of properties. Using the specification technique, we successfully specified 35 temporal properties in the GSTRBAC model and 14 properties in the SBCS system. It is worth emphasizing that during this process, we did not encounter any property that we could not specify. The properties were a mixture of safety and liveness properties, many of which were obvious instances of the proposed patterns. Table 4.7 shows the distribution of the properties.

Table 4.7: Distribution of the properties we specified

Pattern	Absence	Universality	Existence	Bounded Existence
Number of properties	5	2	0	0
Pattern	Precedence	Response	Chain Precedence	Chain Response
Number of properties	4	28	0	0

Our findings regarding the most used patterns is in agreement with the results found by Dwyer et al. [66]. In particular, as shown in the table above, the most widely used patterns are Response, Universality, and Existence. This is similar to the finding in Dwyer et al. [66] as they showed that the most used patterns are Response, Universality, and Absence. Note that the Absence is the logical dual of Existence, i.e., the existence of one event is the absence of its negation.

The second objective of the case studies was to evaluate the effectiveness of the analysis technique in finding design faults. As we demonstrated in this chapter, the analysis technique is capable of finding faults. We were able to uncover more that 50 design faults while designing the GSTRBAC model. Also, 16 design faults were found when we analyzed the SBCS system. We identified the design faults we found into two categories:

1. **Permissive models**. The models in this case were under-specified, that is, the analyses revealed that the models were missing needed constraints. The under-specification results in violations of temporal properties by allowing unwanted states to be reached. The design faults can further be classified as:
   - Missing pre and post conditions of operations
   - Missing class invariants
2. **Restrictive models**. The models in this case were overconstrained, that is, the analyses revealed that the models were augmented with unnecessary constraints. Consequently, some of the required behavior could not be obtained. For example, a combination of operations' specifications do not allow the system to reach desired states. The design faults can further be classified as:
   - Unneeded pre or post conditions
   - Unneeded class invariants

The third objective was to evaluate the usefulness of the optimization technique in saving time when analyzing large models. As shown in Table 4.6, we found that optimization can significantly speed up the analysis for depths greater than 10. For relatively small models, such as the SBCS system, the time saved by optimization is in order of minutes. To illustrate, as shown in Table 4.6, the time saved by optimization is almost 10 minutes, when analyzing SBCS-TP2 with depth 13. However, recall that the optimization technique is property-based. For temporal properties that require a substantial number of model elements of the original model, the optimization technique produces a slightly smaller slice than the original model. In such cases, the technique provides little time improvement. Nevertheless, this situation is very rare in realistic complex system.

Chapter 5

Conclusions

Software is everywhere, though we might be heedless of the extent of its influence, it plays an increasing role in our society and economy. For example, correct functionality of modern cars, mobile phones, and medical devices depends on the correctness of the internal software. However, we have witnessed incidents in which software systems have disappointed us by failing to do what they are intended to do or by doing what is undesired, causing catastrophic consequences to businesses and individuals. Such problems pose a significant challenge to the Software Engineering community. This challenge is how to create techniques and tools that assist designers in reducing, or even eliminating errors in software systems regardless of their complexity.

There has been a great deal of research in developing new techniques to overcome this challenge. Model-Driven Engineering (MDE) has been developed to alleviate such problems, and has been established as a new paradigm for developing reliable, complex software systems. In the context of MDE, designers use the Unified Modeling Language (UML) to create models that drive the entire development process. Once UML models are created, MDE techniques automatically generate code from the models. However, the MDE community lacks adequate techniques and tools for specifying and analyzing temporal properties that aid in uncovering design faults in software models. Model analysis must be done before design faults are automatically propagated to code. Most of today's state-of-the-art research focuses on using Model Transformation to Model Checking, which is the prominent approach to verifying temporal properties. Unfortunately, this approach complicates the process and results in a number of difficulties.

First, effective use of the state-of-the-art techniques requires UML designers to develop deep working knowledge of the Model Checking techniques. These techniques are not based on MDE and UML notations and tools. Further, studies have shown that developing adequate skills is challenging for many Model Checking practitioners, let alone UML designers who might lack proper training and mathematical maturity. Second, ensuring the reliability of the analysis results produced requires a proof of accuracy of the transformations involved. Obtaining such a proof is not trivial. Third, the analysis results obtained by the back-end Model Checking tools need to be presented to designers in UML terms in order to uncover design faults. This requires another transformation process that gives rise to the same difficulties inherent in the initial transformation.

Broadly speaking, the research described in this dissertation aims to enhance the field of specification and analysis of temporal properties using UML models. It does this by developing a native UML-based framework that is composed of a set of UML-based notations, techniques and tools. The proper use of this set amplifies UML designers' skills in developing reliable software systems, instead of forcing the designers to learn and deal with the complications of the existing non-UML approaches.

This concluding chapter is organized as follows. In Section 5.1, we revisit contributions of the research described in this dissertation and discuss the limitations of the work. We then, in Section 5.2, point to some of the future research that can be done to enhance the usability and applicability of the framework.

5.1 Summary of the Contributions and Limitations

In this section, we first examined the aim and objectives that we presented in the introduction. Then, we summarized the contributions and discussed the limitations of our framework. The aim and the objectives of this research were as follows:

- **Aim:** To develop a UML-based framework consisting of techniques and tools to formally specify and analyze temporal properties of software designed using UML.
- **Objectives:**
  1. To explore the state-of-the-art techniques and tools in order to identify research gaps and challenges in the field of Model Checking UML models.
  2. To develop a UML-based analysis technique that exclusively uses UML notations and tools.

3. To streamline the process of specifying temporal properties for UML designers by developing a specification technique that uses UML notations.
4. To develop an optimization technique that reduces the time needed for analysis, allowing the analysis to be scaled to larger UML models.
5. To provide a proof-of-concept tool by implementing the specification, analysis, and optimization techniques.
6. To evaluate the framework through an actual software specification and analysis projects.

In Chapter 2, we examined the state-of-the-art approaches to analyzing temporal properties of systems that are modeled using UML. We categorized the material into specification and analysis techniques and then we applied the Systematic Literature Review (SLR) method to identify open problems and gaps in the current research. We found that many of the existing approaches and tools are inadequate and they introduce accidental complexities that hinder their effectiveness and efficiency. We formulated the challenges as open research questions that, if properly addressed, will lead to improvements toward achieving the research objectives. We then categorized the open research questions into the following groups: (1) specification and analysis integration, (2) Model Transformation, (3) property discovery and patterns, and (4) tooling. This dissertation focused on developing a framework that addresses some of the challenges related to integrating specification and analysis, property pattern, and tools. It is beneficial to the MDE community to find answers to the identified questions, as their answers may yield insights that are applicable to other MDE areas (such as Model Transformation and development of MDE-based tools).

In Chapter 3, we proposed a UML-based framework. We discussed and provided details of the framework's three techniques that addresses the second, third, and fourth objectives of our research. A user of the framework can specify and analyze temporal properties without the need for transforming UML models to other languages. The framework's analysis technique determines if a temporal property holds with respect to a set of automatically generated scenarios (i.e., representations of system executions). The number of the checked scenarios can be increased to increase the designer's confidence that a temporal property holds in a particular model. The analysis technique, however, can not guarantee that the property holds within scenarios that are not checked. In addition, using the framework's specification technique, the process of specifying temporal properties is streamlined by the use of UML-based property patterns. Following two simple steps, UML designers can use the provided property patterns to specify temporal properties on UML class models using UML notations (i.e., TOCL and OCL languages). Hence, any class model analysis tool could be employed to perform the analysis task. Finally, we discussed the framework's optimization technique that is used to scale the analysis to larger class models.

In Chapter 4, we evaluated the framework's techniques using two case studies. The first case study was based on the development of our Generalized Spatio-Temporal Role-Based Access Control Model(GSTRBAC) Abdunabi et al. [67]. We developed this model to address the many application requirements of wireless and mobile devices that make use of the spatio-temporal information of a user to provide better functionality. Such applications necessitate authorization models where access to a resource depends on the credentials of the user, the user's location, and time of access. The second case study was based on the Steam Boiler Control System (SBCS) specification problem, Abrial et al. [68]. The SBCS specification problem has been used extensively to assess the effectiveness of many software specification and verification approaches. Using SBCS, therefore, provided a benchmark study that can be used to compare the framework with other methods.

We developed a proof-of-concept research prototype to investigate the effectiveness of the framework's techniques. The effectiveness of the three techniques was determined as follows: (1) the specification technique was judged based on its ability to specify temporal properties of different kinds, (2) the analysis technique was judged based on the ability of the technique to find design faults, and (3) the optimization technique was judged based on how much time was saved as compared with the time needed for analysis without optimization.

Using the specification technique, we successfully specified 29 temporal properties in the GSTRBAC model and 10 properties in SBCS. The properties were a mixture of the two types of properties: safety and liveness. Many of them were obvious instances of the proposed patterns that provide a mean to specify many types of properties. It is worth emphasizing that we have not encountered any property that we have been unable to specify. In addition, our findings, regarding the most used patterns, agree with studies performed by other researchers (i.e., Dwyer et al. [66]). Though the case studies showed that our specification technique was expressive to specify a variety of properties, a more thorough comparison was needed to confirm the expressiveness of the TOCL and OCL with the original forms of Dwyer's patterns.

The specification technique, however, has a shortcoming. It uses a linear-time temporal logic language instead of branching-time logic. Each type of logic could express certain properties that the other could not express effectively. To accommodate such properties, we plan to develop a new UML-based branching-time logic language and incorporate it in our framework.

The analysis technique has shown itself to be capable of finding design faults. Using the technique we were able to uncover more than 50 design faults while designing the GSTRBAC model. Also, 16 design faults were found when we analyzed the SBCS system. Note that in both case studies, the design faults were not seeded in the models to evaluate the effectiveness of the analysis technique, but were uncovered during the analysis phase, not after the final models were released. The number of design faults found is typical when designing and analyzing complex systems. We identified two categories of design faults that were found by our analyses. The majority of the design faults were due to permissive models. The analyses revealed that the models were missing needed class invariants, or operations' pre- and postconditions. The missing constraints allowed the systems to reach prohibited states that violated temporal properties. To address design faults in this category, we needed to add the missing constraints. The second, less common, category of design faults are due to the models being too restrictive, which resulted in disallowing required behavior, which again violated certain requirements. In this case, the models originally included unnecessary constraints. To correct design faults due to this category, we needed to find and remove the unwanted constraints to allow the desired behavior. The case studies, therefore, showed that the analysis technique is effective in finding design faults.

While the analysis technique is effective in analyzing properties and uncovering design faults, it is only able to do it for certain types of properties. In particular, the technique can only handle bounded safety and liveness properties. These properties are restricted by a boundary that limits the number of transitions to be executed and analyzed. Therefore, the technique is unable to analyze unbounded liveness properties such as fairness properties (e.g., every process should be executed infinitely often). The reason is that fairness properties require the analysis of infinite execution traces while our technique is only able to analyze finite execution traces. To address this limitation, we plan to investigate how other techniques that have this shortcoming address this problem. In particular, we will investigate and leverage the techniques used by Bounded Model Checking [64].

With regard to evaluating the usefulness of the optimization technique, we compared the analyses performed with optimization with the analyses performed without optimization. We focused on two aspects for this comparison. The first was on whether the optimization technique preserved the analysis results. That is, if the analyses performed with optimization found the same design faults that were found by the analyses without optimization. The second aspect was how much speedup was obtained when measuring the analysis time with and without the optimization. The analyses produced the same results in all the cases that were done on 4 temporal properties of the SBCS system. We also found that optimization can significantly speed up the analysis for large models.

A limitation of the optimization technique is that it is property-based. That is, the time improvement of the the optimization technique depends on the nature of the property being analyzed. If a property is connected with a small number of elements, the optimization is very beneficial. However, in case that a property requires a significant number of model elements, the optimization provides little time improvement.

5.2 Future Research

The research described in this dissertation has been successful in accomplishing its objectives and achieving its aim. However, much more work can be done to further improve and refine our contributions. The ideas presented in this dissertation have a great potential to be extended to help meet the vision of MDE. Our future research will be mainly focused on contributing to the global effort to establish MDE as a software development paradigm that is utilized to develop complex industrial software projects.

We have formulated a few future research projects that build on the work of our research. The following is a short description of three of these projects.

Project 1: An Object-Oriented Branching Tree Logic Language

Our framework relies on the use of specification approach that uses a linear-time temporal logic language instead of branching-time logic. Each type of logic could express certain properties that the other could not express effectively. Using the current framework, therefore, a designer could find it difficult to specify non-linear temporal properties. To add more expressiveness and power to the framework, we plan to develop an MDE branching-time logic language for specifying temporal properties that are not expressed as effectively by the current language, TOCL.

Project 2: Evaluating the Property Specification Patterns

As discussed in Chapter 2, specifying temporal properties in formal languages is challenging. Our specification technique alleviates some of the difficulties. The technique is based on Dwyer's eight property specification patterns. However, these patterns, while very useful, were proposed more than 18 years ago. Many different systems have been investigated since then; hence, new types of requirements may have emerged. In this project, we plan to investigate and evaluate the applicability of the patterns to the new requirements and suggest proper improvements. Depending on this evaluation's result, we will adapt our specification technique to meet the emerging requirements.

Project 3: Discovering Software Properties

Many software specification and analysis techniques, such as the framework developed in this dissertation, require specification of the target system's desirable properties. When these properties are known, they can be specified using appropriate formalisms that can then be verified using different tools. Specifying these properties requires a deep understanding of the structure and the behavior of a system. However, for early stages of a system's development, many of the properties are unknown. In this project, we plan to develop a technique that will aid in discovering desirable system properties, which will aid in better understanding the system requirements. By implementing this technique, the designers using our framework will be able to discover desirable properties that may have been overlooked. Subsequently, these properties can be specified and analyzed by the framework. Additionally, because temporal properties can be very similar, this project will also investigate their consistency, independence, and consequences.

Bibliography

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

[2] Wikipedia. Pentium FDIV bug.

[3] Douglas Isbell, Mary Hardin, and Joan Underwood. MARS CLIMATE ORBITER TEAM FINDS LIKELY CAUSE OF LOSS .

[4] Jean-Marc Jézéquel and Bertrand Meyer. Design by Contract: The Lessons of Ariane. *IEEE Computer*, 30(1):129–130, 1997.

[5] Nancy G. Leveson. An investigation of the therac-25 accidents. *IEEE Computer*, 26:18–41, 1993.

[6] Robert L. Glass. Sorting out software complexity. *Commun. ACM*, 45(11):19–21, 2002.

[7] Frederick P. Brooks Jr. *The mythical man-month - essays on software engineering (2. ed.)*. Addison-Wesley, 1995.

[8] Robert France and Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society.

[9] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[10] John Hutchinson, Mark Rouncefield, and Jon Whittle. Model-driven engineering practices in industry. In *ICSE*, pages 633–642, 2011.

[11] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of MDE in industry. In *Proceedings of the 33rd International Conference on*

Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011, pages 471–480, 2011.

[12] Parastoo Mohagheghi and Vegard Dehlen. Where is the proof? - A review of experiences from applying MDE in industry. In *Model Driven Architecture - Foundations and Applications, 4th European Conference, ECMDA-FA 2008, Berlin, Germany, June 9-13, 2008. Proceedings*, pages 432–443, 2008.

[13] Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *IEEE Software*, 31(3):79–85, 2014.

[14] Brian Dobing and Jeffrey Parsons. How UML is used. *Commun. ACM*, 49(5):109–113, 2006.

[15] Marian Petre. Uml in practice. In *ICSE*, pages 722–731, 2013.

[16] Bran Selic. What will it take? A view on adoption of model-based methods in practice. *Software and System Modeling*, 11(4):513–526, 2012.

[17] Object Management Group. UML V2.5 , 2015.

[18] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley Professional, second edition, 2005.

[19] Object Management Group. OCL V2.4 , 2014.

[20] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.

[21] David Harel. On visual formalisms. *Commun. ACM*, 31(5):514–530, 1988.

[22] Amir Pnueli. The Temporal Logic of Programs. In *FOCS*, pages 46–57, 1977.

[23] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[24] Association for Computing Machinery. A.M. TURING AWARD WINNERS, 1996.

[25] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[26] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 2001.

[27] Association for Computing Machinery. A.M. TURING AWARD WINNERS, 2007.

[28] Association for Computing Machinery. ACM SOFTWARE SYSTEM AWARD WINNERS, 2001.

[29] Gerard Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, first edition, 2003.

[30] Amir Pnueli. The temporal semantics of concurrent programs. In *Proceedings of the International Sympoisum on Semantics of Concurrent Computation*, pages 1–20, London, UK, 1979. Springer-Verlag.

[31] Andy Evans, Robert B. France, Kevin Lano, and Bernhard Rumpe. The UML as a Formal Modeling Notation. In *The Unified Modeling Language, «UML»'98: Beyond the Notation, First International Workshop, Mulhouse, France, June 3-4, 1998, Selected Papers*, pages 336–348, 1998.

[32] Johan Lilius, Ivan Porres, Ivan Porres Paltor, Turku Centre, and Computer Science. vuml: a tool for verifying uml models. pages 255–258, 1999.

[33] Luciano Baresi, Gundula Blohm, Dimitrios S. Kolovos, Nicholas Matragkas, Alfredo Motta, Richard F. Paige, Alek Radjenovic, and Matteo Rossi. Formal Verificattion and Validation

of Embedded systems: the UML-based MADES Approach. *Software and System Modeling*, 2(3):164–186, 2013.

[34] Faiez Zalila, Xavier Crégut, and Marc Pantel. Formal Verification Integration Approach for DSML. In *MoDELS*, pages 336–351, 2013.

[35] Karolina Zurowska and Jürgen Dingel. Model Checking of UML-RT Models Using Lazy Composition. In *MoDELS*, pages 304–319, 2013.

[36] Yann Moffett, Jürgen Dingel, and Alain Beaulieu. Verifying Protocol Conformance Using Software Model Checking for the Model-Driven Development of Embedded Systems. *IEEE Trans. Software Eng.*, 39(9):1307–13256, 2013.

[37] Rik Eshuis. Symbolic Model Checking of UML Activity Diagrams. *ACM Trans. Softw. Eng. Methodol.*, 15:1–38, January 2006.

[38] Shao Jie Zhang and Yang Liu. An Automatic Approach to Model Checking UML State Machines. In *SSIRI (Companion)*, pages 1–6, 2010.

[39] Timm Schäfer, Alexander Knapp, and Stephan Merz. Model Checking UML State Machines and Collaborations. *Electr. Notes Theor. Comput. Sci.*, 55(3):357–369, 2001.

[40] Wuwei Shen and Weng Liong Low. Using Abstract State Machines to Support UML Model Instantiation Checking. In *IASTED Conf. on Software Engineering*, pages 100–105, 2005.

[41] Jori Dubrovin and Tommi A. Junttila. Symbolic Model Checking of Hierarchical UML State Machines. In *ACSD*, pages 108–117, 2008.

[42] Alexander Raschke. Translation of UML 2 Activity Diagrams into Finite State Machines for Model Checking. In *EUROMICRO-SEAA*, pages 149–154, 2009.

[43] Artur Niewiadomski, Wojciech Penczek, and Maciej Szreter. A New Approach to Model Checking of UML State Machines. *Fundam. Inform.*, 93(1-3):289–303, 2009.

[44] Fei Xie, Vladimir Levin, and James C. Browne. Model checking for an executable subset of uml. In *ASE*, pages 333–336, 2001.

[45] Franck Chauvel and Jean-Marc Jézéquel. Code generation from UML models with semantic variation points. In *Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, Proceedings*, pages 54–68, 2005.

[46] Bernhard Rumpe and Robert B. France. Variability in UML language and semantics. *Software and System Modeling*, 10(4):439–440, 2011.

[47] Sten Agerholm and Peter Gorm Larsen. A Lightweight Approach to Formal Methods. In *Proceedings of the International Workshop on Current Trends in Applied Formal Method: Applied Formal Methods*, FM-Trends 98, pages 168–183, London, UK, 1999. Springer-Verlag.

[48] Edmund M. Clarke. The Birth of Model Checking. In *25 Years of Model Checking*, pages 1–26, 2008.

[49] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *FMSP*, pages 7–15, 1998.

[50] Robin Milner. *Communication and concurrency*. PHI Series in Computer Science. Prentice Hall, 1989.

[51] Benoit Baudry, Sudipto Ghosh, Franck Fleurey, Robert B. France, Yves Le Traon, and Jean-Marie Mottu. Barriers to systematic model transformation testing. *Commun. ACM*, 53(6):139–143, 2010.

[52] Benoit Baudry, Trung Dinh-trong, Jean marie Mottu, Devon Simmonds, Robert France, Sudipto Ghosh, Franck Fleurey, and Yves Le Traon. Traon. model transformation testing challenges. In *In Proceedings of IMDT workshop in conjunction with ECMDAâĂŹ06*, 2006.

[53] Ismênia Galvão and Arda Goknil. Survey of Traceability Approaches in Model-Driven Engineering. In *EDOC*, pages 313–326, 2007.

[54] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. On Challenges Of Model Transformation From UML To Alloy. *Software and System Modeling*, 9(1):69–86, 2010.

[55] Kyriakos Anastasakis. *A Model Driven Approach for the Automated Analysis of UML Class Diagrams*. PhD thesis, School of Computer Science, 2009.

[56] Daniel Jackson. Alloy: A Lightweight Object Modeling Notation. *ACM Transactions on Software Engneering Methodology*, 11(2):256–290, 2002.

[57] Daniel Jackson. Lightweight formal methods. In *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings*, page 1, 2001.

[58] Steve M. Easterbrook, Robyn R. Lutz, Richard Covington, John Kelly, Yoko Ampo, and David Hamilton. Experiences Using Lightweight Formal Methods for Requirements Modeling. *IEEE Trans. Software Eng.*, 24(1):4–14, 1998.

[59] Paul Ziemann and Martin Gogolla. OCL Extended with Temporal Logic. In *Ershov Memorial Conference*, pages 351–357, 2003.

[60] Mirco Kuhlmann, Lars Hamann, and Martin Gogolla. Extensive validation of ocl models by integrating sat solving into use. In Judith Bishop and Antonio Vallecillo, editors, *TOOLS (49)*, volume 6705 of *Lecture Notes in Computer Science*, pages 290–306. Springer, 2011.

[61] Lijun Yu, Robert B. France, and Indrakshi Ray. Scenario-Based Static Analysis of UML Class Models. In *MoDELS*, pages 234–248, 2008.

[62] Lijun Yu, Robert France, and Indrakshi Ray. Scenario-Based Static Analysis of UML Class Models. In *Proceedings of the 11th international conference on Model Driven Engineering*

Languages and Systems, MoDELS '08, pages 234–248, Berlin, Heidelberg, 2008. Springer-Verlag.

[63] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press. ISBN: 0262101149, 2006.

[64] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *TACAS*, pages 193–207, 1999.

[65] W. Sun, R. France, and I. Ray. Contract-Aware Slicing of UML Class Models. In *Model-Driven Engineering Languages and Systems*, pages 724–739. Springer, 2013.

[66] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 411–420, New York, NY, USA, 1999. ACM.

[67] Ramadan Abdunabi, Mustafa Al-Lail, Indrakshi Ray, Robert France. Specification, Validation, and Enforcement of a Generalized Spatio-Temporal Role-Based Access Control Model. *IEEE Systems Journal*, 2013.

[68] Jean-Raymond Abrial, Egon Börger, and Hans Langmaack. The Stream Boiler Case Study: Competition of Formal Program Specification and Development Methods. In *Formal Methods for Industrial Applications*, pages 1–12, 1995.

[69] Barbara A. Kitchenham, Tore Dybå, and Magne Jørgensen. Evidence-based software engineering. In *ICSE*, pages 273–281, 2004.

[70] Edmund M. Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David E. Long, Kenneth L. McMillan, and Linda A. Ness. Verification of the futurebus+ cache coherence protocol. *Formal Methods in System Design*, 6(2):217–232, 1995.

[71] Leslie Lamport. sometime" is sometimes not never" - on the temporal logic of programs. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, USA, January 1980*, pages 174–185, 1980.

[72] Bilal Kanso and Safouan Taha. Temporal Constraint Support for OCL. In *SLE*, pages 83–103, 2012.

[73] Mustafa Al-Lail, Ramadan Abdunabi, Robert B. France, and Indrakshi Ray. Rigorous analysis of temporal access control properties in mobile systems. In *ICECCS*, pages 246–251, 2013.

[74] Stephan Flake and Wolfgang Müller. Formal Semantics of Static and Temporal State-Oriented OCL Constraints. *Software and System Modeling*, 2(3):164–186, 2003.

[75] Michael Soden and Hajo Eichler. Temporal extensions of ocl revisited. In *ECMDA-FA*, pages 190–205, 2009.

[76] Markus Scheidgen and Joachim Fischer. Human Comprehensible and Machine Processable Specifications of Operational Semantics. In *Proceedings of the 3rd European Conference on Model Driven Architecture-foundations and Applications*, ECMDA-FA'07, pages 157–171, Berlin, Heidelberg, 2007. Springer-Verlag.

[77] Julian C. Bradfield, Juliana Küster Filipe, and Perdita Stevens. Enriching OCL Using Observational Mu-Calculus. In *FASE*, pages 203–217, 2002.

[78] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *ICSE*, pages 411–420, 1999.

[79] Michelle L. Crane and Jürgen Dingel. UML vs. classical vs. rhapsody statecharts: not all models are created equal. *Software and System Modeling*, 6(4):415–435, 2007.

[80] Doron Peled. On projective and separable properties. *Theor. Comput. Sci.*, 186(1-2):135–156, 1997.

[81] Johan Lilius and Iván Porres Paltor. Formalising UML state machines for model checking. In *Proceedings of the 2nd international conference on The unified modeling language: beyond the standard*, UML'99, pages 430–444, Berlin, Heidelberg, 1999. Springer-Verlag.

[82] Tony Clark, Robert B. France, Martin Gogolla, and Bran Selic. Meta-modeling model-based engineering tools (dagstuhl seminar 13182). *Dagstuhl Reports*, 3(4):188–226, 2013.

[83] Jon Whittle, John Hutchinson, Mark Rouncefield, Håkan Burden, and Rogardt Heldal. Industrial adoption of model-driven engineering: Are the tools really the problem? In *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings*, pages 1–17, 2013.

[84] Tony Clark, Robert B. France, Martin Gogolla, and Bran V. Selic. Meta-Modeling Model-Based Engineering Tools (Dagstuhl Seminar 13182). *Dagstuhl Reports*, 3(4):188–226, 2013.

[85] Johan Den Haan. 8 Reasons Why Model-Driven Approaches (will) Fail, 2008.

[86] Adrian Kuhn, Gail C. Murphy, and C. Albert Thompson. An Exploratory Study of Forces and Frictions Affecting Large-Scale Model-Driven Development. In *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings*, pages 352–367, 2012.

[87] Federico Tomassetti, Marco Torchiano, Alessandro Tiso, Filippo Ricca, and Gianna Reggio. Maturity of software modelling and model driven engineering: A survey in the italian industry. In *16th International Conference on Evaluation & Assessment in Software Engineering, EASE 2012, Ciudad Real, Spain, May 14-15, 2012. Proceedings*, pages 91–100, 2012.

[88] Dan Chiorean, Mihai Paşca, Adrian Cârcu, Cristian Botiza, and Sorin Moldovan. Ensuring UML Models Consistency Using the OCL Environment. *Electron. Notes Theor. Comput. Sci.*, 102:99–110, November 2004.

[89] Ivan Porres and Irum Rauf. Generating Class Contracts from Deterministic UML Protocol Statemachines. In *Models in Software Engineering, Workshops and Symposia at MODELS 2009, Denver, CO, USA, October 4-9, 2009, Reports and Revised Selected Papers*, pages 172–185, 2009.

[90] Ivan Porres and Irum Rauf. From Nondeterministic UML Protocol Statemachines to Class Contracts. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010*, pages 107–116, 2010.

[91] Mustafa Al-Lail. A framework for specifying and analyzing temporal properties of UML class models. In *Joint Proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013), Miami, USA, September 29 - October 4, 2013.*, pages 112–117, 2013.

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

[93] Mathias Soeken, Robert Wille, Mirco Kuhlmann, Martin Gogolla, and Rolf Drechsler. Verifying uml/ocl models using boolean satisfiability. In *MBMV*, pages 57–66, 2010.

[94] Dino Distefano. *On Model Checking the Dynamics of Object-Based Software - a Foundational Approach*. PhD thesis, University of Twente, 2003.

[95] Mustafa Al-Lail, Ramadan Abdunabi, Robert B. France, and Indrakshi Ray. An Approach to Analyzing Temporal Properties in UML Class Models. In *MoDeVVa@MoDELS*, pages 77–86, 2013.

[96] Mustafa Al-Lail, Wuliang Sun, and Robert B. France. Analyzing behavioral aspects of UML design class models against temporal properties. In *2014 14th International Conference on Quality Software, Allen, TX, USA, October 2-3, 2014*, pages 196–201, 2014.

[97] Lijun Yu. *A Scenario-Based Technique To Analyze UML Design Class Models*. PhD thesis, Colorado State University, 2014.

[98] Wuliang Sun, Benoît Combemale, Robert B. France, Arnaud Blouin, Benoit Baudry, and Indrakshi Ray. Using Slicing to Improve the Performance of Model Invariant Checking. *Journal of Object Technology*, 14(4):1:1–28, 2015.

[99] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[100] Ravi Sandhu, Edward Coyne, Hal Feinstein, and Charles Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, February 1996.

[101] Larman Craig. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall, third edition, 2004.

[102] James Joshi, Elisa Bertino, Usman Latif, and Arif Ghafoor. A Generalized Temporal Role-Based Access Control Model. *IEEE Transactions on Knowledge and Data Engineering*, 17(1):4–23, 2005.

[103] Gail-Joon Ahn and Ravi Sandhu. Role-based authorization constraints specification. *ACM Transactions on Information Systems Security*, 3(4):207–226, 2000.

[104] Gail-Joon Ahn and Michael Shin. Role-based authorization constraints specification using object constraint language. In *Proceedings of the 10th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 157–162, June 2001.

[105] Object Management Group. UML 2.3 Superstructure, may 2010.

[106] Martin Gogolla, Lars Hamann, Frank Hilken, Mirco Kuhlmann, and Robert B. France. From Application Models to Filmstrip Models: An Approach to Automatic Validation of Model

Dynamics. In *Modellierung 2014, 19.-21. März 2014, Wien, Österreich*, pages 273–288, 2014.

Appendix A

The USE Specification of The Traffic Light System

A.1 The Textual Specification

```
model TrafficLightSTM

enum Color {Green, Red}

class PedestrianButton
attributes
  ID: Integer
  counter: Boolean
  operations
  requestPss(): Boolean
end

class TrafficLight
attributes
  ID : Integer
  requested: Boolean
  pedLight: Color
  carLight: Color
  operations
  swithCarLight(): Color
  switchPedLight(): Color
end
```

```
class Snapshot
    operations
    getNext(): Snapshot = self.nextT.nextS
    futureClosure(s : Set(Snapshot)) : Set(Snapshot)= if s->
       includesAll(s.getNext()->asSet()) then s else
       futureClosure(s->union(s.getNext()->asSet()))endif
    getPost(): Set(Snapshot) = futureClosure(Set{self.getNext()})
    getPrevious(): Snapshot = self.beforeT.beforeS
    previousClosure(s : Set(Snapshot)) : Set(Snapshot)= if s->
       includesAll(s.getPrevious()->asSet()) then s else
       previousClosure(s->union(s.getPrevious()->asSet()))endif
    getPre(): Set(Snapshot) = previousClosure(Set{self.
       getPrevious()})
end

abstract class Transition

end

class PedestrianButton_requestPass < Transition
attributes
    PedButtonPre: PedestrianButton
    PedButtonPost:PedestrianButton
end

class TrafficLight_switchCarLight < Transition
```

```
  attributes
    TrafficLightPre:TrafficLight
    TrafficLightPost:TrafficLight
    ret: Color
end

class TrafficLight_switchPedLight < Transition
  attributes
    TrafficLightPre:TrafficLight
    TrafficLightPost:TrafficLight
    ret: Color
end

----------------------associations----------------

association PedestrianButtonTrafficLight between
  TrafficLight [1] role light
  PedestrianButton [1..*] role program
end



association BeforeTrans between
  Snapshot [1] role beforeS
  Transition [1] role nextT
end

association AfterTrans between
```

```
  Snapshot [0..1] role nextS
  Transition [0..1] role beforeT
end

composition SnapshotPedestrianButton between
 Snapshot [1] role snp
 PedestrianButton [1] role button
end

composition SnapshotTrafficLight between
 Snapshot [1] role snp
 TrafficLight [1] role light
end

--------------------constraints--------
constraints

--------------Transitions Invariants-------------
context PedestrianButton_requestPass
inv: self.PedButtonPre.light.pedLight=#Red
inv: self.PedButtonPost.light.requested= true

context TrafficLight_switchCarLight
inv: self.TrafficLightPost.carLight <> self.TrafficLightPre.
   carLight

context TrafficLight_switchPedLight
```

```
inv: self.TrafficLightPre.requested=true
inv: self.TrafficLightPost.pedLight <>  self.TrafficLightPre.
   pedLight
inv: self.TrafficLightPost.requested=false

---------STM Invariants--------------

context Snapshot inv AcyclicScenario: self.getPost()->excludes(
   self) and self.getPre()->excludes(self)

context Snapshot inv OneScenario: Snapshot.allInstances()->
   collect(s:Snapshot | s.getPrevious().oclIsUndefined())->size()
    = 1 and Snapshot.allInstances()->collect(s:Snapshot | s.
   getNext().oclIsUndefined())->size() = 1

context Transition inv SameTrans : Transition.allInstances()->
   forAll( t:Transition | (self.nextS = t.nextS and self.beforeS
   = t.beforeS) implies self = t)

context Snapshot inv SameSnapshot : Snapshot.allInstances()->
   forAll( s:Snapshot | (self.nextT = s.nextT and self.beforeT =
   s.beforeT) implies self = s)

context PedestrianButton_requestPass inv definedobject: not self.
   PedButtonPre.oclIsUndefined() and not self.PedButtonPost.
   oclIsUndefined()
```

```
context TrafficLight_switchCarLight inv definedobject: not self.
   TrafficLightPre.oclIsUndefined() and not self.TrafficLightPost
   .oclIsUndefined()

context TrafficLight_switchPedLight inv definedobject: not self.
   TrafficLightPre.oclIsUndefined() and not self.TrafficLightPost
   .oclIsUndefined()

context PedestrianButton_requestPass inv sameobjectID:  self.
   PedButtonPre.ID = self.PedButtonPost.ID

context TrafficLight_switchCarLight inv sameobjectID:  self.
   TrafficLightPre.ID = self.TrafficLightPost.ID

context TrafficLight_switchPedLight inv sameobjectID:  self.
   TrafficLightPre.ID = self.TrafficLightPost.ID

--===============The temporal properties======================

------ A perdestrians should be able to pass in next state if they
    request to pass in the current state
context TrafficLight
inv OCLTemporalProperty: let NextSnapshot:Snapshot = self.snp.
   getNext()
in self.requested=true implies NextSnapshot.light.pedLight=#Green

--========TP1===============
-- As soon as a traffic light is requested by a pedestrian, its
   car light turns red.

context TrafficLight
inv TP1: let CS: Snapshot = self.snp
 in let NS :Snapshot = CS.getNext()
in self.requested=true implies NS.light.carLight=#Red

--=========TP2================
-- After a pedestrian request to pass, the pedestrian light stays
    red until the car light turns to red.
context TrafficLight
inv TP2: let CS: Snapshot = self.snp
in let FS1 :Snapshot = CS.getPost()->select(sr:Snapshot | sr.
   light.carLight=#Red)->asOrderedSet()->first()
in let PreFS1: Set(Snapshot)= FS1.getPre()
in let CSPre: Set(Snapshot)= CS.getPre()->including(CS)
in let BTS: Set(Snapshot)= PreFS1->reject(s:Snapshot | CSPre->
   includes(s))
in self.requested=true implies BTS->forAll(s:Snapshot | s.light.
   pedLight=#Red)

---=========TP3================
--Before a pedestrian light turns to green, it must have been
   requested and the car light is red.

context  TrafficLight
```

```
inv TP3: let CS: Snapshot  = self.snp
in let PS: Set(Snapshot) = CS.getPre()
in  self.pedLight=#Green implies PS->exists(s:Snapshot | s.light.
   requested=true)

--=========TP4================
--Between the time when pedestrian request to pass and the time
   when the car light turns red, the pedestrian light must not be
    green. \\

context TrafficLight
inv TP4: let CS: Snapshot = self.snp
in let FS1 :Snapshot = CS.getPost()->select(sr:Snapshot | sr.
   light.carLight=#Red)->asOrderedSet()->first()
in let PreFS1: Set(Snapshot)= FS1.getPre()
in let CSPre: Set(Snapshot)= CS.getPre()->including(CS)
in let BTS: Set(Snapshot)= PreFS1->reject(s:Snapshot | CSPre->
   includes(s))
in self.requested=true implies BTS->collect(s:Snapshot | s.light.
   pedLight=#Green)->isEmpty()

--=========TP5================
--After pedestrian light becomes red, the cars are allowed to
   pass until a request is made by a pedestrian.

context TrafficLight
inv TP5: let CS: Snapshot = self.snp
```

```
in let FS : Set(Snapshot) = CS.getPost()
in let FSR1 :Snapshot = CS.getPost()->select(sr:Snapshot | sr.
   light.requested=true)->asOrderedSet()->first()
in let PreFS1: Set(Snapshot)= FSR1.getPre()
in let CSPre: Set(Snapshot)= CS.getPre()->including(CS)
in let BTS: Set(Snapshot)= PreFS1->reject(s:Snapshot | CSPre->
   includes(s))
in ((self.pedLight=#Red and FSR1.isDefined) implies BTS->forAll(s
   :Snapshot | s.light.carLight=#Green)) or ((self.pedLight=#Red
   and FSR1.oclIsUndefined()) implies FS->forAll(s:Snapshot | s.
   light.carLight=#Green))

--=========TP6================
--The car light and the pedestrian light can not be green
   simultaneously.

context TrafficLight
inv TP6: Snapshot.allInstances->forAll(s:Snapshot | not (s.light.
   pedLight=#Green and s.light.carLight=#Green))

--=========TP7================
--Before pedestrians can cross the street, the car light should
   become Red after the pedestrians request to pass. ---The
   pedestrians must request to pass before they are allowed.

context TrafficLight
inv TP7: let CS: Snapshot = self.snp
```

```
in let BS: Set(Snapshot) = CS.getPre()
in let PS: Set(Snapshot) = BS->select(ps:Snapshot | ps.light.
   requested=true)
in let SS: Set(Snapshot) = BS->select(ss:Snapshot | ss.light.
   carLight=#Red)
in self.carLight=#Green implies PS->forAll(ps:Snapshot | ps.
   getPost()->exists(ss:Snapshot| SS->includes(ss)))

--=========TP8================
--After the pedestrians light become red, and before it turns
   green again, the car light must become Red after pedestrians
   have to request passing again.

context TrafficLight
inv TP8: let CS: Snapshot = self.snp
in let FS : Set(Snapshot) = CS.getPost()
in let FSR1 :Snapshot = FS->select(sr:Snapshot | sr.light.
   carLight=#Green)->asOrderedSet()->first()
in let PreFS1: Set(Snapshot)= FSR1.getPre()
in let PS: Snapshot = FS->any(ps:Snapshot | ps.light.requested=
   true)
in let SS: Snapshot = FS->any(ss:Snapshot | ss.light.carLight=#
   Red)
in self.pedLight=#Red implies (PS.getPost()->includes(SS) and
   FSR1.getPre()->includes(PS)and FSR1.getPre()->includes(SS))

--=========End of USE Specification===========
```

A.2 Validation of The Snapshot Traversal Query Operations

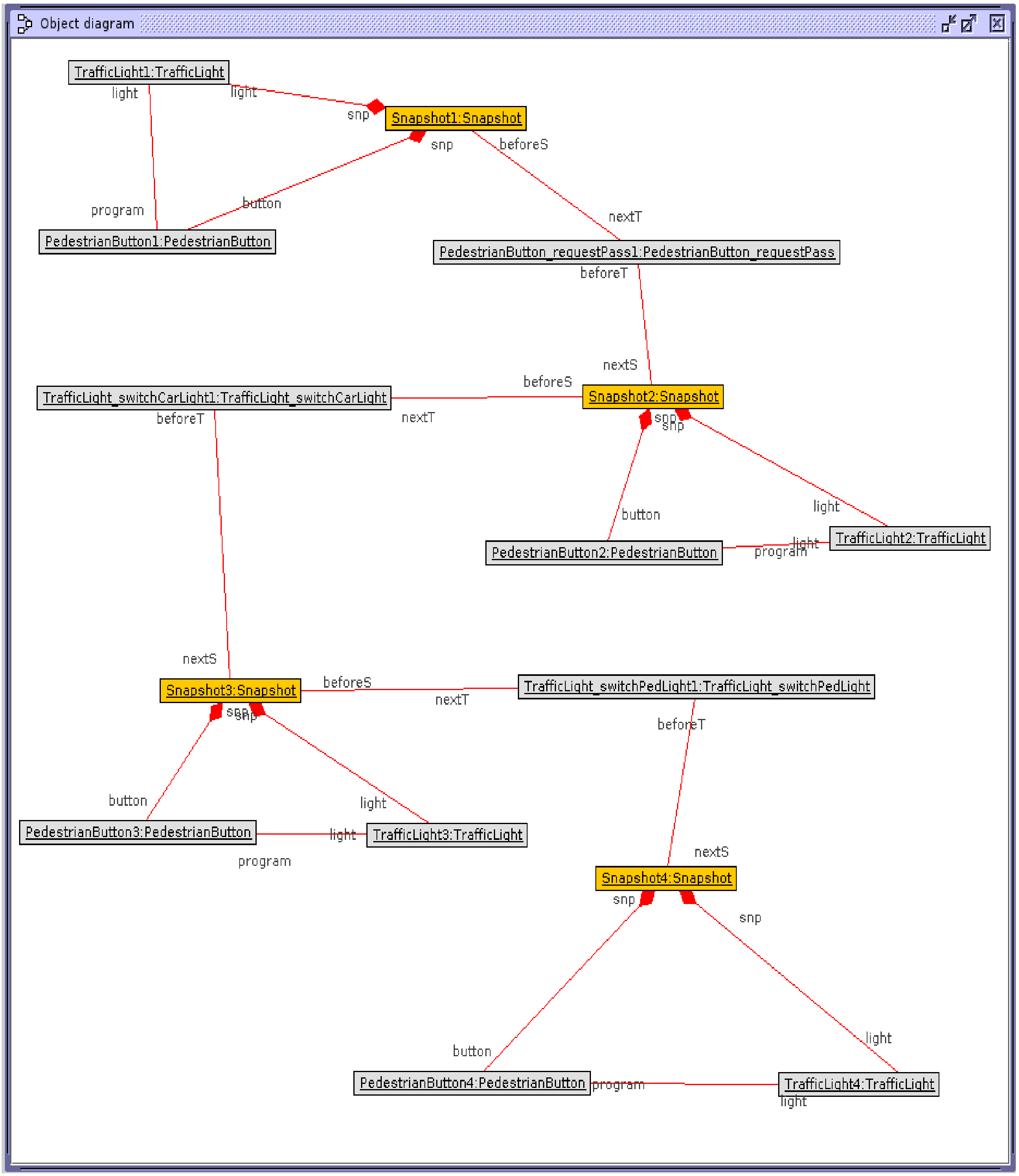

Object diagram
TrafficLight1:TrafficLight
light
light
snp
Snapshot1:Snapshot
snp
beforeS
program
button
PedestrianButton1:PedestrianButton
nextT
PedestrianButton_requestPass1:PedestrianButton_requestPass
beforeT
nextS
beforeS
TrafficLight_switchCarLight1:TrafficLight_switchCarLight
Snapshot2:Snapshot
beforeT
nextT
snp
snp
light
button
PedestrianButton2:PedestrianButton
program
light
TrafficLight2:TrafficLight
nextS
Snapshot3:Snapshot
beforeS
nextT
TrafficLight_switchPedLight1:TrafficLight_switchPedLight
snp
snp
beforeT
button
light
PedestrianButton3:PedestrianButton
light
TrafficLight3:TrafficLight
program
nextS
Snapshot4:Snapshot
snp
snp
button
light
PedestrianButton4:PedestrianButton
program
TrafficLight4:TrafficLight
light

Figure A.1: An example scenario, produced using USE Model Validator to show the correct functionality of the Snapshot Traversal Query Operations specified in Listing 3.1.

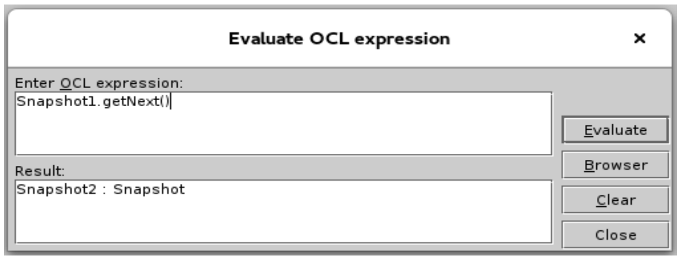

Enter OCL expression:
```
Snapshot1.getNext()
```
Result:
```
Snapshot2 : Snapshot
```

Figure A.2: The getNext() operation yields the expected result when invoked on Snapshot1 of the scenario depicted in Figure A.1

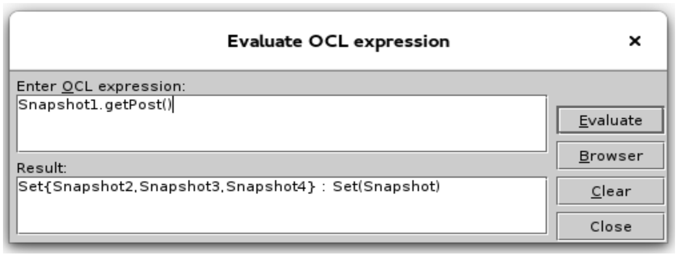

Enter OCL expression:
```
Snapshot1.getPost()
```
Result:
```
Set{Snapshot2,Snapshot3,Snapshot4} : Set(Snapshot)
```

Figure A.3: The getPost() operation yields the expected result when invoked on Snapshot1 of the scenario depicted in Figure A.1

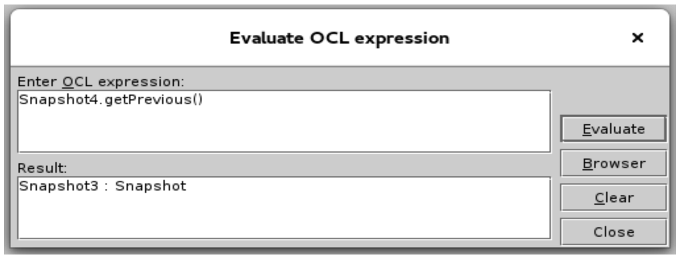

Enter OCL expression:
```
Snapshot4.getPrevious()
```
Result:
```
Snapshot3 : Snapshot
```

Figure A.4: The getPrevious() operation yields the expected result when invoked on Snapshot4 of the scenario depicted in Figure A.1

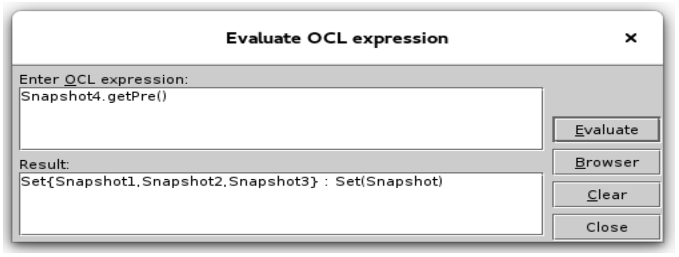

Enter OCL expression:
```
Snapshot4.getPre()
```
Result:
```
Set{Snapshot1,Snapshot2,Snapshot3} : Set(Snapshot)
```

Figure A.5: The getPre() operation yields the expected result when invoked on Snapshot4 of the scenario depicted in Figure A.1

Appendix B

Patterns' Specifications in TOCL and OCL

B.1 The Precedence Pattern

Table B.1: The precedence pattern specifications in TOCL and OCL

Scope	TOCL in ADCM	OCL in STM
Globally	context *[Class]* inv: *[P]* implies sometimePast *[S]*	context *[Class]* inv: let CS: Snapshot = self.getCurrentSnapshot(), in let PS: Set(Snapshot) = CS.getPre() in *[P]* implies PS → exists(s:Snapshot | *[s ⊨ S]*)
Before R	context Class inv: [R] implies sometimePast *[S]* since *[P]*	context Class inv: let CS: Snapshot = self.getCurrentSnapshot() in let BS: Set(Snapshot) = CS.getPre() in let PS: Set(Snapshot) = BS → select(s:Snapshot | [s ⊨ P]) in let SS: Set(Snapshot)= BS → select(s':Snapshot | *[s' ⊨ S]*) in *[R]* implies PS→forAll(ps:Snapshot | ps.getPre()→exists(ss:Snapshot | SS→includes(ss)))
After Q	context *[Class]* inv: *[Q]* implies always (*[P]* implies sometimePast *[S]*)	context Class inv: let CS: Snapshot = self.getCurrentSnapshot() in let FS: Set(Snapshot) = CS.getPost() in let PS: Set(Snapshot) = FS → select(s:Snapshot | [s ⊨ P]) in let SS: Set(Snapshot)= FS → select(s':Snapshot | *[s' ⊨ S]*) in *[R]* implies PS→forAll(ps:Snapshot | ps.getPre()→exists(ss:Snapshot | SS→includes(ss)))
After Q until R	context *[Class]* inv: *[Q]* implies always (*[P]* implies sometimePast*[S]*)	context *[Class]* inv: let CS: Snapshot = self.getCurrentSnapshot() in let BS: Set(Snapshot) = CS.getPre() in let PS: Snapshot= BS→select(s:Snapshot | [s ⊨ P]), SS: Snapshot= BS → select(s':Snapshot | [s' ⊨ S]) in *[Q]* implies BS→includes(PS) and BS→includes(SS) and PS.getPre() → includes(SS)
Between Q and R	context *[Class]* inv: *[Q]* implies (*[P]* implies sometimePast*[S]*) before [R]	context *[Class]* inv: let CS: Snapshot = self.getCurrentSnapshot() in let BS: Set(Snapshot) = CS.getPre() in let BSR1: Snapshot= BS→select(s:Snapshot | [s⊨ R]) → asOrderedSet()→ first(), in let PS: Snapshot= BS → any(s:Snapshot | [s ⊨ P]), SS: Snapshot= BS → any(s':Snapshot | [s' ⊨ S]) in *[Q]* implies (PS.getPost() → includes(SS) and FSR1.getPre() → includes(PS) and BSR1.getPre()→ includes(SS)

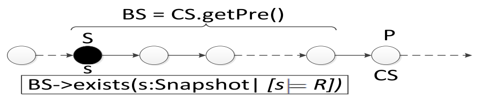


Figure B.1: Graphical illustration of the precedence pattern in the global scope (the first row in Table B.1).

B.2 The Absence Pattern

Table B.2: The absence pattern specifications in TOCL and OCL

Scope	TOCL in ADCM	OCL in STM
Globally	context *[Class]* inv: always not *[P]*	context *[Class]* inv: Snapshot.allInstances → collect(s:Snapshot | *[s ⊨ P]*)→isEmpty()
Before R	context *[Class]* inv: [R] implies alwaysPast not *[P]*	context *[Class]* inv: let CS: Snapshot = self.getCurrentSnapshot() in let BS: Set(Snapshot) = CS.getPre() in *[R]* implies BS → collect(s:Snapshot | *[s ⊨ P]*)→isEmpty()
After Q	context *[Class]* inv: *[Q]* implies always not *[P]*	context *[Class]* inv: let CS: Snapshot = self.getCurrentSnapshot() in let FS: Set(Snapshot) = CS.getPost() in *[Q]* implies FS → collect(s:Snapshot | *[s ⊨ P]*)→isEmpty()
After Q until R	context *[Class]* inv: *[Q]* implies always not *[P]* until [R]	context *[Class]* inv: let CS: Snapshot = self.getCurrentSnapshot() in let FSR1: Snapshot= CS.getPost()→ select(s:Snapshot | *[s⊨ R]*)→ asOrderedSet()→ first() in let PreFSR1=Set(Snapshot) = FSR1.getPre(), in let CSPre: Set(Snapshot)= CS.getPre()→including(CS) in let BTS: Set(Snapshot)= PreFS1→ reject(s:Snapshot | CSPre→includes(s)) in ((*[Q]* and FSR1.isDefined) implies BTS→collect(s:Snapshot | *[s ⊨ P]*)→isEmpty()) or ((*[Q]* and FSR1.oclIsUndefined()) implies FS→collect(s:Snapshot | *[s ⊨ P]*)→isEmpty())
Between Q and R	context *[Class]* inv:*[Q]* implies always not *[P]* until [R]	context *[Class]* inv: let CS: Snapshot = self.getCurrentSnapshot() in let FSR1: Snapshot= CS.getPost()→ select(s:Snapshot | [s⊨ R])→ asOrderedSet()→ first() in let PreFSR1=Set(Snapshot) = FSR1.getPre(), in let CSPre: Set(Snapshot)= CS.getPre()→including(CS) in let BTS: Set(Snapshot)= PreFS1→reject(s:Snapshot | CSPre→includes(s)) in *[Q]* implies BTS → collect(s:Snapshot | *[s ⊨ P]*)→isEmpty()

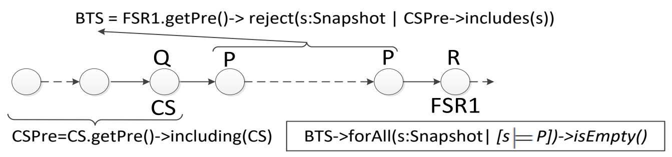

BTS = FSR1.getPre()-> reject(s:Snapshot | CSPre->includes(s))
Q
P
P
R
CS
FSR1
CSPre=CS.getPre()->including(CS)
BTS->forAll(s:Snapshot| [s ⊨ P])->isEmpty()

Figure B.2: Graphical illustration of the absence pattern in the global scope (the fifth row in Table B.2).

B.3 The Existence Pattern

Table B.3: The existence pattern specifications in TOCL and OCL

Scope	TOCL in ADCM	OCL in STM
Globally	context *[Class]* inv: sometime *[P]*	context *[Class]* inv: Snapshot.allInstances → exists(s:Snapshot | [s ⊨ P])
Before R	context *[Class]* inv: [R] implies alwaysPast *[P]*	context *[Class]* inv: let CS: Snapshot = self.getCurrentSnapshot() in let BS: Set(Snapshot) = CS.getPre() in *[R]* implies BS → exists(s:Snapshot | [s ⊨ P])
After Q	context *[Class]* inv: *[Q]* implies sometime *[P]*	context *[Class]* inv: let CS: Snapshot = self.getCurrentSnapshot() in let FS: Set(Snapshot) = CS.getPost() in *[Q]* implies FS → exists(s:Snapshot | *[s ⊨ P]*)
After Q until R	context *[Class]* inv: *[Q]* implies sometime *[P]* until [R]	context *[Class]* inv: let CS: Snapshot = self.getCurrentSnapshot() in let FSR1: Snapshot= CS.getPost()→ select(s:Snapshot | [s⊨ R])→ asOrderedSet()→ first() in let PreFSR1=Set(Snapshot) = FSR1.getPre(), in let CSPre: Set(Snapshot)= CS.getPre()→including(CS) in let BTS: Set(Snapshot)= PreFS1→ reject(s:Snapshot | CSPre→includes(s)) in ((*[Q]* and FSR1.isDefined) implies BTS→exists(s:Snapshot | *[s ⊨ P]*)) or ((*[Q]* and FSR1.oclIsUndefined()) implies FS→exists(s:Snapshot | *[s ⊨ P]*))
Between Q and R	context *[Class]* inv:*[Q]* implies sometime *[P]* until [R]	context *[Class]* inv: let CS: Snapshot = self.getCurrentSnapshot() in let FSR1: Snapshot= CS.getPost()→ select(s:Snapshot | [s⊨ R])→ asOrderedSet()→ first() in let PreFSR1=Set(Snapshot) = FSR1.getPre(), in let CSPre: Set(Snapshot)= CS.getPre()→including(CS) in let BTS: Set(Snapshot)= PreFS1→ reject(s:Snapshot | CSPre→includes(s)) in *[Q]* implies BTS → exists(s:Snapshot | *[s ⊨ P]*)

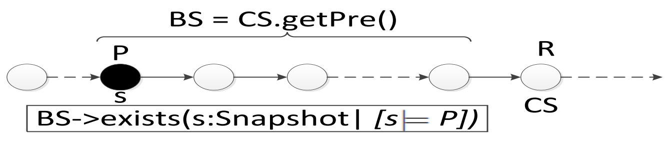


Figure B.3: Graphical illustration of the existence pattern in the global scope (the second row in Table B.3).

Appendix C

The Generalized Spatio-Temporal Role-Based Access Control Model

C.1 The USE Specification of GSTRBAC

```
model GSTRBAC
--***************************Classes ***************************
-- classes


class User
attributes
  name : String
operations
assignRole(r: Role, z:STZone): UserRoleAssignment
deassignRole(r:Role, z:STZone)
activateRole(r:Role, z:STZone): UserRoleActivation
deactivateRole(r:Role,z:STZone)
getAssignedRoles(z:STZone): Set(Role)= self.relations->select(r |
    r.oclIsTypeOf(UserRoleAssignment)and r.zone=z)-> collect( r|
   r.role)->asSet()
getActivatedRoles(z:STZone): Set(Role)= self.relations->select(r
   | r.oclIsTypeOf(UserRoleActivation)and r.zone=z)-> collect( r|
    r.role)->asSet()
```

```
getAuthorizedRoles(z:STZone): Set(Role)= self.getAssignedRoles(z)
    ->union(self.getAssignedRoles(z)->collect(r| r.
    getAllAHInheritedRoles(z))->asSet())
checkAccess(o:Object,a:Activity,z:STZone):Boolean =
    getActivatedRoles(z)->collect( r | r.getAuthorizedPermissions(
    z))->asSet()->exists( p | p.object=o and p.activity=a)
end



class STZone
end


class Role
operations
addAHJuniorRole(r:Role,z:STZone): A_Hierarchy
   deleteAHJuniorRole(r:Role,z:STZone)
   addIHJuniorRole(r:Role,z:STZone): I_Hierarchy
   deleteIHJuniorRole(r:Role,z:STZone)
   addSSoDRole(r:Role,z:STZone): RSSOD
   deleteSSoDRole(r:Role,z:STZone)
   addDSoDRole(r:Role,z:STZone): DSOD
   deleteDSoDRole(r:Role,z:STZone)
   assignPermission(p:Permission,z:STZone): PermissionAssignment
   deassignPermission(p:Permission,z:STZone)
   getSSoDRoles(z:STZone): Set(Role)= self.sod->select( s | s.zone
      =z and s.oclIsTypeOf(RSSOD))->collect(s | s.getInvolvedRoles
      ())->union(self.SOD->select( s | s.zone=z and s.oclIsTypeOf(
    RSSOD))->collect(s | s.getInvolvedRoles()))->excluding(self)
    ->asSet()
getDSoDRoles(z:STZone): Set(Role)= self.sod->select( s | s.zone=z
    and s.oclIsTypeOf(DSOD))->collect(s | s.getInvolvedRoles())->
   union(self.SOD->select( s | s.zone=z and s.oclIsTypeOf(DSOD))
   ->collect(s | s.getInvolvedRoles()))->excluding(self)->asSet()
getJuniorRoles(z:STZone): Set(Role) = RoleHierarchy.allInstances
   ->select(h| h.seniorRole=self and h.zone=z)->collect( rh | rh.
   juniorRole)->asSet()
getAHJuniorRoles(z:STZone): Set(Role)= A_Hierarchy.allInstances->
   select(ah | ah.seniorRole=self and ah.zone=z)->collect(ah1 |
   ah1.juniorRole)->asSet()
getIHJuniorRoles(z:STZone): Set(Role)= I_Hierarchy.allInstances->
   select(ah | ah.seniorRole=self and ah.zone=z)->collect(ah1 |
   ah1.juniorRole)->asSet()
getPrerequisiteRoles(): Set(Role) = self.prerequisiteRole->asSet
   ()
inherits(r:Role,z:STZone): Boolean = if (self.getJuniorRoles(z)->
   includes(r)) then true else self.getJuniorRoles(z)->exists(j |
    j.inherits(r,z))endif

inheritsAH(r:Role,z:STZone): Boolean = if (self.getAHJuniorRoles(
   z)->includes(r)) then true else self.getAHJuniorRoles(z)->
   exists(j | j.inheritsAH(r,z))endif
```

```
inheritsIH(r:Role,z:STZone): Boolean = if (self.getIHJuniorRoles(
   z)->includes(r)) then true else self.getIHJuniorRoles(z)->
   exists(j | j.inheritsIH(r,z))endif

getAllAHInheritedRoles(z:STZone): Set(Role)= Role.allInstances->
   select(r | self.inheritsAH(r,z))->asSet()

getAllIHInheritedRoles(z:STZone): Set(Role)= Role.allInstances->
   select(r | self.inheritsIH(r,z))->asSet()

getAssignedPermissions(z:STZone): Set(Permission)= self.permAssig
   ->select( pa| pa.zone=z )->collect( pa1| pa1.permission)->
   asSet()

getAuthorizedPermissions(z:STZone): Set(Permission)= self.
   getAssignedPermissions(z)->union(self.getAllIHInheritedRoles(z
   )->collect(r | r.getAssignedPermissions(z)))->asSet()

end

class Permission
operations
   addSoDPermission(p:Permission,z:STZone): PSSOD
   deleteSoDPermission(p:Permission,z:STZone)
   getSoDPermissions(z:STZone): Set(Permission)= self.pssod->
      select( s | s.zone=z)->collect(s | s.getInvolvedPermissions
        ())->union(self.PSSOD->select(s | s.zone=z)->collect(s | s.
        getInvolvedPermissions()))->excluding(self)->asSet()
    getPrerequisitePermissions(): Set(Permission) = self.
        prerequisitePermission->asSet()
end

class Object
end

class Activity
end

class Location
end

class TimeInterval
end

abstract class UserRoleRlation
end

class UserRoleAssignment < UserRoleRlation
end

class UserRoleActivation < UserRoleRlation
end
```

```
abstract class RoleHierarchy
end

class A_Hierarchy < RoleHierarchy
end

class I_Hierarchy < RoleHierarchy
end

abstract class SOD
  operations
   getInvolvedRoles(): Set(Role) = self.firstRole->including(self
      .secondRole)
end

class RSSOD < SOD
end

class DSOD < SOD
end

class PermissionAssignment
end

class PSSOD
operations
```

```
  getInvolvedPermissions(): Set(Permission) = self.
     firstPermission->including(self.secondPermission)
end

--*********************** Associations
   ****************************
-- associations

association URRUser between
  User[1] role user
  UserRoleRlation[*] role relations
end

association URRRole between
  Role[1] role role
  UserRoleRlation[*] role relations
end

association URRZone between
  STZone[1] role zone
  UserRoleRlation[*] role relations
end

aggregation ZoneLocation between
  STZone [1..*] role include
  Location [1] role location
end
```

```
aggregation ZoneTimeInterval between
  STZone [1..*] role include
  TimeInterval [1] role interval
end

association UserZone between
  User [*] role users
  STZone [1..*] role currentzones
end



association RoleZone between
  Role [1..*] role roles
  STZone [1..*] role allowedzones
end

association PermissionZone between
  Permission [1..*] role permissions
  STZone [1..*] role zones
end

association ObjectZone between
  Object [*] role objects
  STZone [1..*] role zones
end
```

```
aggregation PermissionObject between
  Permission[*] role permission
  Object[1] role object
end

aggregation PermissionActivity between
  Permission[*] role permission
  Activity[1] role activity
end


association RolePermZone between
  PermissionAssignment [*] role permAssig
  STZone[1] role zone
end

association SODZone between
  SOD [*] role sod
  STZone[1] role zone
end

association PSSODZone between
  PSSOD [*] role pssod
  STZone [1] role zone
end

association RHZone between
```

```
  RoleHierarchy [*] role rh
  STZone[1] role zone
end

association RH1Role between
  Role [1] role juniorRole
  RoleHierarchy[*] role RH
end

association RH2Role between
  Role [1] role seniorRole
  RoleHierarchy[*] role rh
end

association SOD1Role between
  Role[1] role firstRole
  SOD[*] role sod
end

association SOD2Role between
  Role[1] role secondRole
  SOD[*] role SOD
end

association PSSOD1Permission between
  Permission [1] role firstPermission
  PSSOD [*] role pssod
```

```
end

association PSSOD2Permission between
  Permission [1] role secondPermission
  PSSOD [*] role PSSOD
end

association PerAssToRole between
  Role[1] role role
  PermissionAssignment[*] role permAssig
end

association PerAssiToPermission between
  Permission[1] role permission
  PermissionAssignment[*] role PermAssig
end

association PrerequisiteRole between
  Role[*] role prerequisiteRole
  Role[*] role requistorRole
end

association PrerequisitePermission between
  Permission[*] role prerequisitePermission
  Permission[*] role requistorPermission
end
```

```
--************************************************************
   Constraints and Invariants
   ************************************************************
constraints

context RSSOD
inv SSOD_Constraint: not UserRoleAssignment.allInstances->exists(
   ura1,ura2 | ura1.user=ura2.user and ura1.role=self.firstRole
   and ura2.role=secondRole and ura1.zone=self.zone and ura2.zone
   =self.zone)

context User
inv SSOD_With_RH_Constraint: STZone.allInstances->forAll( z | not
    self.getAuthorizedRoles(z)->exists(r1,r2 | r1.getSSoDRoles(z)
   ->includes(r2)))

context User
inv Activation_Constraint_with_RH: self.currentzones->forAll(z|
   self.getAuthorizedRoles(z)->includesAll(self.getActivatedRoles
   (z)))



context Role
inv Permission_Inheritance_Constraint1: STZone.allInstances->
   forAll(z | self.getAuthorizedPermissions(z)->includesAll(self.
   getAllIHInheritedRoles(z)->collect(r | r.
   getAssignedPermissions(z))->asSet()))
```

```
inv Permission_Inheritance_Constraint2: STZone.allInstances->
   forAll(z | self.getAllIHInheritedRoles(z)->forAll( r | r.
   getAssignedPermissions(z)->intersection(self.
   getAuthorizedPermissions(z))=r.getAssignedPermissions(z)))



context User
inv Activation_Constraint: self.currentzones->forAll(z| self.
   getAssignedRoles(z)->includesAll(self.getActivatedRoles(z)))

context Role
 inv Hierarchy_Cycle_Constraint: not STZone.allInstances->exists(
    z| self.inherits(self,z))

context DSOD
inv DSOD_Constraint1: not UserRoleActivation.allInstances->exists
   (ura1,ura2 | ura1.user=ura2.user and ura1.role=self.firstRole
   and ura2.role=secondRole and ura1.zone=self.zone and ura2.zone
   =self.zone)

context User
  inv DSOD_Constaint2: STZone.allInstances->forAll( z | not self.
     getActivatedRoles(z)->exists(r1,r2 | r1.getDSoDRoles(z)->
     includes(r2)))

context PSSOD
```

```
inv PSOD_Constraint1: not PermissionAssignment.allInstances->
   exists(pa1,pa2 | pa1.role=pa2.role and pa1.permission=self.
   firstPermission and pa2.permission=self.secondPermission and
   pa1.zone=self.zone and pa2.zone=self.zone)

context Role
  inv PSOD_RH_Constaint: STZone.allInstances->forAll( z | not
     self.getAuthorizedPermissions(z)->exists(p1,p2 | p1.
     getSoDPermissions(z)->includes(p2)))

context User
inv Prerequist_URAssign: STZone.allInstances->forAll(z | Role.
   allInstances->forAll(r1 | (self.getAssignedRoles(z)->includes(
   r1)) implies (self.getAssignedRoles(z)->includesAll(r1.
   getPrerequisiteRoles()))))

context User
inv Prerequist_URActiv: STZone.allInstances->forAll(z | Role.
   allInstances->forAll(r1 | (self.getActivatedRoles(z)->includes
   (r1)) implies (self.getActivatedRoles(z)->includesAll(r1.
   getPrerequisiteRoles()))))

--**********************************Operations Specifications
   *************************************************

--*******************User Operations
   ******************************************************
```

```
context User::assignRole(r: Role, z:STZone): UserRoleAssignment
    pre assignRolePreCond1_definedObjects: r.isDefined and z.
        isDefined
    pre assignRolePreCond2_ZoneIncluded: self.currentzones->
        includes(z) and r.allowedzones->includes(z)
  pre assignRolePreCond3_RoleNotAssigned: self.getAssignedRoles(z
      )->excludes(r)
  pre assignRolePreCond4_RoleNotSSoD:  self.getAssignedRoles(z)->
      collect(r | r.getSSoDRoles(z))->excludes(r)
  post AssignSTRolePostCond1_NewUserRoleRelation: (self.relations
        - self.relations@pre)->size()=1
post AssignSTRolePostCond2_NewRoleAssignment: (self.relations -
   self.relations@pre)->forAll( r1 | r1.oclIsNew() and r1.
   oclIsTypeOf(UserRoleAssignment) and r1.zone=z and r1.role->
   includes(r))
post AssignSTRolePostCond3_RoleIsAssigned:self.getAssignedRoles(z
   )->includes(r)


context User::deassignRole(r:Role, z:STZone)
pre deassignRolePreCond1_RoleIsAssigned:self.getAssignedRoles(z)
   ->includes(r)
post deassignRolePostCond1_RoleDeassigned: self.getAssignedRoles(
   z)->excludes(r)
post deassignRolePostCond2_RoleAssignmentObjectDeleted:(self.
   relations@pre - self.relations)->size()=1 and (
   UserRoleAssignment.allInstances@pre - UserRoleAssignment.
   allInstances)->size()=1

context User::activateRole(r: Role, z:STZone): UserRoleActivation
   pre activateRolePreCond1_denfinedObject: r.isDefined and z.
       isDefined
pre activateRolePreCond2_ZoneIncluded: self.currentzones->
   includes(z) and r.allowedzones->includes(z)
  pre activateRolePreCond3_RoleNot: self.getActivatedRoles(z)->
     excludes(r)
pre activateRolePreCond4_RoleIsAssigned: getAssignedRoles(z)->
   includes(r)
post activateRolePostCond1_NewUserRoleRelation: (self.relations -
    self.relations@pre)->size()=1
post activateRolePostCond2_NewRoleActivation: (self.relations -
   self.relations@pre)->forAll( r1 | r1.oclIsNew() and r1.
   oclIsTypeOf(UserRoleActivation) and r1.zone=z and r1.role->
   includes(r))
post activateRolePostCond3_RoleIsAssigned:self.getActivatedRoles(
   z)->includes(r)

context User::deactivateRole(r:Role,z:STZone)
pre deactivateRolePreCond1_RoleIsActivated: self.
   getActivatedRoles(z)->includes(r)
post deactivateRolePostCond1_RoleDeactivated: self.
   getActivatedRoles(z)->excludes(z)
```

```
post deactivateRolePostCond2_RoleActivationDeleted: (self.
    relations@pre - self.relations)->size()=1 and (
    UserRoleActivation.allInstances@pre - UserRoleActivation.
    allInstances)->size()=1

--********************************************************Role
    Operations
    ********************************************************

context Role::addAHJuniorRole(r:Role,z:STZone): A_Hierarchy
pre addAHJuniorRolePreCond1_definedObjects:r.isDefined and z.
    isDefined
pre addAHJuniorRolePreCond2_ZoneIncluded: self.allowedzones->
    includes(z) and r.allowedzones->includes(z)
pre addAHJuniorRolePreCond3_NotAHJuniorRole: getAHJuniorRoles(z)
    ->excludes(r)
post addAhJuniorRolePostCond1_NewRoleHierarchy: (self.rh - self.
    rh@pre)->size()=1 and (self.RH - self.RH@pre)->size()=1
post addAhJuniorRolePostCond2_NewRoleA_Hierarchy: (self.rh - self
    .rh@pre)->forAll( rh | rh.oclIsNew() and rh.oclIsTypeOf(
    A_Hierarchy) and rh.zone=z and rh.juniorRole=r and rh.
    seniorRole=self)
post addAhJuniorRolePostCond3_RoleIsAdded:self.getAHJuniorRoles(z
    )->includes(r)

context Role::deleteAHJuniorRole(r:Role, z:STZone)
```

```
pre deleteAHJuniorRolePreCond1_RoleIsJuniorRole:self.
   getAHJuniorRoles(z)->includes(r)
post deleteAHJuniorRolePostCond1_RoleDeleated: self.
   getAHJuniorRoles(z)->excludes(r)
post deleteAHJuniorRolePostCond2_AHierarchyObjectDeleted:(self.
   rh@pre - self.rh)->size()=1 and (self.RH@pre - self.RH)->size
   ()=1 and (A_Hierarchy.allInstances@pre - A_Hierarchy.
   allInstances)->size()=1

context Role::addIHJuniorRole(r:Role,z:STZone): I_Hierarchy
pre addIHJuniorRolePreCond1_definedObjects:r.isDefined and z.
   isDefined
pre addIHJuniorRolePreCond2_ZoneIncluded: self.allowedzones->
   includes(z) and r.allowedzones->includes(z)
pre addIHJuniorRolePreCond3_NotIHJuniorRole: getIHJuniorRoles(z)
   ->excludes(r)
post addIHJuniorRolePostCond1_NewRoleHierarchy: (self.rh - self.
   rh@pre)->size()=1 and (self.RH - self.RH@pre)->size()=1
post addIHJuniorRolePostCond2_NewRoleI_Hierarchy: (self.rh - self
   .rh@pre)->forAll( rh | rh.oclIsNew() and rh.oclIsTypeOf(
   I_Hierarchy) and rh.zone=z and rh.juniorRole=r and rh.
   seniorRole=self)
post addIHJuniorRolePostCond3_RoleIsAdded:self.getIHJuniorRoles(z
   )->includes(r)

context Role::deleteIHJuniorRole(r:Role, z:STZone)
```

```
pre deleteIHJuniorRolePreCond1_RoleIsJuniorRole:self.
   getIHJuniorRoles(z)->includes(r)
post deleteIHJuniorRolePostCond1_RoleDeleated: self.
   getIHJuniorRoles(z)->excludes(r)
post deleteIHJuniorRolePostCond2_IHierarchyObjectDeleted:(self.
   rh@pre - self.rh)->size()=1 and (self.RH@pre - self.RH)->size
   ()=1 and (I_Hierarchy.allInstances@pre - I_Hierarchy.
   allInstances)->size()=1

context Role::addSSoDRole(r:Role,z:STZone): RSSOD
pre addSSoDRolePreCond1_definedObjects:r.isDefined and z.
   isDefined
pre addSSoDRolePreCond2_ZoneIncluded: self.allowedzones->includes
   (z) and r.allowedzones->includes(z)
pre addSSoDRolePreCond3_NotSSODRole: getSSoDRoles(z)->excludes(r)
post addSSoDRoleRolePostCond1_NewSOD: (self.sod - self.sod@pre)->
   size()=1 and (self.SOD - self.SOD@pre)->size()=1
post addSSoDRoleRolePostCond2_NewSSOD: (self.sod - self.sod@pre)
   ->forAll( sod | sod.oclIsNew() and sod.oclIsTypeOf(RSSOD) and
   sod.zone=z and sod.firstRole=r and sod.secondRole=self)
post addSSoDRoleRolePostCond3_RoleIsAdded:self.getSSoDRoles(z)->
   includes(r)

context Role::deleteSSoDRole(r:Role,z:STZone)
pre deleteSSoDRolePreCond1_RoleIsSSoDRole:self.getSSoDRoles(z)->
   includes(r)
```

```
post deleteSSoDRolePostCond1_RoleDeleated: self.getSSoDRoles(z)->
    excludes(r)
post deleteSSoDRolePostCond2_SSODObjectDeleted:(self.sod@pre -
    self.sod)->size()=1 and (self.SOD@pre - self.SOD)->size()=1
    and (RSSOD.allInstances@pre - RSSOD.allInstances)->size()=1

context Role::addDSoDRole(r:Role,z:STZone): DSOD
pre addDSoDRolePreCond1_definedObjects:r.isDefined and z.
    isDefined
pre addDSoDRolePreCond2_ZoneIncluded: self.allowedzones->includes
    (z) and r.allowedzones->includes(z)
pre addDSoDRolePreCond3_NotDSODRole: getDSoDRoles(z)->excludes(r)
post addDSoDRoleRolePostCond1_NewSOD: (self.sod - self.sod@pre)->
    size()=1 and (self.SOD - self.SOD@pre)->size()=1
post addDSoDRoleRolePostCond2_NewDSOD: (self.sod - self.sod@pre)
    ->forAll( sod | sod.oclIsNew() and sod.oclIsTypeOf(DSOD) and
    sod.zone=z and sod.firstRole=r and sod.secondRole=self)
post addDSoDRoleRolePostCond3_RoleIsAdded:self.getDSoDRoles(z)->
    includes(r)

context Role::deleteDSoDRole(r:Role,z:STZone)
pre deleteDSoDRolePreCond1_RoleIsSSoDRole:self.getDSoDRoles(z)->
    includes(r)
post deleteDSoDRolePostCond1_RoleDeleated: self.getDSoDRoles(z)->
    excludes(r)
```

```
post deleteDSoDRolePostCond2_DSODObjectDeleted:(self.sod@pre -
   self.sod)->size()=1 and (self.SOD@pre - self.SOD)->size()=1
   and (DSOD.allInstances@pre - DSOD.allInstances)->size()=1

context Role::assignPermission(p:Permission,z:STZone):
   PermissionAssignment
pre assignPermissionPreCond1_definedObjects: p.isDefined and z.
   isDefined
pre assignPermissionCond2_ZoneIncluded: p.zones->includes(z) and
   self.allowedzones->includes(z)
pre assignPermissionPreCond3_PermissionNotAssigned: self.
   getAssignedPermissions(z)->excludes(p)
pre assignPermissionPreCond4_PermissionNotSSoD: self.
   getAssignedPermissions(z)->collect(per | per.getSoDPermissions
   (z))->excludes(p)
post assignPermissionPostCond1_NewPermissionAssignment: (self.
   permAssig - self.permAssig@pre)->size()=1
post assignPermissionPostCond2_NewRoleAssignment: (self.permAssig
    - self.permAssig@pre)->forAll( pa | pa.oclIsNew() and pa.zone
   =z and pa.permission->includes(p))
post assignPermissionPostCond3_PermissionIsAssigned:self.
   getAssignedPermissions(z)->includes(p)


context Role::deassignPermission(p:Permission,z:STZone)
pre deassignPermissionPreCond1_PermissionIsAssigned:self.
   getAssignedPermissions(z)->includes(p)
```

```
post deassignPermissionPostCond1_PermissionDeassigned: self.
   getAssignedPermissions(z)->excludes(p)
post
   deassignPermissionPostCond2_PermissionAssignmentObjectDeleted
   :(self.permAssig@pre - self.permAssig)->size()=1 and (
   PermissionAssignment.allInstances@pre - PermissionAssignment.
   allInstances)->size()=1

--*****************************************************
   Permission Operations
   *****************************************************

context Permission::addSoDPermission(p:Permission ,z:STZone):
   PSSOD
pre addSoDPermissionPreCond1_definedObjects:p.isDefined and z.
   isDefined
pre addSoDPermissionPreCond2_ZoneIncluded: self.zones->includes(z
   ) and p.zones->includes(z)
pre addSoDPermissionCond3_NotSSODPermission: getSoDPermissions(z)
   ->excludes(p)
post addSoDPermissionPostCond1_NewSOD: (self.pssod - self.
   pssod@pre)->size()=1 and (self.PSSOD - self.PSSOD@pre)->size()
   =1
post addSoDPermissionPostCond2_NewPSOD: (self.pssod - self.
   pssod@pre)->forAll( pssod | pssod.oclIsNew() and pssod.zone=z
   and pssod.firstPermission=p and pssod.secondPermission=self)
```

```
post addSoDPermissionPostCond3_PermissionIsAdded:self.
   getSoDPermissions(z)->includes(p)


context Permission::deleteSoDPermission(p:Permission,z:STZone)
pre deleteSoDPermissionPreCond1_PermissionIsSSoDRole:self.
   getSoDPermissions(z)->includes(p)
post deleteSoDPermissionPostCond1_RoleDeleated: self.
   getSoDPermissions(z)->excludes(p)
post deleteSoDPermissionPostCond2_PSSODObjectDeleted:(self.
   pssod@pre - self.pssod)->size()=1 and (self.PSSOD@pre - self.
   PSSOD)->size()=1  and (PSSOD.allInstances@pre - PSSOD.
   allInstances)->size()=1


--=========End of USE Specification============
```

C.2 The Temporal Properties of The GSTRBAC System

Table C.1: Temporal properties of the GSTRBAC model, GSTRBAC-TP1 to GSTRBAC-TP16

No.	Description	Pattern - Scope
GSTRBAC-TP1	If a role is available in a particular zone, the role should eventually be assigned to a user in that zone.	Response-Globally
GSTRBAC-TP2	When a user activates a role in a zone, the role remains active until the user moves to a different zone.	Universality-Between Q and R
GSTRBAC-TP3	In a spatio-temporal zone, a user can only activate roles that were previously assigned to the user in the zone	Precedence-Globally
GSTRBAC-TP4	A role must be enabled in a spatio-temporal zone before it is assigned to a user in that zone.	Precedence-Globally
GSTRBAC-TP5	A user should not be assigned to two conflicting roles in the same zone.	Absence-Globally
GSTRBAC-TP6	A role can not be assigned to two conflicting permissions in the same zone.	Absence-Globally
GSTRBAC-TP7	A role must be assigned to a less critical role in a given spatio-temporal zone before being assigned more critical roles.	Precedence-Globally
GSTRBAC-TP8	A role can be assigned a permission in a specific zone if some prerequisite permissions are already assigned to that role in the same zone.	Precedence-Globally
GSTRBAC-TP9	If a a user is assigned to a role in paricular zone, the user will eventually activate that role in the zone.	Response-Globally
GSTRBAC-TP10	A user who is assigned to a senior role will be able to use the premissions assigned to any of th ejunior roles.	Response-Globally
GSTRBAC-TP11	When a role is made as a senior role to another role in a zone, the user who is assign to the senior role can activate the junior role in that zone.	Response-Globally
GSTRBAC-TP12	A role will be used to get access to resources using its assigned permissions at a particular zone.	Response-Globally
GSTRBAC-TP13	After a user is assigned a role in a particular zone, the user will not be able to be assigned to any of the conflicting roles	Absence-After Q
GSTRBAC-TP14	A role can not be assigned to two conflicting permissions at a particular zone.	Absence-Globally
GSTRBAC-TP15	After a user is assigned to two conflicting roles dynamically, the user can only activate one of them in a particular zone.	Absence-After
GSTRBAC-TP16	When a user is deassigned from a role in a particular zone, the user will not be able to activate the role.	Response-After

Table C.2: TOCL and OCL specification of the GSTRBAC temporal properties described in Table C.1

No.	TOCL Specification on Class Model	OCL Specification on the Snapshot Transition Model
GSTRBAC-TP1	context r:Role inv GSTRBAC-TP1: r.getAvailableZones()→ includes(z:STZone) implies sometime r.getAssignedUsers(z) → notEmpty()	context r:Role inv GSTRBAC-TP1: let CS: Snapshot= self.getCurrentSnapshot() in let FS: Set(Snapshot)= CS.gePost() in r.getAvailableZones()→ includes(z:STZone) implies FS → exists (s:Snapshot | s.r.getAssignedUsers(z) → notEmpty())
GSTRBAC-TP2	context u:User inv GSTRBAC-TP2: u.getActivatedRoles(u.currentzone)→ includes(r:Role) implies always u.getActivatedRoles(u.currentzone) → includes(r) until u.zonechanged= True	context u:User inv GSTRBAC-TP2: let CS: Snapshot = self.getCurrentSnapshot() in let FSR1 :Snapshot = CS.getPost()→ select(sr:Snapshot | sr.u.zonechanged= True)→asOrderedSet()→first() in let PreFSR1: Set(Snapshot)= FSR1.getPre() in let CSPre: Set(Snapshot)= CS.getPre()→including(CS) in let BTS: Set(Snapshot)= PreFSR1→reject(s:Snapshot | CSPre→includes(s)) in u.getActivatedRoles(u.currentzone)→ includes(r:Role) implies BTS→forAll(s:Snapshot | s.u.getActivatedRoles(u.currentzone) → includes(r))
GSTRBAC-TP3	context u:User inv GSTRBAC-TP3: u.getActivatedRoles(z:STZone)→ includes(r1:Role) implies sometimePast u.getAssignedRoles(z)→ includes(r1)	context u:User inv GSTRBAC-TP3: let CS: Snapshot = self.getCurrentSnapshot() in let PS: Set(Snapshot) = CS.getPre() in u.getActivatedRoles(z:STZone)→ includes(r1:Role) implies PS → exists (s:Snapshot | s.u.getAssignedRoles(z)→ includes(r1))
GSTRBAC-TP4	context r:Role inv GSTRBAC-TP4: r.assignedUsers(z:STZone)→ notEmpty() implies sometimePast (z.enabledRoles()→ includes(r))	context r:Role inv GSTRBAC-TP4: let CS: Snapshot = self.getCurrentSnapshot(), in let PS: Set(Snapshot) = CS.getPre() in r.assignedUsers(z:STZone)→ notEmpty() implies PS → exists(s:Snapshot | (s.z.enabledRoles()→ includes(r))
GSTRBAC-TP5	context u:User inv GSTRBAC-TP5: always not u.getAssignedRoles(z) → (r1,r2:Role | r1.getSSoDRoles(z)→(r2))	context u:User inv GSTRBAC-TP5: Snapshot.allInstances→forAll(s:Snapshot | not s.u.getAssignedRoles(z) → (r1,r2:Role | r1.getSSoDRoles(z)→(r2))
GSTRBAC-TP6	context r:Role inv GSTRBAC-TP6: always not r.getAssignedPersmissions(z) → (p1,p2:Permission | p1.getPSSoDPermissions(z)→(p2))	context r:Role inv GSTRBAC-TP6: Snapshot.allInstances→forAll(s:Snapshot | not s.r.getAssignedPersmissions(z) → (p1,p2:Permission | p1.getPSSoDPermissions(z)→(p2))
GSTRBAC-TP7	context u:User inv GSTRBAC-TP7: u.getAssignedRoles()→includes(r1) implies sometimePast (u.getAssignedRoles(z)→ includesAll(r1.getPreqAssRoles()))	context u:User inv GSTRBAC-TP7: let CS: Snapshot = self.getCurrentSnapshot(), in let PS: Set(Snapshot) = CS.getPre() in u.getAssignedRoles()→includes(r1) implies PS → exists(s:Snapshot | s.u.getAssignedRoles(z)→ includesAll(r1.getPreqAssRoles()))
GSTRBAC-TP8	context r:Role inv GSTRBAC-TP8: r.getAssignedPermissions(z)→includes(p1) implies sometimePast (r.getAssignedPermissions(z)→ includesAll (p1.getPrerequisitePermissions()))	context r:Role inv GSTRBAC-TP8: let CS: Snapshot = self.getCurrentSnapshot(), in let PS: Set(Snapshot) = CS.getPre() in r.getAssignedPermissions(z)→includes(p1) implies PS → exists(s:Snapshot | s.r.getAssignedPermissions(z)→ includesAll(p1.getPrerequisitePermissions()))

Table C.3: TOCL and OCL specification of the GSTRBAC temporal properties described in Table C.1

No.	TOCL Specification on Class Model	OCL Specification on the Snapshot Transition Model
GSTRBAC-TP9	context u:User inv GSTRBAC-TP9: u.getAssignedRoles(z)→includes(r) implies sometime u.getActivatedRoles(z)→(r)	context u:User inv GSTRBAC-TP9: let CS: Snapshot = self.getCurrentSnapshot(), in let FS: Set(Snapshot) = CS.getPost() in u.getAssignedRoles(z)→includes(r) implies FS → exists(s:Snapshot | s.u.getActivatedRoles(z)→(r))
GSTRBAC-TP10	context u:User inv GSTRBAC-TP10: u.getAssignedRoles(z)→(r) implies sometime u.getAuthorizedPermissions(z)→includesAll(p:Permission | r.getJuniorRoles(z)→includes(r2) and r2.getAssignedPermissions(z)→includes(p))	context u:User inv GSTRBAC-TP10: let CS: Snapshot = self.getCurrentSnapshot(), in let FS: Set(Snapshot) = CS.getPost() in u.getAssignedRoles(z)→(r) implies FS → exists(s:Snapshot | s.u.getAuthorizedPermissions(z)→includesAll(p:Permission | r.getJuniorRoles(z)→includes(r2) and r2.getAssignedPermissions(z)→includes(p)))
GSTRBAC-TP11	context r1,r2:Role inv GSTRBAC-TP11: r1.getAHJuniorRoles(z)→includes(r2) implies sometime r1.getAssignedUsers(z)→forAll(u:User | u.getActivatedRoles(z)→includes(r2))	context u:User inv GSTRBAC-TP11: let CS: Snapshot = self.getCurrentSnapshot(), in let FS: Set(Snapshot) = CS.getPost() in r1.getAHJuniorRoles(z)→includes(r2) implies FS → exists(s:Snapshot | s.r1.getAssignedUsers(z)→forAll(u:User | u.getActivatedRoles(z)→includes(r2))
GSTRBAC-TP12	context r:Role inv GSTRBAC-TP12: r.getAssignedPermissions(z)→includes(p) implies sometime r1.getAssignedUsers(z)→includes(u:User | u.checkAccess(o,a,z)=True and p.object=o and p.activity=a)	context r:Role inv GSTRBAC-TP12: let CS: Snapshot = self.getCurrentSnapshot(), in let FS: Set(Snapshot) = CS.getPost() in r.getAssignedPermissions(z)→includes(p) implies FS → exists(s:Snapshot | sr1.getAssignedUsers(z)→includes(u:User | u.checkAccess(o,a,z)=True and p.object=o and p.activity=a))
GSTRBAC-TP13	context u:User inv GSTRBAC-TP13: u.getAssignedRoles(z)→includes(r1) implies always not u.getAssignedRoles(z)→includes(r2:Role | r1.getSSoDRoles(z)→includes(r2))	context u:User inv GSTRBAC-TP13: let CS: Snapshot = self.getCurrentSnapshot() in let FS: Set(Snapshot) = CS.getPost() in u.getAssignedRoles(z)→includes(r1) implies FS → collect(s:Snapshot | s.u.getAssignedRoles(z)→includes(r2:Role | r1.getSSoDRoles(z)→includes(r2)))→ isEmpty()
GSTRBAC-TP14	context r:Role inv GSTRBAC-TP14: always not r.getAssignedPermissions(z)→includes(p1,p2 | p1.getSoDPermissions(z)→includes(p2))	context u:User inv GSTRBAC-TP14: Snapshot.allInstances → collect(s:Snapshot | s.r.getAssignedPermissions(z)→ includes(p1,p2 | p1.getSoDPermissions(z)→includes(p2)))→isEmpty()
GSTRBAC-TP15	context u:User inv GSTRBAC-TP15: (u.getAssignedRoles(z)→includes(r1) and u.getAssignedRoles(z)→includes(r2) and r1.getDSoDRoles(z)→includes(r2)) implies always not(u.getActivatedRoles(z)→includes(r1) and u.getActivatedRoles(z)→includes(r2))	context u:User inv GSTRBAC-TP15: let CS: Snapshot = self.getCurrentSnapshot(), in let FS: Set(Snapshot) = CS.getPost() in (u.getAssignedRoles(z)→includes(r1) and u.getAssignedRoles(z)→includes(r2) and r1.getDSoDRoles(z)→includes(r2)) implies FS → collect(s:Snapshot | s.u.getActivatedRoles(z)→includes(r1) and s.u.getActivatedRoles(z)→includes(r2)))→isEmpty()
GSTRBAC-TP16	context u:User inv GSTRBAC-TP16: deactivateRole(r,z)=True implies always (u.getActivatedRoles(z)→excludes(r))	context u:User inv GSTRBAC-TP16: let CS: Snapshot = self.getCurrentSnapshot(), in let FS: Set(Snapshot) = CS.getPost() in deactivateRole(r,z)=True implies FS →forAll(s:Snapshot | s.u.getActivatedRoles(z)→excludes(r))

Table C.4: Temporal properties of the GSTRBAC model, GSTRBAC-TP17 to GSTRBAC-TP32

No.	Description	Pattern - Scope
GSTRBAC-TP17	When a user enters a zone, the user can only activate the assigned roles in that zone	Response-Globally
GSTRBAC-TP18	When a user leaves a zone, the user's assigned roles in that zone does not change.	Response-Globally
GSTRBAC-TP19	Once an object is available in a zone, users can access that object in that zone.	Response-Globally
GSTRBAC-TP20	When a user deactivate a senior role, the user can not use the permission of the junior roles to gain access.	Response-Globally
GSTRBAC-TP21	When a user is assigned to a role, the user can not be assigned to anly of the conflicting roles in that zone.	Response-Globally
GSTRBAC-TP22	When a role is assigned to a permission in particular zone, the role ca not be assigned to any of the conflicting permissions.	Response-Globally
GSTRBAC-TP23	When an object is assigned to a particular zone, the object can be accessed in that zone.	Response-Globally
GSTRBAC-TP24	When a permission is associate to a zone, it can be assigned to the roles that are enabled in the same zone.	Response-Globally
GSTRBAC-TP25	When a role is assigned to a permission in a particular zone, the role can be used to access the objects associated with the permission only if the object is available in the zone.	Response-Globally
GSTRBAC-TP26	When a user is deassigned from a role, the use can then be assigned to any of the role's conflicting roles in the a zone.	Response-Globally
GSTRBAC-TP27	When a role is added as a junior role to another, the junior role can not inherits permissions from a senior role through the role hierarchy.	Response-Globally
GSTRBAC-TP28	When a role is added as a junior role to another role through activation hierarchy, the user assigne to junior role can not activate the senior role through the hierarchy.	Response-Globally
GSTRBAC-TP29	When a user changes a spatio-temporal zone, all the activated roles get deactivated in the next state.	Response-Globally

Table C.5: TOCL and OCL specification of the GSTRBAC temporal properties described in Table C.4

No.	TOCL Specification on Class Model	OCL Specification on the Snapshot Transition Model
GSTRBAC-TP17	context u:User inv GSTRBAC-TP17: u.zonechanged implies sometime u.getActivatedRoles(u.currentzone)→ excludesAll(r:Role | r.rzones→excludes(u.currentzone))	context u:User inv GSTRBAC-TP17: let CS: Snapshot = self.getCurrentSnapshot(), in let FS: Set(Snapshot) = CS.getPost() in u.zonechanged implies FS → exists(s:Snapshot | su.getActivatedRoles(u.currentzone)→ excludesAll(r:Role | r.rzones→excludes(u.currentzone)))
GSTRBAC-TP18	context u:User inv GSTRBAC-TP18: let AssignRoles: Set(Role) = u.getAssignedRoles(u.currentzone) in u.zonechanged implies next u.getAssignedRoles(u.currentzone) →includesAll(AssignRoles)	context u:User inv GSTRBAC-TP18: inv: let CS: Snapshot = self.getCurrentSnapshot(), in let NS: Snapshot = CS.getNext() in let AssignRoles: Set(Role) = u.getAssignedRoles(u.currentzone) in u.zonechanged implies NS.u.getAssignedRoles(u.currentzone)→ includesAll(AssignRoles)
GSTRBAC-TP19	context u:User, o:Object, a:Activity inv GSTRBAC-TP19: o.ozones→includes(z:STZone) implies sometime u.checkAccess(o,a,z)=True	context u:User, o:Object, a:Activity inv GSTRBAC-TP19: let CS: Snapshot = self.getCurrentSnapshot() in let FS: Set(Snapshot) = CS.getPost() o.ozones→includes(z:STZone) implies FS →exists(s:Snapshot | s.u.checkAccess(o,a,z)=True)
GSTRBAC-TP20	context r:Role inv GSTRBAC-TP20: inv: *[P]* implies sometime *[S]*	context r:Role inv GSTRBAC-TP20: inv: let CS: Snapshot = self.getCurrentSnapshot(), in let FS: Set(Snapshot) = CS.getPost() in *[P]* implies FS → exists(s:Snapshot | $[s \models S]$)
GSTRBAC-TP21	context r:Role inv GSTRBAC-TP21: inv: *[P]* implies sometime *[S]*	context r:Role inv GSTRBAC-TP21: inv: let CS: Snapshot = self.getCurrentSnapshot(), in let FS: Set(Snapshot) = CS.getPost() in *[P]* implies FS → exists(s:Snapshot | $[s \models S]$)
GSTRBAC-TP22	context r:Role inv GSTRBAC-TP22: inv: *[P]* implies sometime *[S]*	context r:Role inv GSTRBAC-TP22: inv: let CS: Snapshot = self.getCurrentSnapshot(), in let FS: Set(Snapshot) = CS.getPost() in *[P]* implies FS → exists(s:Snapshot | $[s \models S]$)
GSTRBAC-TP23	context r:Role inv GSTRBAC-TP23: inv: *[P]* implies sometime *[S]*	context r:Role inv GSTRBAC-TP23: inv: let CS: Snapshot = self.getCurrentSnapshot(), in let FS: Set(Snapshot) = CS.getPost() in *[P]* implies FS → exists(s:Snapshot | $[s \models S]$)
GSTRBAC-TP24	context r:Role inv GSTRBAC-TP24: inv: *[P]* implies sometime *[S]*	context r:Role inv GSTRBAC-TP24: inv: let CS: Snapshot = self.getCurrentSnapshot(), in let FS: Set(Snapshot) = CS.getPost() in *[P]* implies FS → exists(s:Snapshot | $[s \models S]$)

Table C.6: TOCL and OCL specification of the GSTRBAC temporal properties described in Table C.4

No.	TOCL Specification on Class Model	OCL Specification on the Snapshot Transition Model
GSTRBAC-TP25	context r:Role inv GSTRBAC-TP25: inv: *[P]* implies sometime *[S]*	context r:Role inv GSTRBAC-TP25: inv: let CS: Snapshot = self.getCurrentSnapshot(), in let FS: Set(Snapshot) = CS.getPost() in *[P]* implies FS $\rightarrow$ exists(s:Snapshot | *[s $\models$ S]*)
GSTRBAC-TP26	context r:Role inv GSTRBAC-TP26: inv: *[P]* implies sometime *[S]*	context r:Role inv GSTRBAC-TP26: inv: let CS: Snapshot = self.getCurrentSnapshot(), in let FS: Set(Snapshot) = CS.getPost() in *[P]* implies FS $\rightarrow$ exists(s:Snapshot | *[s $\models$ S]*)
GSTRBAC-TP27	context r:Role inv GSTRBAC-TP27: inv: *[P]* implies sometime *[S]*	context r:Role inv GSTRBAC-TP27: inv: let CS: Snapshot = self.getCurrentSnapshot(), in let FS: Set(Snapshot) = CS.getPost() in *[P]* implies FS $\rightarrow$ exists(s:Snapshot | *[s $\models$ S]*)
GSTRBAC-TP28	context r:Role inv GSTRBAC-TP28: inv: *[P]* implies sometime *[S]*	context r:Role inv GSTRBAC-TP28: inv: let CS: Snapshot = self.getCurrentSnapshot(), in let FS: Set(Snapshot) = CS.getPost() in *[P]* implies FS $\rightarrow$ exists(s:Snapshot | *[s $\models$ S]*)
GSTRBAC-TP29	context u:User inv GSTRBAC-TP29: let CS: Snapshot = u.getCurrentSnapshot(), in let NS: Snapshot = CS.getNext() in u.zonechanged implies NS.u.getActivatedRoles(currentzone) $\rightarrow$ isEmpty()	context u:User inv GSTRBAC-TP29: u.zonechanged implies next self.getActivatedRoles(currentzone)$\rightarrow$ isEmpty()

Appendix D

The USE Specification of The Steam Boiler Control System

D.1 The USE Specification

```
model SteamBoilerSTM


enum valveState {open, closed}
enum State {on, off}
enum Mode {Normal, Initialization , Degraded, Rescue,
   EmergencyStop}


class SteamBoiler
attributes
  ready: Boolean
  capacity: Real
  minimalNormal: Real
  maximalNormal: Real
  maximumIncrease: Real
  maximumDecrease: Real
  minimalLimit: Real
  maximalLimit: Real
  valveOpen: valveState
  operations
  getCurrentSnapshot(): Snapshot = self.snp
```

```
end

class SteamMeasurmentDevice
attributes
  ready : Boolean
  evaporationRate: Real
  operations
  getCurrentSnapshot(): Snapshot = self.snp
end

class WaterLevelMeasurmentDevice
attributes
  ready : Boolean
  waterLevel: Real
  operations
  getCurrentSnapshot(): Snapshot = self.snp
end

class Pump
attributes
  ready : Boolean
  capacity: Real
  mode: valveState
  operations
  getCurrentSnapshot(): Snapshot = self.snp
end
```

```
class PumpControler
attributes
  ready : Boolean
  circulating : Boolean
  operations
  getCurrentSnapshot(): Snapshot = self.snp
end

class ControlProgram
attributes
    mode: Mode
    ready: Boolean
    failureDetected:Boolean
    wlmdFailure: Boolean
    smdFailure: Boolean
    pumpFailure: Boolean
    pumpControlerFailure: Boolean
 operations
  getCurrentSnapshot(): Snapshot = self.snp
end

class Snapshot
    operations
    getNext(): Snapshot = self.nextTrans.nextSnapshot
    futureClosure(s : Set(Snapshot)) : Set(Snapshot)= if s->
        includesAll(s.getNext()->asSet()) then s else
        futureClosure(s->union(s.getNext()->asSet()))endif
    getPost(): Set(Snapshot) = futureClosure(Set{self.getNext()})
    getPrevious(): Snapshot = self.previousTrans.previousSnapshot
    previousClosure(s : Set(Snapshot)) : Set(Snapshot)= if s->
       includesAll(s.getPrevious()->asSet()) then s else
       previousClosure(s->union(s.getPrevious()->asSet()))endif
    getPre(): Set(Snapshot) = previousClosure(Set{self.
       getPrevious()})
end

abstract class Transition
--operations
  --  nextTransition(): Transition = self.nextSnapshot.nextTrans
   -- closureTrasitions(s : Set(Transition)) : Set(Transition)=
      if s->includesAll(s.nextTransition()->asSet()) then s else
      closureTrasitions(s->union(s.nextTransition()->asSet()))
      endif
    --futureTransitions(): Set(Transition) = closureTrasitions(
       Set{self.nextTransition()})
end

class WaterLevelMeasurmentDevice_getLevel < Transition
  attributes
    wlmdPre:WaterLevelMeasurmentDevice
    wlmdPost:WaterLevelMeasurmentDevice
    ret: Real
end
```

```
class SteamMeasurmentDevice_getSteam < Transition
  attributes
    smdPre:SteamMeasurmentDevice
    wmdPost:SteamMeasurmentDevice
    ret: Real
end

class SteamBoiler_OpenValve < Transition
  attributes
    sbPre:SteamBoiler
    sbPost:SteamBoiler
end

class PumpControler_OpenPump < Transition
  attributes
    PCPre:PumpControler
    PCPost:PumpControler
end

class PumpControler_ClosePump < Transition
  attributes
    PCPre:PumpControler
    PCPost:PumpControler
end

class ControlProgram_StartOperation < Transition
  attributes
```

```
    CPPre: ControlProgram
    CPPost: ControlProgram
end

----------------------associations----------------

association SteamBoilerControlProgram between
  SteamBoiler [1] role sb
  ControlProgram [1] role program
end

association SteamBoilerWLMD between
  SteamBoiler [1] role sb
  WaterLevelMeasurmentDevice [1] role wlmd
end

association SteamBoilerSMD between
  SteamBoiler [1] role sb
  SteamMeasurmentDevice [1] role smd
end

association SteamBoilerPump between
  SteamBoiler [1] role sb
  Pump [1] role pump
end

association ControlProgramPump between
```

```
  ControlProgram [1] role program
  Pump [1] role pump
end

association PumpControlerPump between
  PumpControler [1] role controler
  Pump [1] role pump
end

association ControlProgramPumpControler between
  ControlProgram [1] role program
  PumpControler [1] role PC
end

association ControlProgramWLMD between
  ControlProgram [1] role program
  WaterLevelMeasurmentDevice [1] role wlmd
end

association ControlProgramSMD between
  ControlProgram [1] role program
  SteamMeasurmentDevice [1] role smd
end

association BeforeTrans between
  Snapshot [0..1] role previousSnapshot
  Transition [0..1] role nextTrans
```

```
end

association AfterTrans between
  Snapshot [0..1] role nextSnapshot
  Transition [0..1] role previousTrans
end

composition SnapshotSteamBoilers between
 Snapshot [1] role snp
 --Snapshot [*] role snapshots
 SteamBoiler [1] role boiler
end

composition SnapshotWMD between
 Snapshot [1] role snp
 WaterLevelMeasurmentDevice [1] role WLMD
end

composition SnapshotPump between
 Snapshot [1] role snp
 Pump[1] role pump
end

composition SnapshotPumpControler between
 Snapshot [1] role snp
 PumpControler[1] role PC
end
```

```
composition SnapshotSMD between
 Snapshot [1] role snp
 SteamMeasurmentDevice [1] role SMD
end

composition SnapshotControlProgram between
 Snapshot [1] role snp
 ControlProgram [1] role program
end
----------------constraints-------
constraints

--TP1
context ControlProgram
inv: let CS:Snapshot = self.snp in let NS: Snapshot= CS.getNext()
in self.wlmdFailure implies NS.program.mode= # Rescue

--TP2
context ControlProgram
inv: let CS:Snapshot = self.snp in let NS: Snapshot= CS.getNext()
in (self.smdFailure or self.pumpFailure or self.
   pumpControlerFailure) implies NS.program.mode= # Degraded

--TP3
context SteamBoiler
inv: let CS:Snapshot = self.snp in let NS: Snapshot= CS.getNext()
```

```
in (self.wlmd.waterLevel > self.maximalNormal or self.wlmd.
   waterLevel < self.minimalNormal) implies NS.program.mode= #
   EmergencyStop



--TP4
context SteamBoiler
inv: let CS:Snapshot = self.snp in let FS: Set(Snapshot)= CS.
   getPost()
in self.valveOpen = # open implies FS->exists(s:Snapshot | s.WLMD
   .waterLevel <= maximalNormal)

--TP5
context ControlProgram
inv: let CS:Snapshot = self.snp in let NS: Snapshot= CS.getNext()
in (self.mode=# Initialization and self.wlmdFailure) implies NS.
   program.mode= # EmergencyStop



-- initial class diagram invariants

context WaterLevelMeasurmentDevice
inv: self.waterLevel < self.sb.capacity

context SteamBoiler
inv: self.valveOpen=#open implies self.program.mode=#
   Initialization
```

```
context PumpControler_OpenPump
inv: PCPre.pump.mode = #off
inv: PCPost.pump.mode = #on
--=========End of USE Specification============
```

D.2 The Temporal Properties of The SBCS System

Table D.1: The Temporal properties of the SBCS system, SBCS-TP1 to SBCS-TP13

No.	Description	Pattern - Scope
SBCS-TP1	As soon as the program recognizes a failure of the water measuring device unit it goes into the rescue mode.	Response-Globally
SBCS-TP2	Failure of any physical units except the water measuring device puts the program into degraded mode.	Response-Globally
SBCS-TP3	If the water level is close to reaching the maximalLimit or minimalLimit values (i.e., greater than maximalNormal or less than minimalNormal) the program enters the mode emergency stop.	Response-Globally
SBCS-TP4	When the valve of the steam boiler is open, then eventually the water level will be lower or equal to the maximal normal level.	Response-Globally
SBCS-TP5	When the program is in the initialization mode and a failure of the water level measurement device is detected it puts the program in the emergency stop mode.	Response-Globally
SBCS-TP6	When the system is in initialization mode, it remains in this mode until all physical units are ready or a failure of the water level measurement device has occurred.	Universality-between Q and R
SBCS-TP7	When the pump fails, the system goes into the degraded mode.	Response-Globally
SBCS-TP8	When the steam measuring device fails, the system goes into the degraded mode.	Response-Globally
SBCS-TP9	When the pump controller fails, the system goes into the degraded mode.	Response-Globally
SBCS-TP10	When a failure is detected, the system goes into the degraded mode.	Response-Globally

Table D.2: TOCL and OCL specification of the SBCS temporal properties described in Table D.1, taken from our previous work Al-Lail et al. [95]

No.	TOCL Specification on Class Model	OCL Specification on the Snapshot Transition Model
SBCS-TP1	**context** *ControlProgram* **inv:** *self.wlmdFailure implies* *next self.mode=# Rescue*	**context** *ControlProgram* **inv:** *inv: let CS: Snapshot= self.snp* *in NS: Snapshot= CS.getNext()* **in** *self.wlmdFailure implies NS.program.mode= # Rescue*
SBCS-TP2	**context** *ControlProgram* **inv:** *(smdFailure or pumpFailure* *or pumpcontrollerFailure) implies* *next self.mode=# Degraded*	**context** *ControlProgram* **inv:** *let CS: Snapshot= self.getCurrentSnapshot()* in let NS: Snapshot = CS.$getNext()$ **in** *(self.pumpcontrollerFailure or self.pumpFailure or* *self.smdFailure) implies NS.program.mode =# Degraded*
SBCS-TP3	**context** *SteamBoiler* **inv:** *(self.wlmd.waterLevel >* *self.maximalNormal or self.wlmd.waterLevel* *< self.minimalNormal) implies next* *self.program.mode = # EmergencyStop*	**context** *SteamBoiler* **inv:** *let CS: Snapshot = self.snp* in let NS: Snapshot = CS.$getNext()$ **in** *(self.wlmd.waterLevel > self.maximalNormal or* *self.wlmd.waterLevel < self.minimalNormal) implies* *NS.program.mode = # EmergencyStop*
SBCS-TP4	**context** *SteamBoiler* **inv:** *self.valveOpen = # open implies* *sometime* *(self.wlmd.waterLevel < = maximalNormal)*	**context** *SteamBoiler* **inv:** *let CS: Snapshot = self.snp* in let FS: Set(Snapshot) = CS.$getPost()$ **in** *self.valveOpen = # open implies FS → exists* *(s:Snapshot | s.WLMD.waterLevel < = maximalNormal)*
SBCS-TP5	**context** *ControlProgram* **inv:** *(self.mode = # Initialization* *and self.wlmdFailure) implies* *next self.mode =# EmergencyStop*	**context** *ControlProgram* **inv:** *let CS: Snapshot = self.snp* *in let NS: Set(Snapshot) = CS.*$getNext()$ **in** *(self.mode = # Initialization and self.wlmdFailure)* *implies NS.program.mode = # EmergencyStop*
SBCS-TP6	**context** *ControlProgram* **inv:** *self.mode = # Initialization implies* *always self.mode = # Initialization* *until (PhysicalUnit.allInstances→* *forAll(u: PhysicalUnit | u.ready))*	**context** *ControlProgram* **inv:** *let CS: Snapshot = self.snp* *in let* FS_1*: Snapshot = CS.getPost() → select(s:Snapshot |* *s.boiler.ready and s.SMD.ready and s.pump.ready* *and s.PC.ready and s.WLMD.ready)→ first()* *in let* $PreFS_1$*=Set(Snapshot) =* FS_1*.getPre()* *in let BTS: Set(Snapshot)=*$PreFS_1$ *→ excluding(CS.getPre())* **in** *self.mode = # Initialization implies BTS → forAll* *(s1:Snapshot | s1.program.mode= # Initialization)*

Table D.3: TOCL and OCL specification of the SBCS temporal properties described in Table D.1, taken from our previous work Al-Lail et al. [95]

No.	TOCL Specification on Class Model	OCL Specification on the Snapshot Transition Model
SBCS-TP7	**context** *ControlProgram* **inv:** *self.pumpFailure implies* *next self.mode=# Degraded*	**context** *ControlProgram* **inv:** *inv: let CS: Snapshot= self.snp* *in NS: Snapshot= CS.getNext()* **in** *self.pumpFailure implies NS.program.mode= # Degraded*
SBCS-TP8	**context** *ControlProgram* **inv:** *self.smdFailure implies* *next self.mode=# Degraded*	**context** *ControlProgram* **inv:** *inv: let CS: Snapshot= self.snp* *in NS: Snapshot= CS.getNext()* **in** *self.smdFailure implies NS.program.mode= # Degraded*
SBCS-TP9	**context** *ControlProgram* **inv:** *self.pumpControllerFailure implies* *next self.mode=# Degraded*	**context** *ControlProgram* **inv:** *inv: let CS: Snapshot= self.snp* *in NS: Snapshot= CS.getNext()* **in** *self.pumpControllerFailure implies* *NS.program.mode= # Degraded*
SBCS-TP10	**context** *ControlProgram* **inv:** *self.failureDetected implies* *next self.mode=# Degraded*	**context** *ControlProgram* **inv:** *inv: let CS: Snapshot= self.snp* *in NS: Snapshot= CS.getNext()* **in** *self.failureDetected implies* *NS.program.mode= # Degraded*

D.3 The Temporal Properties Fragments

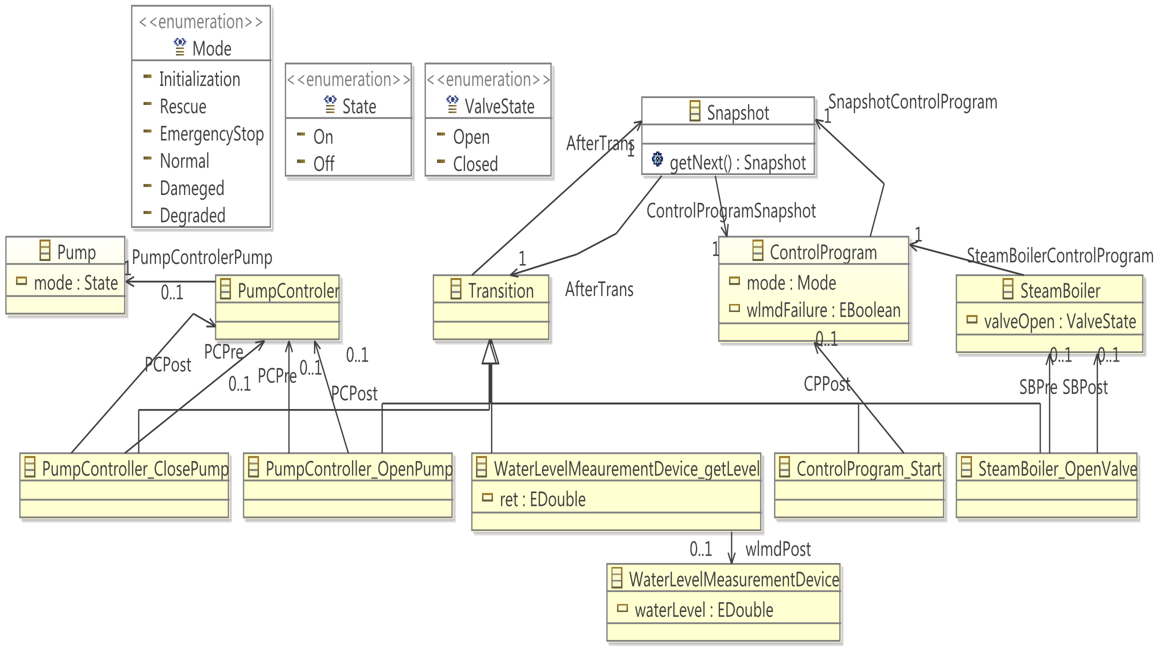


Figure D.1: The STM slice with respect to SBCS-TP1, STM-TP1-Slice

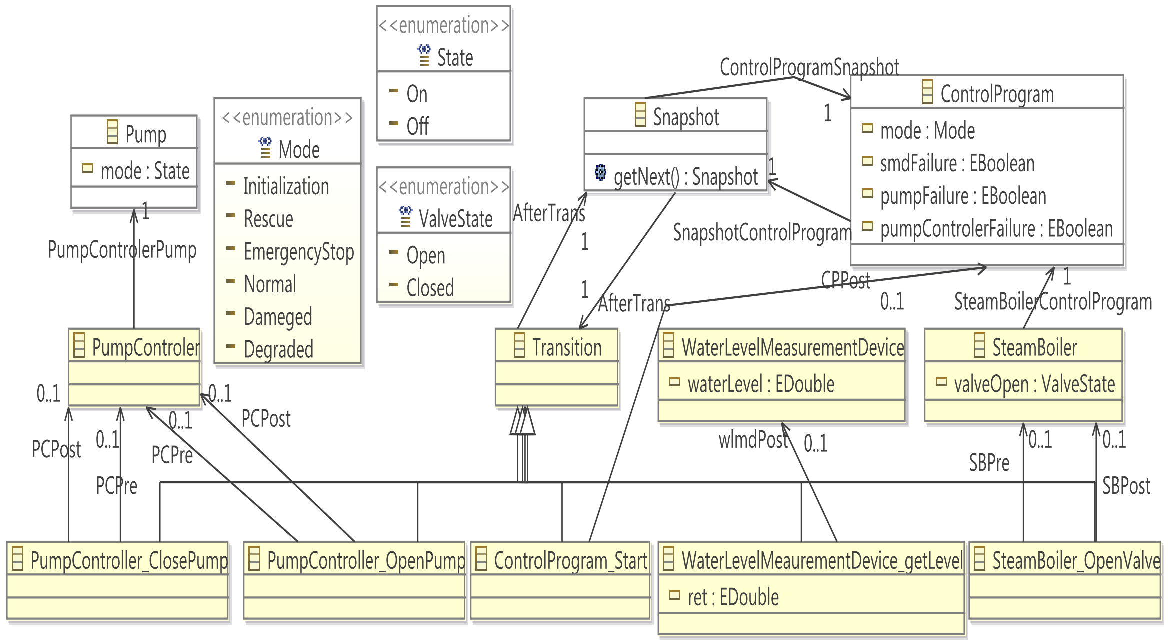


Figure D.2: The STM slice with respect to SBCS-TP2, STM-TP2-Slice

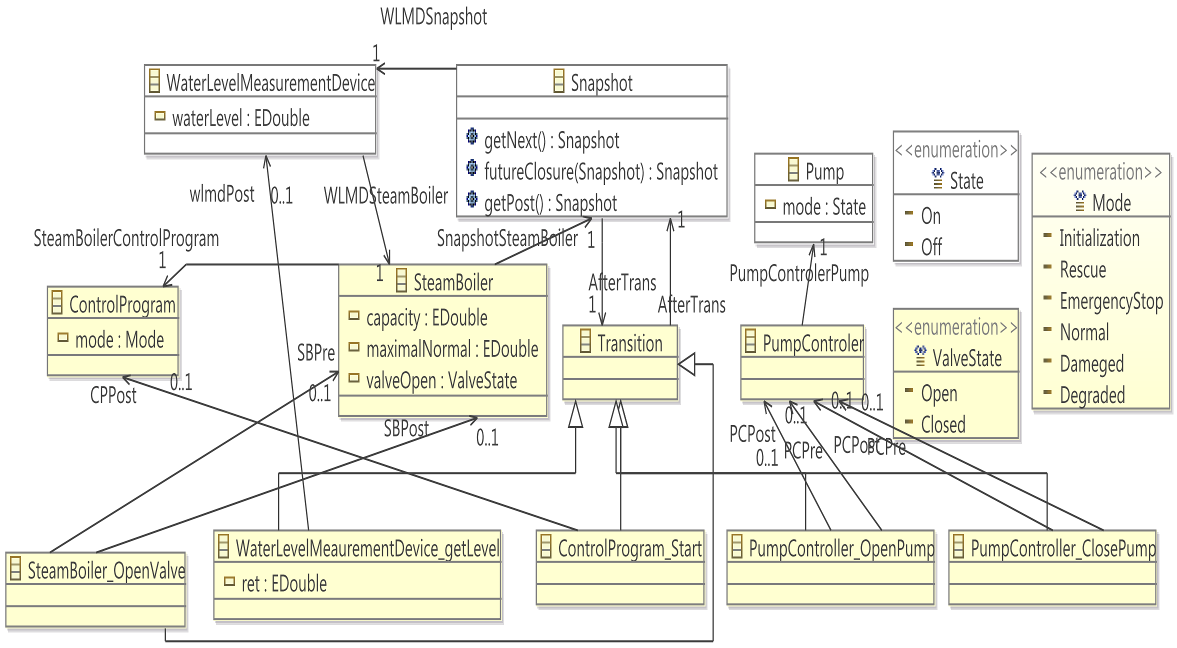


Figure D.3: The STM slice with respect to SBCS-TP3, STM-TP3-Slice

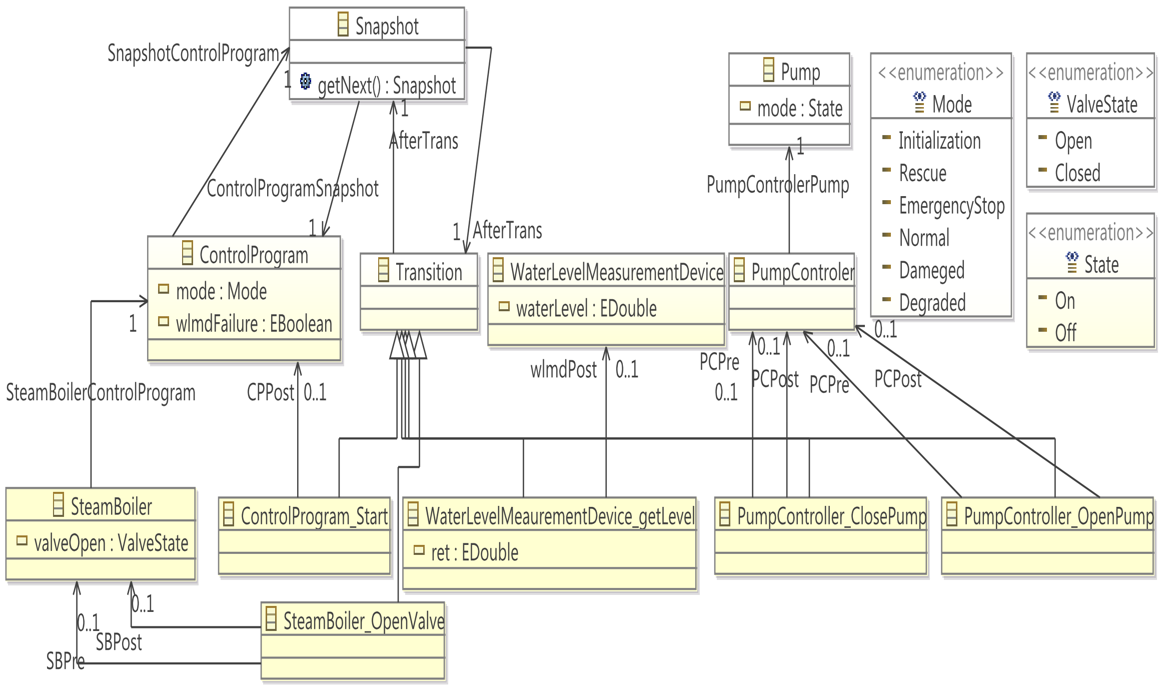


Figure D.4: The STM slice with respect to SBCS-TP4, STM-TP4-Slice