Åbo Akademi University

Faezeh Siavashi

Model-based Verification and Testing of Web services

Functionality, Robustness and Vulnerability Analysis

Faezeh Siavashi

Born 1983

BSc. in Software Engineering, Azad Tehran University 2006
MSc. In Embedded Systems, Åbo Akademi University 2012
PhD. dissertation defence in Åbo Akademi 2020

Model-Based Verification and Testing of Web Services

Functionality, Robustness and Vulnerability Analysis

Faezeh Siavashi

To be presented, with the permission of the Faculty of Science and Engineering of Åbo Akademi University, for public criticism in Auditorium XX, the Agora building, on April 2nd, 2020, at 13:00.

Åbo Akademi University
Faculty of Science and Engineering
Agora, Vattenborgsvägen 3,
20500 , Åbo, Finland

2020

Supervisors

Adjunct Professor Dragos Truscan
Faculty of Science and Engineering
Åbo Akademi University
Agora, Domkyrkotorget 3, 20500, Åbo
Finland

Professor Jüri Vain
Department of Software Science
Tallinn University of Technology
Ehitajate tee 5, 19086, Tallinn
Estonia

Professor Ivan Porres
Faculty of Science and Engineering
Åbo Akademi University
Agora, Domkyrkotorget 3, 20500, Åbo
Finland

Reviewers

Associate Professor Cristina Seceleanu
Mälardalen University
School of Innovation, Design and Engineering, Västerås
Sweden

Professor Fevzi Belli
Department of Electrical Engineering and Information Technology
Universität Paderborn
Melkweg 37 c, D-33106 Paderborn
Germany

Opponent

Associate Professor Cristina Seceleanu
Mälardalen University
School of Innovation, Design and Engineering, Västerås
Sweden

ISBN 978-952-12-3935-9 (printed)
ISBN 978-952-12-3936-6 (digital)
Painosalama Oy, Åbo, Finland 2020.

تقدیم به پدر و مادر عزیزم

بسان رود که در نشیب درّه
سر به سنگ می زند رونده باش،
امید هیچ معجزی ز مُرده نیست،
زنده باش

ه. ا. سایه

Abstract

Web services are software systems that are designed to support machine-to-machine interactions over a network. These services enable people to access a wide variety of resources through their personal computers or mobile devices. Nowadays, most of the traditional business activities and manual services are being enhanced by web services.

When using web services, users expect that not only they are continually available, but also they work correctly and provide secure access to data. However, ensuring the quality of web service implementations is not simple for several reasons. First, they are accessible using public communication protocols from anywhere in the world by a large number of users. Therefore, web services should be robust enough to be able to accept correct input and reject incorrect and malicious ones. Secondly, the reputation of web services greatly depends on how they perform as expected while securing users' data. Finally, the implementation of web services should be developed in such a way that it can ensure access control of users. Implementing new features or modifying a developed feature requires additional attention to ensure that access control is not altered.

Software testing is one of the techniques used for quality assurance of software systems. In software testing in which a set of inputs is provided to the system under test and the outputs produced are compared against the expected outputs. Testing is typically a largely informal process, which in many cases is left at the end of the software development process and reduced to fit the project deadlines. Thus, there is a need to deploy novel testing methods that will make testing of web services both efficient and effective.

In this thesis, we define a model-based testing approach to evaluate the behavior of web services and their compositions. The goal of using models is to introduce a verifiable specification of the web service that is used later on to generate tests that are executed against the implementation of the web service. In our approach, we use model checking to make sure that the requirements of the web service are satisfied by the model-based specification. Then, we use model-based testing to check that the implementation of the web service corresponds to verified specifications, and consequently to

its requirements.

As a first step, we study how the interaction of the web service with their environment can be modeled and used for test generation. To this extent, we conduct a systematic literature review on the use of environment models in model-based testing. Then, we provide a first approach in which the interactions of web services with their environment are modeled with UML sequence diagrams, while the behavior of the web services is specified with UML state machines. Both models are transformed into UPPAAL Timed Automata, to verify that the behavior of the services allows the specified interactions. Besides, we verify that different requirements of the web service are satisfied. The resulting verified model is used for online test generation against the implementation of the web service.

As another contribution of this thesis, we define an approach to assess the robustness and security vulnerabilities of web services in the presence of unexpected or invalid conditions and inputs. To that extent, we extend our previous MBT approach in the context of mutation testing. In model-based mutation testing, the original test model is altered via mutation operators to provide slightly incorrect behavior. The mutant models are used to generate tests that will provide invalid test inputs to evaluate how robust the service implementation is to such inputs and if any security vulnerabilities are exposed.

Furthermore, we extend the above approach with two contributions. We first define a set of mutation operators for UPPAAL timed automata and we evaluate them empirically in the context of web services. Then, we propose a novel mutation-selection technique that eliminates the mutant models that are not useful for testing and we, consequently, reduce the test execution time. The results show that these techniques increase the efficiency and effectiveness of our approach.

In this thesis, several tools that facilitate testing activities support the testing approach. In some cases, we used existing tools like the UPPAAL model checker and the UPPAAL TRON test generator. In other cases, we have complemented the existing toolset with new tools. Notably, for the mutation testing approach, we implemented a tool called μUTA, which automates the mutant generation, mutant-selection, and mutant execution processes.

The approaches defined in this thesis have been applied in two case studies. The results show that our testing methodology can create test cases that explore the behavior of the systems extensively and reveal new faults that remain undetected by traditional model-based testing methods.

Sammandrag

Webbtjänster är mjukvarusystem som är designade för att stöda växelverkan mellan maskiner över ett nätverk. Dessa tjänster ger tillgång till ett brett utbud resurser från sina persondatorer och mobiltelefoner. I dagens läge kan man se en förbättring av traditionell affärsverksamhet och manuella tjänster orsakad av webbtjänster.

Användare av webbtjänster förväntar sig inte bara att de är kontinuerligt tillgängliga, men att de också uppfyller alla krav på datasäkerhet. Dock är så finns det många orsaker som gör att det inte är lätt att säkerställa kvaliteten av webbtjänster implementationer. För det första, är de tillgängliga till ett stort antal användare globalt, genom allmänna kommunikationsprotokoll. Därför borde webbtjänster vara tillräckligt robusta för att kunna acceptera en korrekt inmatning men också rata en inkorrekt eller skadlig inmatning. För det andra, hänger webbtjänsters anseende på hur de förväntas prestera på samma gång som de måste bevara integriteten för alla användares data. Den tredje orsaken är att webbtjänst implementationer borde utvecklas så att de kan garantera att den implementerade åtkomstkontrollen är verksam. När man utvecklar ny funktionalitet och förändrar existerade funktionalitet krävs att man uppmärksammar att det inte sker oavsiktliga förändringar i åtkomstkontrollen.

Mjukvarutestning är en av metoderna som används för säkerställning av kvaliteten för mjukvarusystem. Mjukvarutestning går ut på att man gör en inmatning till systemet under testning och jämför utmatningen från systemet jämförs med en förväntad utmatning. Mjukvarutestning är vanligen en informell process som ofta lämnas kvar till slutet av utvecklingsprocessen, så att den ofta minskas i omfång för att passa projektets deadline. Därför finns det ett behov av att använda sig att nya mjukvarutestningsmetoder så att testninen blir både effektiv och verksam.

I denna avhandling definierar vi ett tillvägagångssätt för modellbaserad testning för att utvärdera beteendet av webbtjänster och kombinationer av webbtjänster. Målet med att skapa modeller är att man ska kunna göra en verifierbar specifikation av webbtjänsten, som man senare ska kunna använda för att generera mjukvarutest som jämförs mot implementeringen av webbtjänsten. I vårt tillvägagångssätt använder vi en modelchecking-

metodik för säkerställa att webbtjänstens krav möter den modellbaserade specifikationen. Efter detta använder vi modellbaserad mjukvarutestning för att säkerställa att implementeringen av webbtjänsten motsvarar den verifierade specifikationen och som konsekvens också webbtjänstens krav.

Först studerar vi hur samverkan mellan webbtjänster och dess omgivning kan modelleras och hur den kan användas för generering av mjukvarutest. För detta ändamål gjorde vi en systematiskt litteraturstudie om hur andra har använt modeller av omgivningen i modellbaserad mjukvarutestning. Efter detta presenterar vi en ett tillvägagångssätt där webbtjänsten omgivning modelleras med ett UML sekvens diagram, medan webbtjänsten själv beskrivs med UML tillståndsmaskiner. Båda modeller översätts till UPPAAL Timed Automata för att man ska kunna verifiera att tjänsternas beteenden tillåter specificerad samverkan. Därtill verifierar vi att webbtjänstens olika krav uppfylls. Den verifierade resulterande modellen används för online generering av mjukvarutest mot en implementering av webbtjänsten.

Som ett andra bidrag i denna avhandling, definierar vi ett tillvägagångssätt för att utvärdera webbtjänsters robusthet och säkerhets sårbarheter i närvaro av oförväntade och ogiltiga tillstånd och inmatningar. För detta utökar vi vårt modellbaserade tillvägagångssätt i avseende på mutationstestning. I modellbaserad mutationstestning modifieras det ursprungliga testet med en muteringsoperator för att skapa ett aningen inkorrekt beteende. Mutantmodeller används för att generera mjukvarutest som ger inkorrekta inmatningar, så att robustheten mot sådana inmatningar kan utvärderas, på samma gång som nya säkerhetsbrister kan uppdagas.

Slutligen breddar vi det ovanstående tillvägagångssättet med två bidrag. Först definierar vi en samling muteringsoperatorer for UPPAAL Timed Automata som vi utvärderar empiriskt inom sammanhanget för webbtjänster. Sedan presenterar vi en ny metod för att välja ut muteringar som kan utesluta mutationer som inte är användbara i testningssyfte. Genom att använda denna metod kan man minska mjukvarutestens tidsanspråk. Våra resultat visar att dessa metoder förbättrar vårt tillvägagångssätts effektivitet och gör det mera verksamt.

Under arbetet för denna avhandling har flera verktyg använts som stöder den presenterade mjukvarutestningsmetodiken. I några studier använde vi verktyg som UPPAAL modelchecker och UPPAAL TRON test generator. I andra fall så har vi kompletterat existerande verktyg. Ett exempel på ett verktyg som vi implementerade är μUTA som automatiserar mutant generering, urval av mutanter och en mutantexekveringsprocess.

Tillvägagångssätten som definieras i denna avhandling har tillämpats i två fallstudier. Våra resultat visar att vår mjukvarutestningsmetodik kan skapa testfall kan göra omfattande utforskning av ett systems beteende, så att nya fel som blir annars skulle förbli oupptäckta av traditionella modellbaserade metoder kan bli avslöjade.

Acknowledgements

Completing this thesis, a product of several years of work, I feel indebted to many people who have supported me during this journey.

I would like to sincerely thank my supervisor, Adjunct Professor Dragos Truscan, for his invaluable guidance, encouragement and generous help through these years from initiation to completion. I would like to extend my gratitude to Professor Jüri Vain, for his incredible guidance and support. I would also like to thank Professor Ivan Porres for providing the opportunity to pursue my doctoral studies in Software Engineering Laboratory.

My sincere thanks also go to Professor Fevzi Belli and Adjunct Professor Cristina Seceleanu for providing valuable comments which helped in preparing the thesis into its current form. I am also thankful to Adjunct Professor Cristina Seceleanu for accepting to act as the opponent at my doctoral defense.

This thesis is a product of collaborations of many ideas and efforts from several people. I would like to thank the past and present members of the Software Engineering Laboratory, my teachers, coauthors, and colleagues for the great discussions, collaborations and feedback. Special thanks to administrative and technical staff, especially Christel Engbolm, Minna Carla, Pia-Maria Kallio, and Karl Rönnholm who never wavered their support. I am also grateful to Wictor Lund for translating the abstract of this thesis to Swedish.

I am deeply grateful for generous grants from Åbo Akademi University for funding my doctoral studies, Harry Elvings legat grant for my research visit, and Ulla Tuominen Foundation research grant. I would also like to express my gratitude to the generous scholarship that Anita Borg Institute granted me to be part of the Grace Hopper Celebration on Women in Computing. All of these support have encouraged me to continue my work, and to aspire to inspire.

It would have been a farfetched dream to reach this point if it was not for the permanent love and encouragement of my family. I owe a debt of gratitude to my dear parents, Azam and Ali, for being my first and foremost educators. Thank you for believing in me before I believed in myself. I am profoundly grateful to my brother, Amir, and my sisters, Firouzeh and

Ferdows, for their continual motivation and love. Special thanks to my dear sister Ferdows for designing the cover photo of this thesis. My deepest gratitude belongs to my best friend, my beloved husband, Dr. Tewodros Deneke for enlightening my life with his love, wisdom, and kindness, and for inspiring me every day.

Faezeh Siavashi
Helsinki, March 2020

List of Original Publications

Publication I I. Rauf, F. Siavashi, D. Truscan, and I. Porres. An integrated approach for designing and validating REST web service compositions. In *WEBIST 2014 - Proceedings of the 10th International Conference on Web Information Systems and Technologies*, volume 1, pages 104–115, Barcelona, Spain, 2014. SCITEPRESS Digital Library

Publication II F. Siavashi and D. Truscan. Environment Modeling in Model-based Testing: Concepts, Prospects and Research Challenges: A Systematic Literature Review. In *EASE 2015 - Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–6, Nanjing, China, 2015. ACM

Publication III I. Rauf, F. Siavashi, D. Truscan, and I. Porres. Scenario-based design and validation of REST web service compositions. In *Web Information Systems and Technologies – Revised Selected Papers*, pages 145–160. Springer International Publishing, 2015

Publication IV F. Siavashi, D. Truscan, I. Rauf, and J. Vain. On Mutating UPPAAL Timed Automata to Assess Robustness of Web Services. In *ICSOFT 2016 – Proceedings of the 11th International Joint Conference on Software Technologies*, volume 1, pages 15–26, Lisbon, Portugal, 2016. SCITEPRESS - Science and Technology Publications

Publication V F. Siavashi, J. Iqbal, D. Truscan, and J. Vain. Testing web services with model-based mutation. In *Software Technologies – Revised Selected Papers*, pages 45–67. Springer International Publishing, 2017

Publication VI F. Siavashi, D. Truscan, and J. Vain. Vulnerability assessment of web services with model-based mutation testing. In *QRS 2018 – IEEE 18th International Conference on Software Quality, Reliability, and Security*, pages 301–312, Lisbon, Portugal, 2018. IEEE Computer Society Conference Publishing Services

Contents

Part I

Research Summary

Chapter 1

Introduction

Software is an integral part of many systems and devices. It defines and controls the behavior of various types of systems, from large-scale infrastructures, such as financial networks, the Web, traffic control systems, to personal devices such as mobile phones, web applications, and home appliances. Software technologies have become an essential part of people's life enabling instant access to remote resources and automating most of their daily activities. *Web services* are such software systems that are designed to enable machine-to-machine interactions over a network. They are accessed via a set of well-defined interfaces supplying information to other servers or applications, which in turn provide the information to end-users in a human readable format [80, 109].

Web services concentrate large amounts of sensitive data related to their users and are expected to be responsible for their integrity, robustness, and security in the presence of stressful conditions [30, 110]. Robustness and security of web services are among important subjects in software testing. Web services are expected to not only be fully tested based on their specifications but also they should not have any unexpected behavior in unknown conditions. To evaluate whether a web service is correctly implemented, a set of tests can be executed against the system and the test results can be compared with the expected results. In contrast, in order to examine the robustness of a web service, its implementation should be executed against negative or malicious test inputs intended to break its functionality and reveal internal defects. *Robustness testing* in service-based systems consists of techniques for the evaluation of the systems in presence of erroneous input conditions to explore potential security vulnerabilities [30, 58].

Authentication and *authorization* are among the main security concerns in web services and data that they maintain. While authentication procedure verifies user identity, authorization defines and maintains access controls of content within the system. In web services such as social networks,

in which users define and change their relationships, groups or privacy of their content, specifying the authorization and authentication are challenging. Flaws in the implementation or specification of user credentials in web services are still among top security concerns reported by Open Web Application Security Project (OWASP) [105]. The reason behind this problem is that the development of service-based systems has been evolving rapidly, resulting in poor software quality [77].

Software quality is mainly achieved through applying *verification and validation* (V&V) procedures throughout software development life-cycle [2]. The goal of verification and validation is to produce precise and correct systems. V&V have been described from different perspectives which are provided in the literature [47]. In the context of software engineering, verification and validation are defined as follows:

> *Verification* determines whether the products of a given development phase satisfy the conditions imposed at the start of that phase [47].
>
> *Validation* evaluates a system or component during or at the end of the development process to determine whether it satisfies specified requirements [47].

In other words, verification provides evidence that the system conforms to requirements (e.g., for correctness, completeness, consistency) for all system life-cycle activities. In addition to manual reviews and walkthroughs, verification can also include formal and mathematical activities such as model-checking and theorem proving to determine the correctness of system behavior. In contrast, validation includes a set of activities such as executing the system to gain confidence that the system can accomplish its intended use, goals, and objectives. Software testing is the method of V&V, examining software product compliance with specifications in each development stage, as well as evaluating them against user requirements.

In a typical testing process, the requirements of a system under test (SUT) are analyzed to define the test criteria that are used to design test cases. After test design, the test cases are implemented into executable test scripts, which include a *test oracle*. The executable tests generate a range of inputs and check if the expected output is received, based on the test oracle.

While the testing process for most types of software systems is relatively the same, many new testing techniques have been progressively developed to deal with their steadily growing complexity [19, 24, 74, 85]. In the scope of testing web services, studies show that simulating stressful conditions and applying advanced techniques such as fault injection [112], cross-site scripting (XSS) [51], threat modeling [78], and mutating input data [62,

63, 65] can significantly improve the web service quality and speed up the expensive process of testing.

As software systems are getting more advanced, modeling has been widely accepted for design and testing [29, 103, 108]. Models reduce the complexity of specifications by removing unnecessary details and focusing on more significant parts of the systems. Formal modeling languages are used to specify complex and critical systems and verify their behavior rigorously. Models can represent either the behavior of a SUT or its environment or both [45, 46, 49]. In a system that the environment (i.e., nature, human, or machine) can influence its behavior, using the model of the environment facilitates evaluating the behavior of the system.

Model-based Testing (MBT) is a testing technique that relies on models of a SUT and/or its environment to derive test cases [108]. MBT is primarily a black-box testing technique that generates tests from abstract behavioral models with the goal to validate that the actual behavior of a system complies with its specification. In the domain of safety-critical systems, MBT is becoming an increasingly important technology recommended by safety standards such as IEC-61508-3 [32]. Such a testing approach that verifies whether a product performs according to its specified requirements are referred to as *conformance testing*.

To achieve a higher quality of software systems in terms of robustness and security, they need to be tested further under unexpected conditions. To assess the robustness of a system with MBT, one can model negative test scenarios that intend to break the functionality of the software. However, manually defining the possible negative scenarios is generally error-prone and unfeasible in complex systems. One way to create the invalid tests is to mutate the test cases by combining MBT and mutation testing. The combination is known as *Model-based Mutation Testing* (MBMT) [8] or *specification mutation testing* [12]. In MBMT, the original test model is altered systematically by *mutation operators* creating multiple versions of a model (known as *mutant* models). The mutants can be used for the automatic generation of mutated test inputs that are executed against the SUT.

With the use of web services in businesses and critical applications, there is an increasing need for design approaches that support complex scenarios and timed behavior. Moreover, security concerns such as authentication and authorization require specific attention in testing. Identifying such properties and verifying them with formal models have been done in small number of studies and requires more research. Even though the available MBT testing tools and techniques show valuable results, there are no broadly-adopted testing method for web services [106]. In a study by Utting et al., [108] on MBT approaches, it has been identified that using MBT for non-functional requirements such as security is still an open issue.

Recent surveys by Papadakis et al., [83] and Jia and Harman [50] pro-

vide a comprehensive overview of the studies in the mutation testing field. In both surveys, combining mutation testing with other testing methods has been concluded to be promising. However, the majority of the generated mutants are usually insignificant (i.e., they are equivalent, redundant or trivial mutants). Due to the complexity of identifying such mutants, in many approaches, all mutants are used for test execution, which decreases the speed of the testing and leads to low efficiency in mutation testing. Moreover, due to dynamic changes in web service features, their security should be evaluated and measured constantly. Efficiency of evaluating the robustness and vulnerabilities of web services is, therefore, essential for providing secure systems as it reflects on their quality of service (QoS) and overall cost and is the main topic of this thesis [110, 57].

1.1 Research Questions

The overall objective of this thesis is to investigate the use of model-based testing techniques based on Uppaal timed automata and to enable a model-based evaluation/verification of robustness and vulnerability of web services, by employing mutation testing. To achieve these objectives, we define the following research questions:

(RQ1:) How to design and verify the model of web services with Uppaal Timed Automata? We explore how to build models from web service specifications. A correctness property can then be stated with a formal specification language, which can be used to verify the model automatically via model-checking. **(RQ2:) What is the role of environment models in model-based testing?** Namely, we investigate the state-of-the-art in utilizing environment models to enhance model-based testing. Additionally, we review application domains and modeling languages that use environment models. Answering this question will also help to understand the main features of environment models used for testing web services and how can they further improve the quality of tests. Which leads to our the next question: **(RQ3:) How to use the model of web services and their environment for functional testing?** We plan to extract tests from models of web services and their environment models and examine the functional behavior of the web services against their implementations. Beside testing functionality of web services, another important issue is to test the robustness of web services under unexpected conditions. Thus, the next question is: **(RQ4:) How to mutate the models for robustness and vulnerability testing?** We aim to explore a set of mutation operators to create invalid test inputs that intend to examine the robustness of the system and expose its vulnerabilities. Generating mutants usually causes testing overhead and reduces the efficiency of the testing. Therefore, the

final question to answer is **(RQ5:) How to select a subset of mutants to increase the efficiency of testing?** We investigate a mutant reduction strategy that eliminates insignificant mutants, such as equivalent, or redundant mutants, to reduce the number of tests.

The research contributions of this thesis and the original publications address these research questions.

1.2 Research Contributions

In this thesis, we use the formal specification for evaluating the robustness and security of web services with Timed Automata (TA). Figure 1.1 illustrates an overview of the approach which is a combination of multiple phases. The upper part of the diagram shows the five main steps of our testing approach, while the lower part of the diagram presents the corresponding activities in each step.

The blue part of the diagram follows the principle of the MBT. From the requirements, a *System Model* is created and verified, leading to a *Verified Model*. For generating executable test cases, the abstract (model-level) tests that are derived from the verified model are translated during the Test Generation step. The test cases, then, will be executed against the implementation of the SUT and the test results will be analyzed.

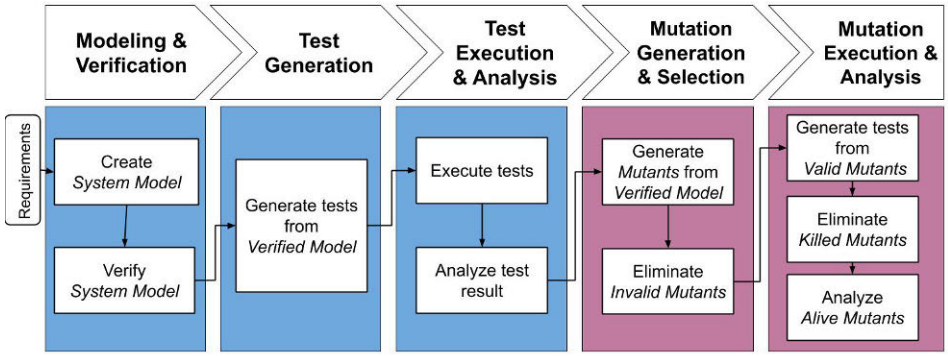


Figure 1.1: Overview of our testing approach utilizing MBT (blue) and MBMT (purple)

The purple part represents the MBMT steps, which start only after the MBT steps are completed. The precondition for MBMT is that all generated tests in MBT have passed (i.e., the model conforms to the implementation). The first step is to generate *Mutants*, i.e. variations of the *Verified Model* with slight changes. These mutants will be used individually for test generation. However, to select only the relevant mutants, we eliminate *Invalid Mutants* and generate tests from the remaining *Valid Mutants*. The tests are

executed against the implementation and if an error is detected during test execution, the model used is considered as *Killed Mutant*, otherwise, as a *Alive Mutant*. The remaining *Alive Mutants* are further analyzed to reason whether a defect is present within the implementation or in the specification.

This thesis advances the state of the art by proposing a set of tool-supported approaches for functional and non-functional model-based testing of web services in the semantic framework of timed automata. The approach allows simulation and verification of service specifications, including time properties, and test generation from such specifications. In addition, it proposes a novel method for model-based mutation testing using UPPAAL timed automata that can be extended and applied to other application domains. The efficiency of the approach is improved by defining a novel mutation selection approach which reduces the number of mutants that have to be executed.

Our testing approach relies on the following contributions, that are presented in the List of Original Publications on page *V*.

Designing and Verifying Web Services with Formal Models

The first contribution addresses RQ1 and includes an approach to model web services and their compositions with UPPAAL timed automata, allowing to simulate and verify the behavior of the web services. Models can either be built directly from the web service specifications, or can be transformed frm other modeling languages, such as Unified Modeling Language (UML). As part of this contribution, we present an approach to convert the specifications represented as UML models into UPPAAL Timed Automata (UTA). The UML models that we use for transformation consist of State Machines and Sequence Diagrams. The service requirements are mapped to the UTA model during the transformation. We define a set of correctness properties that are included in the specifications and UPPAAL model-checker decides whether the formal specifications satisfies the functional requirements.

This contribution is presented in Publication I and Publication III. Publication I demonstrates model transformation, verification, model-based testing of a web service composition that consists of three web services that are orchestrated by a central web service. The central web service (i.e, the composition service) can invoke other services while exhibits timed behavior. The web service composition is built upon RESTful architecture [89] and its specification in UML State Machine format is transformed into the UTA model and then is used for online model-based testing.

Publication III extends the transformation and test generation for web service composition. A set of transformation rules are defined that maps user scenarios from UML Sequence Diagrams into UTA models. The resultant UTA model consists of an environment model (the user behavior) interacting

with the model of web services. The environment model is used for verifying the interactions between SUT and its environment and for generating tests and generating tests.

Exploiting Environment Models for Model-Based Testing

Due to interactions of web services with various systems, or applications, understanding the role of environment models is essential for the assessment of web services. The second contribution is to systematically collect, review, and classify the studies on employing environment modeling in testing. We define a set of questions that identify the main characteristics of environment models, the types of systems or application domains that they are applied to, and how they can facilitate test generation. The literature review is performed from publications retrieved from different academic publication databases such as ACM, Science Direct, and Springer.

This contribution addresses RQ2. The complete process of the literature review is presented in Publication II. The results show that environment models are especially useful in testing systems with high complexity and non-deterministic behavior and can improve automatic test generation. Besides, the environment models can be used for defining invalid test scenarios that simulate stressful conditions to check some of the non-functional properties such as robustness and safety of a system.

Chronologically, the literature review performed after the work of Publication I, as a way to improve the test generation process. The information that is attained from the results is used to test the behavior of a web service with environment models, as presented in Publication III.

Testing Functionality of Web services with Uppaal Models

As third contribution of this thesis, we evaluate the functionality of web service compositions with model-based testing (MBT) using the Uppaal Tron testing tool. From the model, abstract test cases are derived and with a test adapter are translated to executable test cases, which are then automatically executed against the implementation of the web service compositions. The presented testing approach is an online testing, in which each test case is generated and executed based on the result of the previous test inputs. This testing technique provides an increased probability of test coverage and an ease of test case maintenance.

This contribution addresses RQ3 and is presented in Publication I and Publication III. In Publication I, the model of web service composition is used for the test generation, whereas in Publication III, the environment model (user behavior) is used for test generation.

Evaluating the Robustness and Security Vulnerabilities of Web Service Implementations Using mutations of Uppaal Timed Automata

The fourth contribution is to mutate test models to automatically generate invalid test inputs for evaluating the robustness of web services and finding possible vulnerabilities. To this extent, we implement and extend a set of mutation operators that systematically change the model and create several unique mutant models, each carrying a mutation. By applying the MBT technique on the mutant models, invalid test inputs are generated and executed against the SUT and possible hidden faults in the behavior of the SUT can be revealed.

This contribution addresses RQ3 and is evaluated in two case studies within three publications. Publication IV presents the details of the MBMT approach and evaluates it with the web service composition which was tested with the MBT approach in Publication I and II. The result of applying MBMT shows that the system contained some hidden faults that were not exposed by the MBT approach. The second case study consists of user interactions via social web services. Publication V describes a model of single-user activity within a blog web service and tests its implementation. Publication VI presents multi-user activities results of the experiment and shows that mutating specifications helps in detecting faults that could not be revealed in MBT conformance testing previously.

Defining Mutant Selection Criteria for Model Mutations to Improve MBMT Approach

One of the main challenges in generating mutations is to distinguish significant mutants from a large number of trivial or unsuitable mutants. As the fifth contribution, we tackle this problem by defining mutation-selection criteria that eliminate equivalent, or unreachable (i.e., the mutated part of the model is not reachable and thus, the mutated behavior can never be enabled) mutants. This contribution includes modifying the process of selecting mutants with model-checking. A mutation-selection technique for timed automata is defined to systematically verify each generated mutant and increase the efficiency of the testing approach. This contribution answers RQ3 and is presented in Publication VI.

Developing a Supporting Tool for MBMT

Our last contribution is related to implementing a tool for automating mutation generation and verification. As a part of the improvement in the MBMT approach, we develop a tool named μUTA that automates mutation generation, mutation verification, and mutation-selection processes. It consists of

several mutation operators, each systematically alters elements of Uppaal models such as deleting edges, or changing variables. From each mutation operator, various unique mutant models are generated. For each mutation operator, a set of model-checking rules such as reachability or deadlock freeness is defined that distinguishes invalid mutants and eliminates them from test generation. The tool is introduced primarily in Publication IV and Publication V. The tool is modified in Publication VI to support new mutation operators and mutation-selection criteria. This contribution partially addresses RQ3 and RQ4.

1.3 Overview of Original Publications

In this section, a summary of original publications is presented as well as the contribution of the authors towards the publications, the relationship between the publications and how they address the research questions posed in Section 1.1. The research questions are addressed in 6 publications as listed in Table 1.1.

Publication	**Research Questions**	**Case Study**	**Model**	**Testing Activity**
I	RQ1, RQ3	Hotel-Booking	UML (SM) & UTA	MBT
II	RQ1, RQ2	-	-	-
III	RQ1, RQ3	Hotel-Booking	UML(SD)& UTA	Environment model, MBT
IV	RQ4	Hotel-Booking	UTA	MBMT, robustness, Testing tool
V	RQ4, RQ5	Blog	UTA	MBMT, robustness
VI	RQ1, RQ4, RQ5	Blog	UTA	MBMT, security

Table 1.1: Original publications, the research questions they addressed, case studies used, modeling formalism, and testing activity

Publication I introduces an MBT approach for validating a web service composition. The functional requirements of a web service are transformed from UML State Machine (SM) to Uppaal TA, and the model is verified with model-checking. The study answers RQ1 and RQ2 using a `Hotel Booking` web service composition as a case study.

To extend the validation process with various test cases, we study the role of environments in test generation. Publication II is a systematic literature review to understand the state of the art of environment modeling in MBT. RQ2 is answered by defining search criteria and carefully selecting related work on MBT approaches.

The results of Publication II are used to include environment models in

our MBT approach. In Publication III, we transform user scenarios in the form of UML Sequence Diagrams (SD) into UTA and include user behavior in the model of `Hotel Booking`. This study also addresses RQ1 and RQ2.

In Publication IV, we utilize MBT for testing the robustness of web services by combining MBT and mutation testing. From UTA model of the `Hotel Booking`, various unique mutants are generated, each artificially creates invalid test inputs to evaluate the robustness of the implementation of the SUT. We introduce our model-based mutation testing tool, μUTA, which systematically generates mutants from the UTA model. The result of the study answers RQ3.

Publication V reports an experiment of our approach on a different case study, `Blog`, with different properties. The test model is created in UTA directly from the Use Case Diagram. The study answers RQ3 and RQ4.

In Publication VI, the model of `Blog` is extended by adding security properties such as authentication and authorization for users. In this study, we define the security requirements for two users that have access to shared resources via `Blog`. We also improve μUTA by introducing two more mutation operators and mutation-selection criteria. The work addresses RQ2, RQ3, and RQ4.

In the following, we highlight the contributions of the author in each publication.

Publication I: An Integrated Approach for Designing and Validating REST Web Service Compositions

In this publication, we present a model-based testing approach. The target subject in this paper is to model and test a web service composition, such as hotel booking that orchestrates other web services, such as banking, searching, and booking web services. In web service compositions, a user only uses one web service interface, namely a web application to book a hotel and all other interactions among web services are hidden. We show how a web service requirements modeled in the UML modeling language are transformed into UTA models. The transformed model is verified using the (T)CTL model-checking technique. Once the model of the web service is verified, it is used for generating test cases to evaluate the correctness of the implementation of the web service. We develop a test adapter to translate the model-level tests into executable test inputs and execute them with the Uppaal Tron testing tool.

Author's contribution: Faezeh Siavashi applied the transformation of the model and verified it based on the web service requirements. The author also set up a test adapter and performed online MBT and evaluated the approach by comparing the results with the result of applying code-based mutation testing.

Publication II: Environment Modeling in Model-Based Testing: Concepts, Prospects and Research Challenges. A Systematic Literature Review

In this publication, we aim to understand the-state-of-art of using environment models and their advantage in MBT. First, we define a set of research questions to identify the scope of our review, then we retrieve related work, read them and analyze them to answer our research questions. From the result, we conclude that using environment models can be helpful in robustness testing, safety testing, and regression testing. Besides, we discuss what modeling languages have been used in related research to capture environment behavior.

Author's contribution: Faezeh Siavashi defined the research questions, conducted retrieving and reviewing papers from different scientific databases under supervision of Dragos Truscan. The search process included defining search strings, collecting data from the selected papers and analyzing them.

Publication III: Scenario-Based Design and Validation of REST Web Service Compositions

We utilize the results from Publication II to employ environment modeling best practices to design test models. Similar case study as Publication I is used in this study. We define a set of transformation rules that maps from elements in UML sequence diagram to corresponding UTA model separating user behavior (as an environment model) from the web service behavior (the SUT model). We verify that the UTA model complies with the service requirements via model-checking technique and execute tests that are derived form the environment model against the implementation of the web service composition. We evaluate our approach in environment model-based testing by applying a number of code mutations.

Author's contribution: The main idea was developed in collaboration with all authors. Faezeh Siavashi defined the transformation steps from the sequence diagram to a UTA environment model and defined verification rules for the requirements in the model. Furthermore, she evaluated the testing approach by comparing it with the previous study.

Publication IV: On Mutating UPPAAL Timed Automata to Assess Robustness of Web Services

In this publication, the goal is to assess the robustness of web services. We further examine the robustness of the web service composition that its functionality is evaluated previously in Publication I and III. Specifications of the web service composition are modeled in UTA, and online MBT is used

to verify the conformance between the model and the implementation. We adopt, modify a set of well-defined mutation operators for UTA and generate mutants. The mutants provide artificial invalid test inputs to examine the vulnerabilities of the system under test. We evaluate our approach by employing it for testing the web service composition and comparing the results with the test results from the MBT approach in Publication I and III. The experiment shows that the mutating specifications are useful in detecting errors that were not revealed previously in the conventional conformance testing methods.

Author's contribution: Faezeh Siavashi proposed the idea of mutation generation, extracted a set of previously defined mutation operators, and conducted an experiment on testing web service compositions. She evaluated the approach under supervision of Dragos Truscan and Juri Vain.

Publication V: Testing Web Services with Model-based Mutations

In this publication, we improve our MBMT approach by experimenting on a different type of web service, such as a social network, in which the user's activities are intensive and they can create and manage groups, news feeds, and relationships which brings more complexity in testing such web services. Moreover, we modify the mutation generation process by detecting and eliminating equivalent mutants, which have identical input-output behavior with the original test model, thus are not valuable for mutation testing. The process of detecting equivalent mutants is done by applying *bi-simulation*, which executes each mutant against the original model and compares its input-output sequence and criteria. The resulting mutants are then used in the online testing tool, Uppaal Tron to check whether the latter allows for unspecified behavior. The experiment shows that the proposed approach of mutating the specifications is effective in detecting errors both in the system functionality and in the test model.

Author's contribution: Faezeh Siavashi designed and specified the model of the case study from the requirements, verified the model and conducted conformance testing and model-based mutation testing with the model of Blog and its user and evaluated the results.

Publication VI: Vulnerability Assessment of Web services using Model-based Mutation Testing

In this publication, we improve our MBMT approach by experimenting on a different type of web service, such as a social network, in which the user's activities are intensive. As such, users can create and manage groups, news feeds, and relationships which brings more complexity in testing such web services. Furthermore, we modify the mutation generation process by detect-

ing and eliminating equivalent mutants, which have identical input-output behavior with the original test model, thus are not valuable for mutation testing. The process of detecting equivalent mutants is done by applying *bi-simulation*, which executes each mutant against the original model and compares its input-output sequence and criteria. The resulting mutants are then used in the online testing tool, UPPAAL TRON to check whether the latter allows for unspecified behavior. The experiment shows that the proposed approach of mutating the specifications is effective in detecting errors both in the system functionality and in the test model.

Author's contribution: Faezeh Siavashi suggested the idea of the study and designed and verified multi-user behavior of a social web service using UTA model. The mutation-selection criteria were defined with the collaboration of all authors.

1.4 Research Settings

The research approach we followed in this thesis work is a combination of design science, which is more concerned with building artifacts, and empirical studies, which are based on applying experiments. Figure 1.1 presents the testing approach that we present in this thesis including modeling and verification, test generation, test execution, mutation generation and mutation execution. The major part of this research revolves around model-based design and model-based mutation. The presented testing technologies are validated in two industrial projects.

The research on model-based testing was initiated as part of the PAM project [82] including five Finnish industrial partners. The focus of the project was on investigating the capability of model-based testing principles in the continuous integration process in the telecommunication domain. The goal of the project was to evaluate the use of model-based testing to address the increasing complexity of software specifications, to enable test design automation and to provide dedicated tool support for industrial software testing. The author's task in this project was to study MBT for online and offline testing and evaluate a real-time system's behavior based on various test cases. As outcomes of research in this project, Publication I and III are reported.

The next phase of research included conducting a comprehensive study on the principles of model-based testing, their domains, and potentials on advancing test automation. Furthermore, using models for combining mutation testing techniques with model-based testing was investigated in this phase. The Faculty of Science and Engineering funded this part of the research at Åbo Akademi University. Publication II and IV and V are conducted during this phase.

Finally, the last phase of the thesis has been done within MegaM@rt^2 European project [75] which is a large scale project with 27 academic and industrial partners from Finland and other European countries. The goal of the project is to reduce the cost of software development and testing and in deploying scalable model-based methods. The objective of the project was also to assess the vulnerability of intensive software with model-based mutation testing. Publication VI is the result of the research during this phase.

1.5 Structure of Thesis

The thesis is prepared as a collection of peer-reviewed articles and contains two parts. Part I provides a summary of the research, while Part II presents the original publications. Part I continues as follows: Chapter 2 provides essential background and covers essential related work. Chapter 3 presents a summary of the main contributions of this thesis and focuses on answering the research questions. Finally, Chapter 4 presents discussions and evaluations of the study and draws plans for some future works.

Chapter 2

Background and Related Work

This chapter is divided into the following parts. First, we briefly present the concept of testing software quality, its different approaches, and terminologies. Next, an overview of web services is provided including their robustness and security concerns and related work on testing web services. Besides, the preliminaries of modeling frameworks such as UML and UTA formalism are presented and the related studies are provided on using models for model-based testing. Finally, the concept of Model-Based Mutation Testing with UTA is presented and related work in different studies is described. We compare related work on model-based mutation testing with our approach.

2.1 Overview on Software Testing

As mentioned in Chapter 1, software testing is a method of examining the compliance of software products with their specifications in each stage of software development, as well as evaluating them against user requirements. The goal of software testing is to increase the confidence in the correctness of a software product by executing certain test data with the intent of finding inconsistencies. IEEE Standards Collection for Software Engineering [116] has classified different types of anomalies as follows: A human action that creates an incorrect result in a software product is referred to as an *error*. Misconception and misunderstanding of software specifications or incorrect implementations are common causes of errors. The lack of success in a system or system component to perform the required function within specified limits is known as *failure*. It is an externally visible deviation from the system's specifications. Manifestation of an error during software execution is called a *fault* (*defect*). Incorrect processes or wrong data definitions can lead

to faults. Detecting faults has been explored from different perspectives, and various testing approaches have been presented [76].

In most traditional testing approaches, the testing activity accompanies the software development process. A software development process starts with user and system requirements analyses, followed by system architecture, designing components, and implementation [5]. As shown in Figure 2.1, for each stage of the development, different levels of testing are provided. To assess the software with respect to its specifications, *acceptance testing* can be applied, while to evaluate the end to end user scenarios across the components that make up the system, *system testing* can be used. *Integration testing* and *unit testing* can be applied to assess the integration of the components and evaluate the correctness of the individual components respectively.

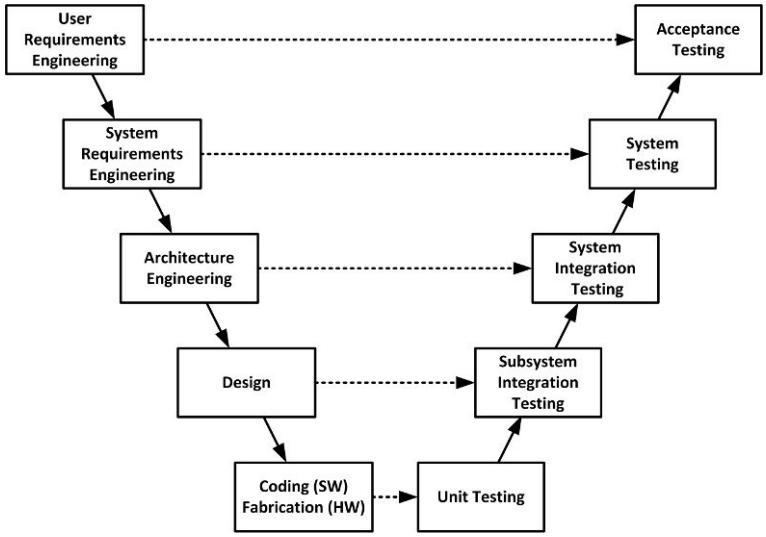


Figure 2.1: Traditional software development process with testing activity. Reprinted [or adapted] from [5]

From a more abstract perspective, testing is divided into two types: *black-box* and *white-box* testing [84]. Black-box testing is a testing method, in which the internal structure of the system under test (SUT) is not known. Thus, the test focus is on the observable behavior of the system under test. In contrast, white-box testing is a testing method in which the implementation of a system under test is known. It is based on an analysis of the internal structure of the system by measuring coverage of code, branches, paths or conditions in the implementation of the system under test.

Many studies have presented tools and frameworks based on test generation strategies and algorithms that are surveyed, same as previously [19, 24, 74, 85]. These frameworks automate the test generation process,

reduce the time of testing, and increase the possibility of finding certain faults that otherwise may not be possible to detect.

Despite the widespread use of testing in the contemporary software industry, ensuring that the software works correctly in practice is challenging. The first challenge originates from the well-known limitation in testing as introduced by Dijkstra: "*Testing shows the presence, not the absence of bugs (faults)*" [21]. In other words, testing may be able to detect a fault in a system, but it cannot ensure that the system is fault-free. Therefore, test inputs should be designed to explore more extensive and more complex behavior of the system.

Software applications are constantly upgraded to newer versions. Usually, large numbers of test cases are designed and executed to examine the correctness of new features. Moreover, the integration of the newly added parts with the rest of the programs should be tested before each new release. As a result, continuous testing of programs and maintaining the test cases are generally expensive and time-consuming, and in many projects, the testing process is terminated before products are tested thoroughly. This problem is evidence of the need for defining and establishing testing methods that can effectively reduce the cost of testing.

In recent years, many new testing techniques are progressively developed that can detect many unknown and hidden faults in systems that were previously tested. Many studies have presented tools and frameworks based on test generation strategies and algorithms that are surveyed, the same as previously [19, 24, 74, 85]. These frameworks automate the test generation process, reduce the time of testing, and increase the possibility of finding certain faults that otherwise may not be possible to detect.

In this thesis, the testing approach involves the system level and acceptance level in the V model in Figure 2.1. Specifications and user requirements of web services are instantiated as models.Our testing process comprises the requirements of the SUT are analyzed to define the test criteria that are used to design test cases. From the test model, abstract test cases are generated and converted into executable test cases. The executable tests create a range of test inputs that are defined in the test design stage. Finally, the tests are executed against the system and results are evaluated.

2.2 Overview on Web Services

Service-oriented architecture (SOA) is a software paradigm where components are created with a well-defined interface, and each component contains a distinct functionality known as a service [25]. Service-oriented applications are *loosely coupled*, meaning that they have little or no knowledge of internal structures of other services [52]. This feature makes SOA applications to be

independent of any specific technology. SOA allows developers to combine and reuse existing services in the production of new applications in an ad hoc manner. For instance, an online shopping system can be created by utilizing other services such as online payment and delivery services that are already provided by other businesses.

A *web service* is an example of SOA offered by a machine or an electronic device via the World Wide Web [15] and intended for machine-to-machine communication. Web services are self-contained applications, can be published over a network and can be invoked or updated remotely via Hypertext Transfer Protocol (HTTP) [61]. Messages between web services can be in machine-readable file formats such as Extensible Markup Language (XML) [38] and JavaScript Object Notation (JSON) [11].

The most common technology choice for transferring messages is Simple Object Access Protocol (SOAP)[15]. A SOAP web service defines the structure of messages with Web Service Definition Language (WSDL) [102]. WSDL creates a machine-readable description of how a service can be invoked, what parameters it accepts, and what data structures it returns whereas SOAP sets up processing rules for information exchanged between the service provider and receivers. Both SOAP and WSDL are XML-based. Thus, all messages are required to be encoded into XML format. The drawback of using SOAP web services is that in web services with intensive communications, processing data to XML format causes massive overhead.

A less restrictive form of SOA is called REpresentational State Transfer (REST), which is more flexible in the format of data transfer in the web [26]. RESTful systems use *stateless* operations, meaning that the server does not need to retain the status of each client's communications. By utilizing the stateless protocol, RESTful web services achieve high performance, reliability, and re-usability. Unlike SOAP, creating REST requests and responses are done merely with HTTP URL connections and the format of data can be in XML, JSON, or HTML. As REST architecture is lightweight and format-agnostic, it has become a popular standard for many companies. Several major companies such as Amazon, Linked In, and eBay have transformed their web service technologies to REST.

Vulnerabilities and Security risks in Web Services

Web services usually concentrate large amounts of sensitive data related to their users and are expected to be responsible for their integrity and security. The security of web services is an important focus of software testing. Due to the distributed and loosely coupled nature of web services, vulnerabilities of these systems should be continuously examined before attackers exploit them [93]. *Vulnerability* is a weakness of an asset that can be exploited by one or more threats, where an asset is anything that has value to the or-

ganization including information resources [48]. A *threat* is an event with the potential to impact operations or assets of a system via unauthorized access, destruction, disclosure, modification of information, or denial of service [48]. Any weakness in specifications, design or implementation could make a system vulnerable to threats.

Authentication and *authorization* are the two main ways of securing a web service and the data it maintains. Authentication is the process of verifying a user's identity, while authorization is the process of proving the user's permission to access resources provided by a web service. For instance, in a social network, when users sign in (authentication), they can manage/edit their profiles, however, to change, or add to shared resources, such as other page's news feeds, group chats, etc., they require access right (authorization). Since such systems are dynamic and change constantly, evaluating the correctness of their security requirements are challenging.

A large body of studies has been presented for testing the security of different types of systems. However, we focus on relevant work that has been done for testing the robustness and security of web services. Fault injection [112], cross-site scripting (XSS) [51], and threat modeling [78] are some of the common methods in evaluating the security of web services. Fault injection is based on inducing faults in a system to penetrate its functionality. XSS is an example of fault injection in which malicious scripts are seeded into web applications to assess their robustness. Several studies such as [34, 69, 93, 94] present different XSS approaches. Threat modeling describes a set of security aspects and possible attacks in a system. This method allows developers to identify risks in early phases of software development [18, 104].

Mutation testing of web services and applications has been advocated as a promising methodology for evaluating their security. A comprehensive analysis is done on all available mutation testing methods by Jia and Harman in [50]. More recently, Papadakis et al. presented the current state of the art in the field of mutation testing, model-based mutation testing, and the open challenges [83]. Mutation can be applied to the communications between client and server and malicious behavior in the communications can be tested [23]. It can be also done in transferred data such as XML messages [63], XML schema [65], or the web semantics [62].

Lee and Offutt defined a set of mutation operators for XML data models to mutate inputs in web components [63]. Li and Miller presented a mutation of XML schema to create invalid XML data automatically [65]. Lee et al. presented ontology-based mutation operators on OWL-S, which is an XML-based language for specifying semantics on web services [62]. They mutated semantics such as data mutation and condition mutation in specifications and compared the behavior of the mutated specifications with the expected one.

The main difference of the testing approach that we are presenting in this thesis and the related work is that in the related work that is discussed above, the input data are mutated, while in our work, we mutate the behavior of the SUT or the environment to check the SUT's expected behavior. For instance, the sequence of the request/response that is designed in a web service is mutated in our approach to evaluating the behavior of the web service in the presence of unusual inputs. Another difference is that the SUTs are built upon RESTful architectural styles, whereas the related work is mainly on SOAP web services. Moreover, we combine MBT with mutation testing and generate mutated tests for our case studies. We extend the testing approach for evaluating security vulnerabilities by modeling the authentication and authorization of users and define mutations for them.

2.3 Modeling Frameworks

Using models as a way of describing the behavior of the system has the advantage of simplifying the design and focusing on more important parts of the system by abstracting the details. Modeling helps to understand and predict the behavior of complex systems with intense internal and external interactions [27]. Modeling has been adopted in developing, testing, and automating the verification and validation processes.

Unified Modeling Language (UML)

UML is a modeling language that has been widely used in both industry and research. It is a standard method of visualizing the design of software systems by abstracting away the details of the requirements [90]. UML provides different types of diagrams that help developers to understand the system from different points of view. Usually, in developing systems, UML models are used as part of the specification to facilitate understanding of some complex parts of the systems.

Some types represent structural information, and others represent general types of behavior such as interactions among components or systems. Depending on the need of the system under design or test, a specific type of diagrams such as *sequence diagram*, or *state diagram* are created. A sequence diagram represents an interaction between components (or objects) in sequential order. A state diagram is used to give an abstract description of the states of a system.

Execution of a system can be defined as a sequence of events that change the state of the system. A set of such execution sequences is known as the *behavior* of the system [4]. Systems with finite numbers of states can be defined with finite-state modeling formalism. Finite automata (FA) and

finite state machines (FSMs) are formal models that are used to simulate the sequential execution of computer systems.

Figure 2.3 shows an excerpt of two UML diagrams, sequence diagram and state machine of an ATM.

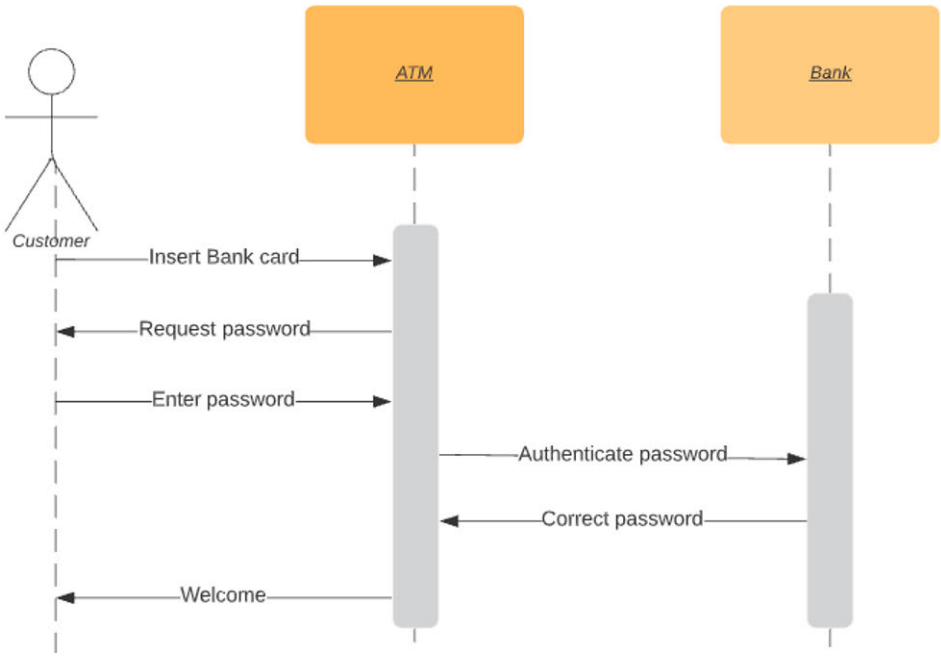


Figure 2.2: Excerpt Sequence Diagram of an ATM

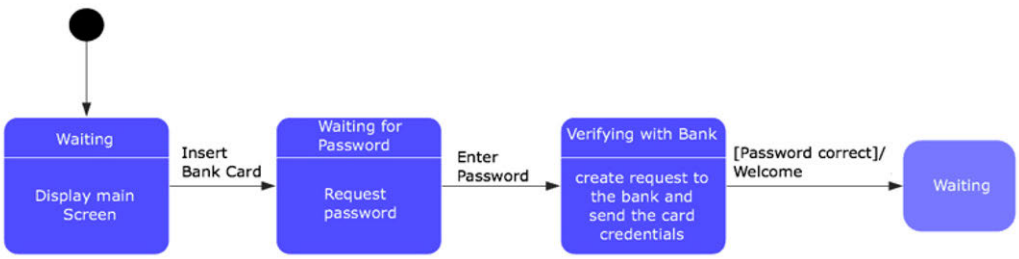


Figure 2.3: Excerpt State Machine of an ATM

Timed Automata Formalism

Although Finite automata and FSMs offer effective constructions and decision procedures for automatically controlling and analyzing system behavior, they do not modeling and verification of support timing constraints. In the

early '90s, Alur and Dill [4] introduced Timed automata (TA) to represent more accurately the behavior of real-time systems.

TA are finite automata that are enriched with several real-time clocks that can be reset during the system execution. A *timed automaton* is defined as a set of locations and directed edges that can connect the locations, and use real-valued variables called *clocks* to measure the elapse of time. A timed automaton can execute individually or in synchrony with other automata. During the execution of the TA, all clocks increase at the same speed. Clocks can be updated or reset along with the edges of TA. Traversing an edge can be constrained by *guards* which enable or disable executing the edge.

If there is more than one enabled edge at a time, then one of them will be chosen non-deterministically. According to [43], this characteristic provides more freedom to design non-deterministic behavior in the systems with arbitrary discrete events, such as real-time systems.

Formally, a timed automaton can be defined as follows:

Definition 1. Timed Automaton (TA) Let C be a set of non-negative real valued variables of n clocks and $\mathcal{G}(C)$ be a set of clock constraints that are conjunctions in form of $x \bowtie c$, where $x \in \mathcal{G}$, $c \in \mathbb{N}$, and $\bowtie \in \{\leq, <, =, >, \geq\}$. Let $v : C \rightarrow \mathbb{R}_{\geq} c$ indicate a real value is assigned to every clock $c \in C$ and Let $\mathcal{U}(C)$ denote the set of *updates* of clocks. A timed automaton is a tuple (L, l_0, I, E), where:

- L is a finite set of *locations* and $l_0 \in L$ is *initial* location;
- $E \subset L \times A \times \mathcal{G}(C) \times 2^c \times L$ is a set of edges including an action, a guard and a set of clocks.
- $I : L \rightarrow \mathcal{G}(C)$ assigns invariants of location.

The semantics of a timed automaton is a *timed transition system* [9], where a transition is denoted by $l \xrightarrow{\text{a,g,u}} l'$, iff $(l, g, a, u, l') \in E$. A state in TA is defined in form of $s = (l, \overline{v})$, where l is a location and $\overline{v}$ is a non-negative clock value vector that satisfies the invariant of l. A TA progresses either by changing from a state to other by executing an edge, i.e., $(l, \overline{v}) \xrightarrow{a} (l', \overline{v}')$, or by staying in a location and passing time, i.e, $(l, \overline{v}) \xrightarrow{d} (l, \overline{v} + d)$, as long as the invariant of location l is true.

In modeling distributed systems, a component can be defined as a timed automaton that can interact with other components. In such systems, a component has little or no knowledge about states or internal transitions in other components. Lynch described such systems as *Timed Input/Output automata* [70]. Actions in such systems are in three types: *input*, *output*, and *internal*. The input and output actions in a timed automaton are used to interact with its environment, whereas the internal actions are only visible to the automaton itself. Input actions come from the environment and cannot

be controlled by the automaton, however, internal and output actions are generated and controlled by the automaton.

A timed I/O automaton can be formally defined as follow:

Definition 2. Timed Input-Output Automata

Timed Input-Output Automata (TIOA) is an extension of TA, in a way that the actions set, A is divided into two sets of inputs and outputs actions, A_i and A_o respectively. Thus, for each input action in the system, there is an output. A TIOA, A is a tuple $< I_A, O_A, L_A, l_A^0, C_A, T_A >$, where:

- I_A is a finite set of inputs, labeled by "?", O_A is a finite set of outputs, labeled by "!",
- L_A is a set of locations that indicates the state of the system after the transition,
- l_A^0 is the initial location,
- C_A is a set of clocks instantiated to zero at l_A^0, and
- T_A is a set of transitions in the system.

Modeling with UPPAAL Timed Automata

Uppaal is a tool providing a platform for modeling, validation, and verification of real-time systems that are modeled as networks of timed-automata [59]. Compared to TA it is extended with data types such as bounded integers and arrays as well as with synchronization mechanisms/actions via channels and committed and urgent locations. The tool has been applied for verification and testing in a wide range of case studies including real-time systems, web services, protocols, and has improved to a mature model checker [14, 44, 88]. The formal definition of Uppaal TA is as follows:

Definition 3. Uppaal Timed Automata (UTA)

An UTA, A_i is a tuple $< L_i, l_i^0, C, A, E_i, I_i >$, $1 \leq i \leq n$, where:

- L_i is a set of locations in A_i automaton,
- l_i^0 is the initial location in A_i,
- C is a set of clocks set to zero at l_i^0,
- A is a set of actions,
- E_i, $E_i \subseteq L_i \times A_i \times B(C) \times 2^C \times L$ is a set of edges between locations with an action, a guard, and a set of clocks to be reset,
- $I = L \rightarrow B(C)$ assigns invariants to locations.

UTA model can be composed to form a *network of UTA* (NUTA) as the parallel composition of n UTA that communicate via shared variables, clocks, and transitions. As mentioned above, UPPAAL extends TA with other data types in addition to clocks. In UPPAAL TA, due to the distinction between local and global variables, it is possible to model systems and their environment as separate interacting automata. Such functionality enables refining the specification of either of a system or the environment without having a significant change in the other. Moreover, various testing goals can be designed in the environment such as safety, robustness, user scenarios.

Figure 2.4 shows an example of a UTA. The model of a system (on the left) and its environment (on the right) can be synchronized by *channels* over edges. Channels are labeled by "!" as emitting and "?" as receiving. In this example, the environment can trigger the system by executing "a!", which is synchronized with the same action as "a?". In response, the system can only execute "b!", which in the environment can be received by "b?". Within a limited time (i.e., $cl \leq 10$), "c?" can be executed by the environment, followed by "b!" from the system and incrementing n by 1. This sequence will continue until $n > 5$, which enables execution of "d!" and taking the environment to the "Final" state. This example shows the potential of UTA to leverage modeling complex systems by supporting parallel transitions of different automata.

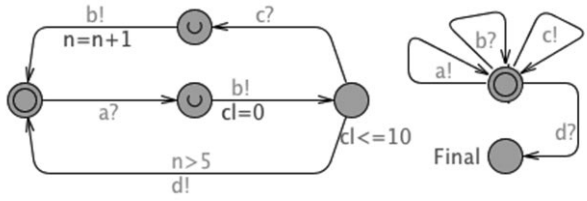


Figure 2.4: An example of a UTA model and its environment

Reachability Analysis in UTA

A UTA model of a system can be analyzed for correctness, i.e. if a particular requirement will be satisfied during the execution of the model. In this thesis, we resort to model checking reachability properties to identify traces of execution that support identifying abstract test cases.

Reachability problem in UTA is represented in terms of a finite *symbolic state space* and *symbolic computation steps*. A symbolic state denoted as (l, D), where l is a location of a timed automaton and D, clocks valuations, represents $\{(l', \overline{v}) | l' = l \wedge \overline{v} \in D\}$. The initial state of the automaton is (l_0, D_0), where $D_0 = \{\overline{v} | (l_0, \overline{v_o}) \xrightarrow{d} (l_0, \overline{v})\}$. The reachability for symbolic state indicates that the location l is reachable at some point in time. A symbolic computation step is defined in the form of $(l, D) \xrightarrow{a} (l', D')$,

representing an action followed by some delay. An action is reachable iff $(l, \overline{v}) \xrightarrow{a} (l', \overline{v}')$ and $D' = \{\overline{v}'' | (l, \overline{v}) \xrightarrow{a} (l', \overline{v}') \wedge (l, \overline{v}') \xrightarrow{d} (l'', \overline{v}'') \wedge \overline{v} \in D\}$.

The UPPAAL model checker contains an engine for verifying such properties [7]. Once a model of a system is verified based on its specification, it can be used for validation of the behavior of SUT or environment it models. UPPAAL utilizes an online testing tool, TRON ,which generates test cases from UTA models and executes them against systems. In the following section, we describe how to use a UTA model for automatic test generation.

In the context of modeling web services, UML is the most commonly used approach for specifying the behavior of web services and their compositions [91]. UML provides a standard for expressing the semantics of web services as described by [6, 54, 71]. Besides, modeling web service compositions with UML are investigated in [36, 101].

Web services are also verified with model checking techniques, which are surveyed in [10] and [92]. Directed graphs have been used by Leymann et al. to model the sequence of interactions in web services [64], while Petri nets are reported in [107, 16] and [95] for designing web services formulated in specification languages. Petri net models were also applied for verification and testing web service compositions in a number of studies such as [37, 68, 79, 96, 114].

Several studies have been done on the transformation of models of web services to verify their dynamic behavior. For instance, [13, 20, 22] present the transformation of models to ensure the correctness of web services with time restrictions. In these studies, the web service compositions are specified by transforming Web Service Choreography Description Language (WSCDL) into Timed Automata and verified the model by UPPAAL. Similarly, in [31] an approach is presented that converts Business Process Execution Language (BPEL) specifications of web services into automata that are bounded with guards, translated into Promela language, and verified with SPIN model-checker. We transform UML models (state machine diagram and sequence diagram) of web services into UPPAAL TA and verify the transformed model by tracing the requirements reachability in UTA.

2.4 Model-Based Testing

According to Utting and Legeard, *Model-based Testing* (MBT) is a testing paradigm that combines formal models and conventional testing techniques while offering faster and cheaper results [108]. In model-based testing, it is assumed that the model can be used as the oracle (i.e, a pivot to determine the correctness of a system) of the corresponding system [113]. Consequently, differences between the behavior of a model and the observed behavior of the system help in finding bugs in the system [73]. MBT is

primarily a black-box testing technique that generates tests from abstract behavioral models. In the domain of safety-critical systems, MBT is becoming an increasingly important technology recommended by IEC-61508-3 [32]. The goal of MBT is to validate that the actual behavior of a system complies with its specification. Such testing approach that verifies whether a product performs according to its specified requirements is categorized as *conformance testing*.

Generally, MBT contains three main phases, *modeling*, *test generation*, and *test execution*. Figure 2.5 shows a diagram including MBT phases. The model of a system specifies the expected behavior of the system or/and its environment. The modeling phase usually includes also verification of the system specification, based on its requirements.

In the test generation phase, test cases are derived from the model. Test cases that are derived from the model are abstract, and they should be later converted into executable test cases. In most MBT environments, a mapping between the model-level (abstract) test cases and executable test cases is implemented by *test adapters*.

In the test execution phase, the executable tests are run against the SUT, and the results determine if there is any difference between the behavior of the model and the SUT by examining actual against expected outputs. If a fault is found during the test execution, the test verdict is considered as *failed*, and the SUT or the model should be debugged (fixed). Once the test cases comply with the SUT, the test verdict is considered as *passed*.

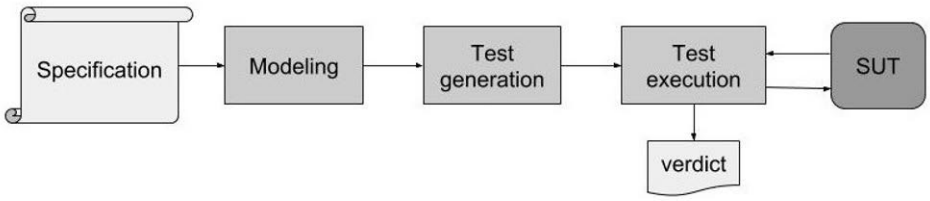


Figure 2.5: Overview of the steps in Model-Base Testing

Online Model-Based Testing With UPPAAL-TRON

Uppaal Tron is an online testing tool from the Uppaal family and is suited for black-box conformance testing of systems with timing constraints [44]. It generates symbolic timed traces of UTA models.

A symbolic timed trace $TTrS$ of a UTA model is a sequence of symbolic states, each state being defined as a tuple (l, D, v), where l is a location,

D is the clock constraints and v a set of non-negative variables' values. Similar to the TA progress described above, a transition in UTA from a symbolic state to another is possible either by an action or by some delay $(l, D, v) \xrightarrow{a/d} (l', D', v')$.

TRON takes the environment constraints of a system into account. Thus, the model of a system and its environment are defined as TIOAs, where the model is split into two parts that of the SUT and the environment. These parts are synchronized by input/output actions. The interaction between the system and its environment are identified as observable actions in TRON. The actions within the SUT (or environment) with other systems are known as internal actions and are not observable. During the execution, these internal actions are abstracted as delays by TRON. Therefore, conformance testing contains delays and observable actions.

Definition 4. Relativized Timed Conformance

For input enabled timed input/output labeled transition systems $i, s \in \mathcal{S}$ and $e \in \mathcal{E}$, relativized timed input/output conformance is defined asl below:

$$\begin{aligned} i \ \mathbf{rtioco}_e \ s \ \Leftrightarrow \forall \sigma \in \texttt{TTr(e)}.\mathbf{Out}(\langle i, e\rangle \ \mathbf{after} \ \sigma) \\ \subseteq \mathbf{Out}(\langle s, e\rangle \ \mathbf{after} \ \sigma), \end{aligned}$$

where $\mathcal{S}$ and $\mathcal{E}$ are TIOAs with observable inputs and outputs, i, s and e are initial states of the implementation under test, specification and environment respectively. `TTr(e)` is a set of timed i/o traces of the environment e, $\langle i, e\rangle$ and $\langle i, s\rangle$ are observable i/o actions that are synchronized. $\langle i, e\rangle$ **after** σ indicates that observable trace σ is executed on implementation i via environment e and $\langle i, e\rangle$ **after** σ means that an observable trace σ is evaluated on the specification s via environment e and returned a set of possible states. **Out**(*states*) contains a list of possible output actions or delays.

The practical implication of the previous definitions is that that the implementation conforms to the specification within a shared environment if and only if the observable i/o behavior of the implementation conforms to that of the specification. Thus, the result of conformance testing with TRON will be one of the three cases: *passed*, *failed*, or *inconclusive*. When the specified behavior in the model conforms with the implementation, the test will pass; otherwise, it will fail. If the output is not in the set of inputs for the environment or no input/output is provided within the defined time (test timeouts), then the test result is interpreted as inconclusive.

2.5 Model-Based Mutation Testing

To achieve a higher quality of software systems in terms of robustness and safety, they need to be tested further under unexpected conditions. To mea-

sure the robustness of the software, one can design several invalid scenarios and test whether the software can operate correctly even with invalid inputs. Modeling all possible conditions may not be feasible or scalable. It is, thus, required to develop automatic construction of invalid test inputs that simulate unexpected conditions. One way to automatically generate invalid test inputs is to combine MBT with mutation testing to obtain a more powerful testing technique, known as *Model-based Mutation Testing* (MBMT) [8] or *specification mutation testing* [12].

In MBMT, the original test model is altered systematically by *mutation operators* creating multiple versions of a model (known as *mutant* models). The mutants can be used for the automatic generation of invalid test inputs that are executed against the SUT. The goal in MBMT is to find whether any invalid tests can pass the testing. Therefore, they can reveal unexpected behavior (i.e., fault) in the SUT. Hence, MBMT can expose the mistakes that are caused by the missed or incorrect implementation of requirements. Once the mutants are generated, the equivalent mutants (i.e., the mutants that behave as same as the original model) should be detected and eliminated to reduce the number of executed tests. Besides, to increase the efficiency of mutation testing, the redundant generated tests should be eliminated, since more generated tests could kill the same mutants. As such, the minimal set of tests that maximizes the fault coverage (or mutation score) can be determined.

Mutation testing has been widely studied for decades and has been improved with a large number of contributions in tools, techniques and empirical studies as reported in [50] and [83]. The central principle in mutation testing is to evaluate the effectiveness of test cases by injecting artificial faults to the SUT. If the test cases are strong enough to detect the faults in the program under test, then they are scored as adequate. Otherwise, the test cases are not adequate (are weak) and should be changed, or new tests should be added to diagnose the undetected faults. Injecting faults involves having a set of rules, called *mutation operators*, that systematically modify the syntax of the program. Each mutation operator is in charge of a specific action such as changing arithmetic operands, negation or omission. Designing mutation operators depends on the type of programming language.

In addition to the programming languages, mutation testing has been studied for specifications or models of a program. The mutation testing for a specification that is defined as models is known as model-based mutation testing. The difference between program mutation and specification mutation is that in model-based mutation testing, the mutants are generated from the test model and not the implementation of the SUT. Therefore, tests that are generated from the mutant models will exhibit different inputs (incl. faulty inputs) against the original SUT. In such condition, if the

SUT does not detect the mutated test cases, then either the SUT is too permissive, or there is a fault is an implementation.

In the following, we present the studies on creating mutation operators in TA and use them to exemplify some of the mutant models.

Mutation Operators for Timed Automata

From a verified test model, numerous altered versions can be constructed and the tests that are created from the mutants are also mutated [8]. Aboutrab et al. and Aichernig et al. defined mutation operators of UTA in [1] and [3], respectively. Table 2.1 presents lists of mutation operators defined in these studies. Aichernig et al. defined some generic mutation operators, which slightly alter a given TA. Clocks are considered as same as other variables and the mutation operators that are applicable for variables, are used for clocks as well. For instance, $\mu_c g$, $\mu_n g$, $\mu_c i$, and $\mu_i r$ mutate clock variables. Aboutrab et al. specifically focused on timing properties in their study. STC, WTC, RTC, RC, and NRC are defined only for clock variables in TA. They also added three mutation operators for mutating actions. However, they did not define any operators for other model elements such as data variables, condition negation, or location.

Table 2.1: Mutation operators of timed-automata

Aichernig et al. [3]	**Aboutrab et al.** [1]
Change Guard($\mu_c g$)	Shifting Timing Constraints (STC)
Negate Guard ($\mu_n g$)	Widening Timing Constraints (WTC)
Change invariant($\mu_c i$)	Restricting Timing Constraints (RTC)
Change action ($\mu_c a$)	Exchanging Output Actions (EOA)
Change target ($\mu_c t$)	Exchanging Output Actions (EOA)
Change source ($\mu_c s$)	Exchanging Output Actions (EOA)
Sink location ($\mu_s l$)	Resetting a Clock (RC)
Invert reset ($\mu_i r$)	Not-Resetting a Clock (NRC)

The elements in TA include locations, actions, guards, and invariants. Based on the requirements of the case study and the SUT, for each element, various mutation operators can be defined. Adding a new element entity or removing an entity are additional mutation operators that can be applied for each element.

The operators given in these two studies are different in the type of alternations that are exercised in each element. For instance, Aichernig et al. define a general mutation operator (μcg) for guards, while Aboutrab et al. categorize three mutation operators (RTC, WTC, and STC) for timing guards. Moreover, since the evaluation of real-time systems was the main focus of the work in Aboutrab et al. [1] case study, the mutation operators

are defined for timing constraints, while Aichernig et al. [3] define more diverse and general mutation operators.

We developed a selection of the mutation operators of TA presented by these studies for testing the robustness of web services. Similar to [3], we applied mutations on non-deterministic models, however, in their work, they use only the UTA model of the IUT and do not consider the environment. In our approach, each mutant is a closed model communicating with its environment as well as other systems.

MBMT has been studied and advanced in other modeling languages as well as optimized in test generation and test execution. A comprehensive study is conducted in [83], illustrating a wide range of case studies, tools, and empirical studies in the literature.

The TA mutation operators in [3] and [1] have been adopted, extended and applied also in other studies, such as [66] and [60]. Lindstrom et al. extended the MBMT for timed automata for aspect-oriented models. In their approach, each aspect of a system under test is modeled as an individual automaton connected to a base automaton [66].

Larsen et al. use the mutation operators with the tool Ecdar, which belongs to the Uppaal tool family. They used it to perform conformance checks between the correct specification and the mutants using time refinement [60]. Belli et al. proposed two elementary mutation operators for directed graph-based models (e.g., FSM, ESG) [8]. They categorized the mutants into several groups after the test execution. Even though the mutation operators are roughly defined to cover generating all possible mutations, remarkable operators are introduced in our thesis. We developed three new mutation operators: *remove guards (RG)*, *remove actions (RA)*, and *Duplicate Actions (DA)* to create additional mutations which provide valuable results to our study. The details of our work are presented in Chapter 3.

Chapter 3

Contributions of the Thesis

In this chapter, we explain the research contributions that are included in this thesis. As described in Chapter 1, the goal of this thesis is to present a model-based testing approach, which automates the verification and validation of web services and allows assessing their robustness and vulnerabilities. Because web services have intensive interactions with other systems or users, we study how to model web services concerning the behavior of their environment. We utilize environment models to create test cases with different user scenarios and validate web services. The contributions of this thesis are validated in two different case studies; each exemplifies certain features of web services.

Case Studies

The first case study is a collection of three different web services, i.e., Web Service Composition (WSC) of a hotel booking system, initially specified in Publication I. The requirements of the WSC are specified using Unified Modeling Language (UML), and a code skeleton is generated with partial automation with the Django Framework. The `Hotel Booking` is a RESTful web service, created to have a case study similar in complexity to real services. The case study provides different functionality such as booking a hotel room, payment through connecting to a bank service, checking-in, cancellation, and refund within specific timing constraints. The case study is used as the base of the empirical study on modeling a WSC, transforming the models for verification of timing requirements, and validating the implementation of the WSC. Moreover, the case study is used for further research on the robustness testing with model mutations. The case study is presented in Publication I, Publication III, and Publication IV.

The second case study is `Blog` web service that is a representation of social networks. The `Blog` web service is also implemented in the REST architectural style using the Flask web developing micro framework [35]. `Blog`

supports functionality such as creating a user account, posting new articles, commenting, deleting/editing posts and comments, managing user profiles, similar to the real social networks. Besides, it supports the authorization of users to access the content of the web service. The case study was utilized to model and test both individual and multi-user behaviors based on their authentication and authorization to the resources. The case study is presented and used in Publication V and Publication VI.

Each case study is used to investigate specific properties in web services. In `Hotel Booking` web service, the focus is firstly on the design and validation of a composite of some web services. The model of the WSC consists of three web services and one user behavior. With 1a single user's behavior, all the requirements of the WSC are verified and tested. In contrast, in `Blog` web service, the activity of users can influence the content of the web service. Users can set, change, or remove their relationships in groups, events, or friendships. Consequently, users create and manage the privacy of their information on social networks. Thus, not only the web service should correctly operate based on the requirements of a single user, but it also should be validated based on its intensive interactions between multiple users and their information security.

In the following sections, we provide a summary of the main contributions of this thesis with a brief overview of the challenges that they address. The detailed presentation of the contributions is in Part II, The Original Publications.

3.1 Modeling and Verification of Web Services with UPPAAL Timed Automata

As the first contribution in this thesis, we studied how to model and test a web service composition (WSC) that carries timing constraints. We used the Uppaal model-checker as the supporting tool for our research because it enables modeling time constraints and verification. It also supports the modeling of a system under test and its interactions with other systems (as the environment).

Figure 3.1 shows a part of the `Hotel Booking` WSC (as described earlier) modeled in UML [86]. The model's states represent the states of the WSC. The transitions in the model are the requests between the services in the composition. Note that the states and transitions in UML directly correspond to the original user and system requirements.

We converted the UML model into the Uppaal TA model by mapping the UML elements to the Uppaal TA elements. As such, the UML states are mapped to Uppaal TA locations, and the UML transitions are mapped into edges. The UML state invariants are added as location invariants in

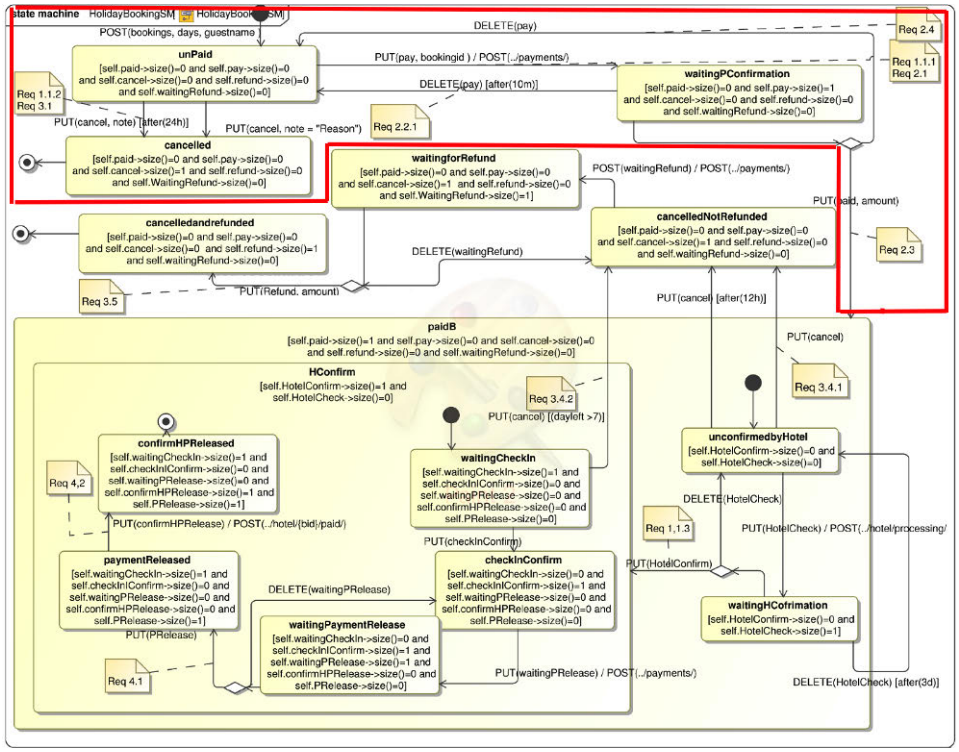


Figure 3.1: Excerpt of the UML model of Hotel Booking ([86])

the Uppaal TA, and the timing events in UML are defined as clocks in the Uppaal TA model. The user requirements are traced by defining Boolean variables called *traps* that update on the edges.

In addition to mapping the elements, in the Uppaal TA model we added user's behavior as the environment of the WSC model. The environment model triggers the WSC by emitting channels that simulate the interface method calls in the UML state machines. The receiving side of the channel is specified in the model of the web service. In return, the desired response of the request is emitted from the model of the web service to the environment model. Simulating these automata initiates the request from the user.

In order to exemplify this approach, and yet keep the example compact, we present how the region marked in red in the UML model in Figure 3.1 is converted into Figure 3.2 as the Uppaal TA model of the web service composition and Figure 3.3 as its environment model. The details of this contribution are presented in Publication I.

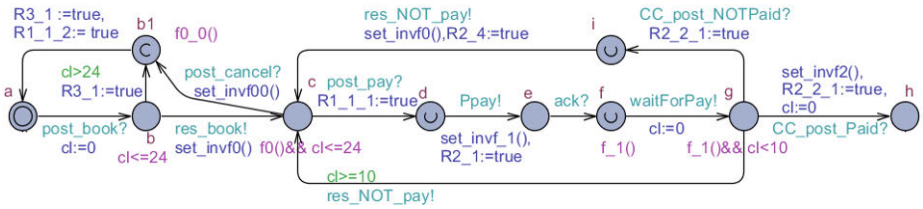


Figure 3.2: UTA model of Hotel Booking transformed from Figure 3.1

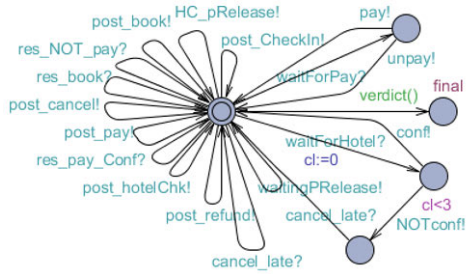


Figure 3.3: The environment model of Hotel Booking ([86])

Transforming User Scenarios from UML to UTA

One way of specifying user scenarios of a web service is to use UML Sequence Diagrams (SD). The UML SD supports communications among different entities in a system as well as their timing constraints such as deadlines. As a part of the first contribution, we define a set of mappings to transform a UML sequence diagram to an Uppaal TA (SD→UTA) as below:

LifeLines: Each SD may have several lifelines, which are categorized into two parts: the SUT and the environment. Messages that are exchanged between the groups create the testing interface.

Messages: For each input message to the SUT group, we define an edge in UTA. The edge is labeled by the name of the message and annotated as an emitting channel (!). Similarly, for each output message from the SUT group, we define an edge with a receiving notation ("?").

Fragments: In SD fragments (i.e., alt, loop, opt), for the number of conditions in the fragment, we define edges from a location and use the fragments' conditions as the guards on the edges.

Timing Constraints: The timing constraint and the duration constraint are transformed into location invariants and edge guards in UTA.

Multiple user scenarios from different SDs can be included in one environment model.

The resulting Uppaal TA environment will have channel synchroniza-

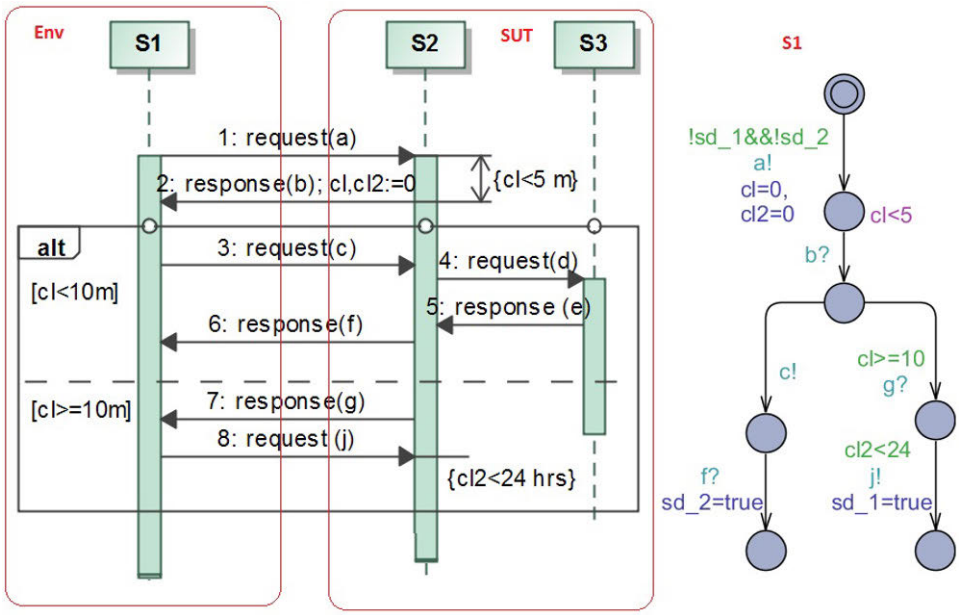


Figure 3.4: Example of SD of three services (left) and the UPPAAL TA model of S1 as environment (right)

tions matching the SUT model obtained in the first transformation. Figure 3.4 illustrates an example of an SD and its mapped model in UPPAAL TA.

We add traps as tracking variables for each user scenario in the UTA model to measure the test coverage. If a scenario has more than one exit point (alternative), then a variable will be assigned to each exit point. A tracking variable is an updated tuple *(sd_no = false, sd_no = true)* on the first edge in the scenario trace and respectively on the last edge in the trace. In the UTA model, the variable is initiated to false and will be set when its corresponding condition is satisfied.

Modeling Multi-User Interactions in UPPAAL

In web services such as `Hotel Booking`, it is enough to model the behavior of only one user and create test cases. However, in more intensive web services such as social networks with intensive user interactions and shared resources, modeling behavior of just one user does not cover all essential test scenarios. Thus, we extended the design of the test models by adding the behavior of multiple users. Similar to real web services, the model supports non-deterministic user activities with shared resources as well as their authorization and authentication. Adding these new properties to the model extends the evaluation of the `Blog` with exhaustive test cases that not only validate the behavior of the system with single user requirements

but also the privacy and security requirements of multiple users.

An excerpt of two user interactions within a blog web service is illustrated in Figure 3.1. From a user's point of view, the activities are defined as posting articles, commenting, reading, editing, and deleting. In the web service's side, the events are in the form of HTTP requests that trigger corresponding functions creating or modifying resources or interacting with other systems.

The automata described the behavior of two users (Figure 3.1) that are distinguished with two different *user-id* parameters. The model contains shared variables for tracking the shared resources that are accessible by both users. Each user can create new resources such as adding a new article (or add judgments) or have access to available resources arbitrarily. To enable such functionality, we defined the following global variables:

- *cu, sa, su* and *sc*, indicate "current user", "selected article", "selected user" and "selected comment" respectively. These variables enable random selection of users, articles and comments.

- *Users*, *max_art* and *max_cmt* variables indicate the number of users, maximum number of articles and maximum number of comments, respectively. Bounding the maximum numbers in the model prevents the possibility of a state space explosion.

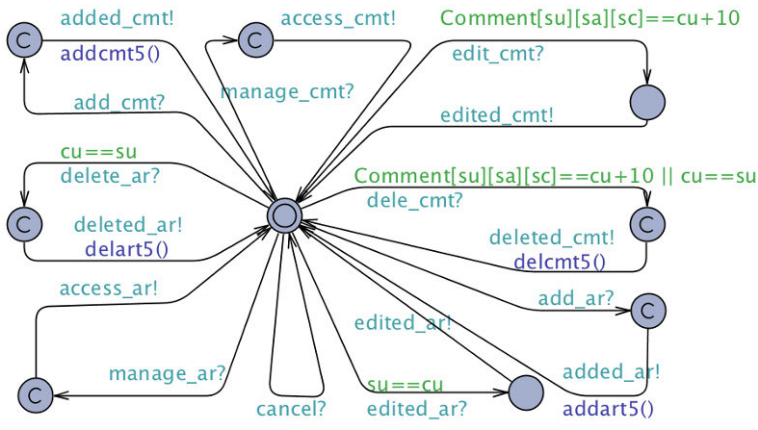
- $article[Users][max\_art]$ assign articles to the users, thus, *article[2][3]* means that each user can create up to three articles. For instance, *User1* can create three articles which are mapped to *article[0][0], article[0][1] and article[0][2].* The initial values are set to zero.

- $Comment[Users][max\_art][max\_cmt]$ contains data about the *user-id* of the person who comments on the articles. Thus, when *User1* (with id=0) adds a comment on *article 1* of *User2*, then we have *Comment[1][0][0]=0+x.* The constant value x is an offset number that prevents confusing the initial values (zero) with the *user-id* (id=0).

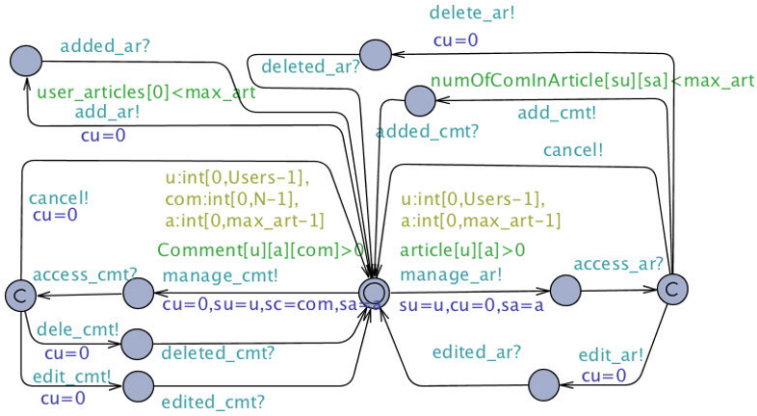
The *article* and *Comment* variables are updated by functions $addart()$ and $delart()$, $addcmt()$, and $delcmt()$. As it is shown in user automata, the values of *user-id* and *article* are selected randomly: $u : int[0, Users - 1], a : int[0, max\_art - 1]$

Verifying Web Service Requirements with UPPAAL Timed Automata

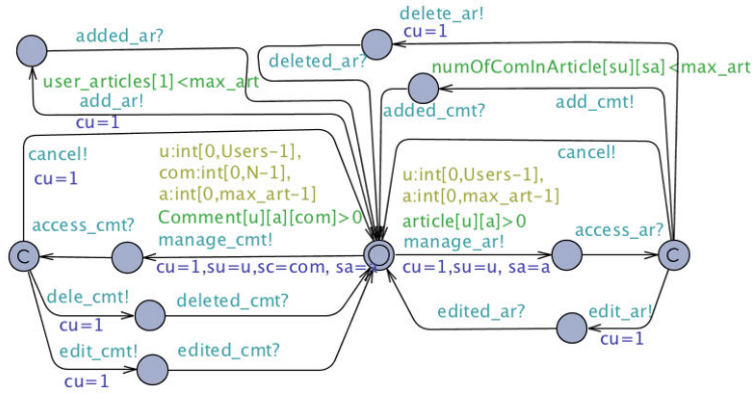
Once the requirements are modeled as service goals, they should be verified to assure that the service does what it is required to do. By defining verification properties for the service requirements at the design phase and



(a) Model of a Blog web service



(b) Model of Blog user1



(c) Model of Blog user2

Figure 3.5: Excerpt model of two users interaction with a Blog

proving them in the validation stage, we can recognize whether the model behavior complies with the service requirements and if not, the fault can be traced in the design stage.

An UPPAAL model can be verified concerning the reachability, deadlock-freeness, liveness, and safety properties. The query language used in UPPAAL is a simplified version of TCTL [4] that enables specifying state formulae that are interpreted by model checker in states and path formulae interpreted over model execution paths. The UPPAAL model checker contains a verification engine named *VerifyTA* that checks whether a query (specified as TCTL formula) is satisfied, meaning that the corresponding query requirements are satisfied.

As an alternative to encoding properties to TCTL formulae, the requirements can be defined also as simple boolean variables which are set to True, whenever the corresponding requirement is satisfied during the simulation. Moreover, a set of requirements can be defined as invariant in specific locations, restricting transitions. These requirements are represented in UTA so that the links between requirements and the model elements are preserved. These requirements are included in all the models and traced throughout the process, i.e., at UML, UTA, and test level, respectively. A detailed description and examples of verification queries for the `Hotel Booking` case study is presented in Publication I.

3.1.1 Testing Web Services Compositions With Uppaal Timed Automata

Generating tests from a model requires a test adapter. As models abstract the details of a system, the test cases that are generated are symbolic and not executable as actual test cases. In online MBT, a test adapter is a script that converts symbolic test cases into executable tests and vice versa on-the-fly. We used the UPPAAL TRON online MBT tool, which provides a test environment to connect the model to the SUT. TRON utilizes a test adapter and communicates the inputs and outputs to/from the SUT based on the test generation algorithms.

Once the model complies with the requirements, it can be utilized for generating test inputs. If the behavior of the model execution conforms to the behavior of the implementation of the web service, then the system under test is validated.

3.2 Exploiting Environment Modeling in Model-Based Testing

Intensive systems usually operate in environments with large numbers of events and with different timings. Using environment models to understand the actual behavior of such systems and to create tests accordingly has gained attention in research in recent years [40, 41, 44, 45, 49]. Therefore, the second contribution of this thesis is to identify the use of environment modeling in model-based test generation. The primary objective presented in Publication II is to understand how an environment model can enhance the MBT approach, to which types of systems it can be applied, at which testing levels it can be supported, and what are the challenges in defining it.

We first defined a set of research questions to clarify the scope of the review. Then, we defined keywords for searching the scientific databases to retrieve all relevant studies available. By reviewing the suitable studies, we obtained information about the environment modeling, which is presented below.

Role of Environment Model in Testing

An environment model has several fundamental characteristics that define its behavior. A brief overview of these properties is presented below. Such statistics are useful in modeling different types of systems and understand the actual behavior of the systems.

Modeling specific aspects of the SUT: An environment model can be specified in such a way that only the specific part(s) of the SUT will be tested. It can contain a test scenario (test suite) to violate certain behaviors or functionality of the SUT.

Non-determinism: Some studies present environment models with non-deterministic behavior. Non-deterministic models are especially useful in modeling the environment of systems which have continuous and unpredictable interactions with their environment [28].

Include multiple entities: We found that some of the papers presented their work using different environment entities, such as users, other systems, or a part of the actual environment (i.e. temperature or the sunlight). There are two ways in specifying multiple environments in a test model: it can be defined as a single model containing all environment interactions, or it can be defined as multiple environment components. In both cases, the environment model should capture all assumptions of the environment.

Dynamic and static behavior: It is reported that environment models can have the support of dynamic and static behaviors. Static behaviors mostly indicate what are the inputs of the environment model into the SUT

and what type of data and properties are supported by the environment model, whereas dynamic behaviors model the interactions of the environment model with the SUT, the timing properties and the order of test inputs base on the current outputs at the time of testing [115].

Abstraction: Abstraction in environment models can be achieved by ignoring the internal interactions of the SUT. An abstract model can be defined by restricting the range of input values, omitting some functions or reducing the time span. Creating an environment model is one of the most challenging tasks in model-checking because the model should be precise and at the same time abstract [111]. Multiple studies emphasize the importance of the abstraction level of the test model when using environment models. Utting et al. point out the different abstraction issues related to MBT [108].

Control of time: Environment models can be used for testing the systems with timing properties, such as real-time systems. Environment models generate timed input traces, which can occur in the real environment, to ensure that the system can satisfy the specified timing properties.

Explicit behaviors: Having separate models for the SUT and its environment has advantages in modifying each of them separately. For instance, when environment models are used in test generation, they typically encode test goals. Whenever test requirements change, only the environment models should be changed. Environment models used in MBT have explicit nature depicting the expected behavior of the system.

Source of knowledge for modeling: The source of knowledge about an environment can come from the requirements or the assumptions of the test designers. The requirements are a list of the specifications that a system must follow and need to be tested. However, when the system specifications are not available, the assumptions of the environment can be observed from the real environment and then formalized as a model. Several studies claim that their environment models are extracted from the requirements that are provided in the documentation of the SUT. They explicitly claimed that they define their environment via assumptions.

Advantage of using Environment Modeling in Testing

Based on the MBT taxonomy illustrated by [108, 72], testing of a system using MBT consists of three main activities: modeling, test generation, and test execution. Based on our findings, environment modeling brings the following benefits:

Test oracle creation: In MBT, a test oracle is usually encoded in the test model, and during test generation, it is assigned to the generated test cases [56, 108]. Thus, an environment model is useful in automatically creating test oracles. A few primary studies discussed explicitly test oracle generation using environment models.

Automated test generation: Environment models are used to generate test inputs for the SUT during testing. It prevents human errors that might occur with manual testing and reduces the time of generating test cases. Furthermore, two papers claim that the advantage of automating test generation can lead to a *test harness*, which is a collection of test data used to automatically run the tests and monitor the outputs [17, 53].

Optimal test generation: Optimized test case generation is discussed as a benefit of using environment models in testing, making the testing process more efficient. It is caused by having the support of abstraction in the environment modeling.

Reducing the size of the state space: One of the main issues in execution and simulation of complex models (or models with a wide range of inputs) is that the number of symbolic states that should be explored increases during the test execution, and thus the memory of the test machine will be filled. This problem is known as the state space explosion. It is reported by multiple studies that well-defined environment models significantly reduce the search space by constraining the ranges of specific test inputs. Reducing the size of state space can be done by defining a limited range of numbers instead of defining a general data types, resetting the clock variables after passing the corresponding state, or defining model invariants which limit the enabled states at a given time.

Early validation of requirements: Using explicit environment models can help validating the requirements in the early stage of the system design. They can be used to guide the simulation of early prototypes of the SUT. Such characteristic has the advantage of using environment models.

Re-usability: It is reported that with the same environment model, different SUTs, or different versions (regression) of the same SUT can be tested. Generally, environment models will be changed relatively rarely unless some errors are discovered from the requirements during testing. Therefore, the modeling efforts can be reduced by using the same models in different testing contexts.

Identifying Application Domain of Environment Modeling

The types of systems can be divided into two categories, the systems containing hardware and software, and systems which are only software. The first category includes different systems as introduced below:

Real-time Systems: Systems with timing properties and activities that must be performed within specific timing constraints [67].

Reactive systems: Systems with continuous behavior interact with their environment are known and reactive systems. A reactive system receives inputs from the environment and based on its configurations, changes its internal states, and sends the outputs as results to the environment [55].

Hybrid systems: Systems that combine two or more different systems are named hybrid systems. Hybrid systems may perform both continuous and discrete behaviors [42].

Embedded systems: Systems with dedicated design or function for a part of the hardware, dealing with the control of the physical environment through sensors and actuators [33].

Embedded systems can act as both reactive systems and real-time systems. Thus, we present them in either the domain of real-time systems (for the real-time embedded systems) or reactive systems (for the reactive embedded systems).

In the second category we have:

Software systems: Software systems are applications or programs that are based on interactions between their components, describing a system of a part of a system [39].

A large number of studies are on applying environment models in testing real-time and embedded systems. They present using environment models in testing time-constrained systems. It is because the environment models can control the timing of the SUT and also provides test cases to verify timing properties.

Some other studies used environment models for testing reactive systems, that contain continuous communications with its environment. Only a few studies were found experimenting with environment models on hybrid systems.

It has been reported in several papers that environment model is used in testing software components. When the entire software is not complete, and a component needs to be tested, then an environment model represents the observable behavior of other components in order to test the component.

Formalism and Tools

The formalism and modeling tools that have been studied are in the context of MBT. The results show that most of the studies use UML or its various profiles as the base of environment modeling. Timed Automata, in particular, UppaalTimed Automata is also a popular formalism in modeling and simulating test models with their environment. One of the advantages of using Uppaalis that it allows the simulation and verification of timing properties of real-time systems. Another major work in investigating environment modeling is done in attributed event grammar, in which the environment is based on a set of event grammars and applied for safety assessment in different case studies. In Table 3.1 the formalism and their tools that use environment models are listed. Further details of studies that are included in this table is presented in Publication II.

Table 3.1: Formalism and tools for environment modeling

Formalism	**Tools**	**# of studies**
UML	UML diagrams	10
	UML/MARTE	5
	UML Fondue	2
	UML/SysML	1
	ESML	1
	MbRTE	1
Timed Automata	UPPAAL	4
	UPPAAL TRON	1
	Maude	1
Event Grammar	AEG	6
Petri nets	TINA	4
Lutin	Lurette	3
Java	BEG	2
QR	QR models	2
TSML	AUTOSAR	1
Esterel	Esterel	1
SPIN	Promela	1
TML	JUMBL	1
Markov model	Markov model	1
TTCN-3	TTCN-3	1
SLAM	SLAM	1
DoB	Degree-of-Belief(DoB)	1
BLAST	BLAST	1

Identified Problems and Challenges

The studies report that even though the environment modeling assist in automatic test generation, still manual testing should be done in some of the case studies. The translating of the symbolic test cases (i.e., model-level test cases) into executable test scripts is a manual process. There is a shortage of empirical studies on utilizing environment models in regression testing for different types of systems. Another issue is on re-usability of environment models in testing new versions of a software/hardware as suggested by [41].

There are few reports about using environment models in hybrid systems and orchestrating communications among diverse applications with the environments as well as communication among the environment entities are interesting research subjects that have not been studied. There is still plenty of potential for investigating environment modeling and automating test generation, especially w.r.t. non-functional testing approaches. Extensions of the current methodologies are needed to overcome these limitations.

In this thesis, we used environment modeling for specifying user behavior to test web services. In Publication III, we identified user scenarios as an environment model (Figure 3.4) to test the `Hotel Booking` web service. In Publication VI, multi-user behavior is modeled as the environment `Blog` to

test the security of web service.

3.3 Using Model Mutations for Assessment of the Robustness and Security of Web Services

Web services are the type of systems that share resources with other systems or applications through loosely coupled communication protocols. This kind of systems should be robust against erroneous inputs. Not only the expected behavior of the implementation should be tested, but it also should not contain any unexpected behavior. The functionality of the system can be checked by running test cases derived from the specification while finding unexpected behaviors of the system can be done by robustness testing, which executes invalid inputs and detects the vulnerabilities or unexpected behavior of the implementation.

To this extent, the third contribution of this thesis is the method of mutating the test models to evaluate the robustness and security of web services. The idea behind using mutations is that since the model is previously verified and used for testing the SUT, it can be considered as the "correct" model. Any other model (i.e., mutant) which does not behave identically is expected to be rejected during conformance testing against the SUT. Thus, if a test that is generated from a mutant is passed, it indicates that the SUT accepts a mutated behavior, which should be analyzed for any possible defects.

Mutating Uppaal Model for Test Generation

We combined the model-based testing with the mutation testing using a selection of mutation operators presented in Chapter 2. Besides, we introduced three new mutation operators for UTA to generate additional mutations. The goal of this combination is to create copies of the "slightly faulty" model. The execution of the mutant models will generated invalid test inputs. If the implementation complies with the mutations without raising an exception, it indicates that it accepts an unspecified sequence of inputs.

Mutation Operators

Mutation generation process of TA for testing dynamic behavior of real-time systems was first presented by Nilsson et al. in [81], and various mutation operators on timed automata elements are formally defined by Aboutrab et al. [1] and Aichernig et al. [3] as are listed in Table 3.3. We categorized the mutation operators of the studies based on the model element and compared the operators.

Table 3.2: Comparing mutation operators of timed-automata in two different studies

Elements	**Mutation operators defined by [3]**	**Mutation operators defined by [1]**	**Informal definition**
Guard	Change Guard(μCG)	Restricting Timing Constraints (RTC)	Restricts, expands or alters guards
		Widening Timing Constraints (WTC)	
		Shifting Timing constraints (STC)	
	Negate guard (μCg)	-	Guard will be replaced by its negation
Invariant	Change invariant(μCi)	-	Restricts, expands or change value of invartiants
Clock	Invert reset (μIr)	Resetting a Clock (RC)	Removes or adds clock resets
		Not-Resetting a Clock (NRC)	
Action	Change action (μCa)	Exchanging Input Actions (EIA)	Changes names of actions
		Exchanging Output Action (EOA)	
	Change source (μCs)	-	Changes source location of actions
	Change target (μCt)	Transferring Destination Locations (TDL)	Changes target locations of actions
Location	Sink location (μSl)	-	Makes a new locations and changes targets of all actions to the new location

From the available mutation operators, we used a collection of operators that are suitable for our testing approach and added some new more mutation operators to the list. The new mutation operators generate additional mutants that cannot be provided by other operators.

During testing the `Hotel Booking` case study, we selected seven mutation operators: CG, NG, CN, CT, CS, CI and IR as presented in Publication IV and V. In Blog case study, in addition to the previous mutation operators, we extended CG (CGL and CGV) and introduced three new mutation op-

erators: RG, RA, and DA as reported in Publication VI. Below, we present the description of the mutation operators implemented in μUTA.

- **Change Guard (CG):** We followed the definition of μCG defined by Aichernig et al. [3] that changes the clock constants in guards by a random value. It is useful for mutating the enabling condition of actions. This operator is used in Publication IV and V. In Publication VI, we extended CG as follow:

  - **Change Guards Logical operators (CGL):** This operator changes the logical operators (i.e., $==, <=, >=, !=, < and >$) in guards which creates additional mutants.
  - **Change Guards Variables (CGV):** CGV alters the value of the variables that are used in guards and creates additional mutants that cannot be defined by other mutation operators.

- **Negate Guard (NG):** It is the same operator as μNG which negates the guards and may cause some paths of the test model to become unreachable.

- **Change Name of actions (CN):** This operator replaces the name of an action with the name of some other actions. Thus, the expected sequence of the inputs to the implementation will be different.

- **Change Targets of actions (CT):** It changes the target location of an action to another location. This operator breaks the flow of test inputs and violates the state of the model. This operator can mutate both input and output actions.

- **Change Sources of actions (CS):** This operator changes the source location of an action to another location. Similarly to CT, this operator mutates the sequence of input/outputs.

- **Change Invariant (CI):** Similarly to μCI, it shifts values of the invariant to a different range, extending or restricting the constraints of the model. It can cause actions to fire earlier (or later) than expected by the original model.

- **Invert Reset (IR):** This operator is the same as μIR, which deletes the resetting of the clock and moves it to one action before or after. It means that the resetting is shifted one edge earlier or later.

- **Remove Actions (RA):** This mutation operator randomly deletes one action at a time and creates a mutant. Omitting an action will manipulate the sequence of input/output actions.

Element	Mutation Operator		Description
Guard	RG*		Remove guards
	Change Guard (CG)	CGL	Change guards logical operators
		CGV	Change guards variables
	NG		Negate Guard
Action	CN		Change Name of actions
	CT		Change targets of actions
	CS		Change sources of actions
	RA*		Remove actions
	DA*		Duplicate actions
Invariant	CI		Change Invariant
Clock	IR		Invert Reset

Table 3.3: Mutation operators in μUTA (* new operators)

- **Duplicate Actions (DA):** This operator randomly copies an action in different parts of a model, thus alternates the sequence of input/output by repeating actions in unexpected states of the model.
- **Remove Guards (RG):** This operator randomly selects an action and removes its guard. Actions that are mutated by RG will always be enabled.

  In Table 3.3, we list mutation operators that are implemented in μUTA.

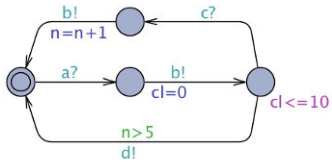
Figure 3.6 shows the generated mutants of a model and sample mutants using the above operators. In our approach, we only apply the first-order mutation. That is, a mutant model contains only one mutated segment based on a single operator.

Mutation Classification

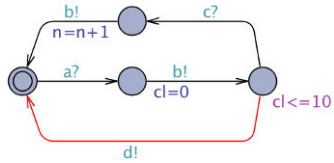
Belli et al., [8] provided an extended classification of the mutants for MBMT after executing them against the SUT. Such designation elevates the quality of testing and distinguishes whether the faults are raised by the mutants or by poor implementation of the SUT. In this thesis, however, we start classifying the mutants in an earlier stage: mutation generation. The reason behind this is to identify more suitable mutants from the beginning and reduce the time of test execution and thus attain more efficient testing.

Figure 3.7 gives the classification in each step of our approach, starting from the mutation generation until the test analysis.

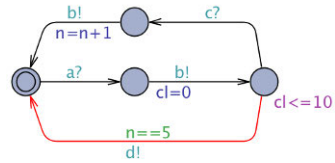
After the model is designed and its conformance with the implementation is approved, we used the mutation operators to generate all possible



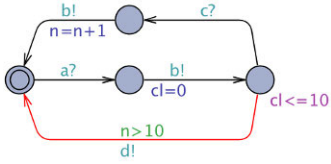
(a) The original model



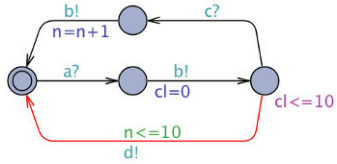
(b) RG: Remove Guard($n > 5$)



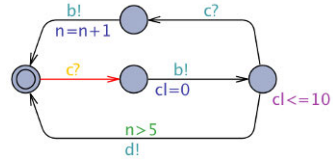
(c) CGL: Change Guard Logic ($n > 5$)



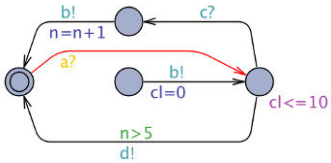
(d) CGV: Change Guard Variable ($n > 5$)



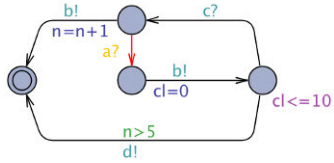
(e) NG: Negate Guard ($n > 5$)



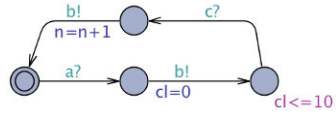
(f) CN: Change the name of input action (a?)



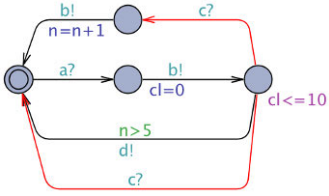
(g) CT: Change Target (a?)



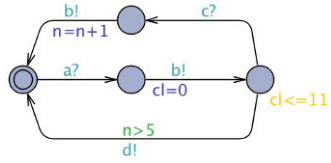
(h) CS: Change Source (a?)



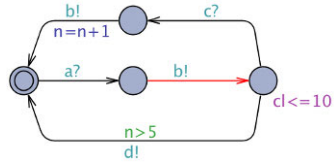
(i) RA: Remove Action (d!)



(j) DA: Duplicate Action (c?)



(k) CI: Change Invariant ($cl <= 10$)



(l) IR: Invert Reset ($cl = 0$)

Figure 3.6: A model with examples of mutants generated by a number of mutation operators.

combinations of mutants. Then, the reachability and infection properties of the mutants will be checked. The outcomes of this step are as follows:

- An **invalid** mutant model is either incorrect syntactically, or does not satisfy the mutation-selection criteria (reachability and infection).
- A **valid** mutant model must be a syntactically correct model and satisfy the mutation selecting criteria (reachability and infection).

Valid mutants are used for generating faulty test cases. During test execution, they will be classified further as follows:

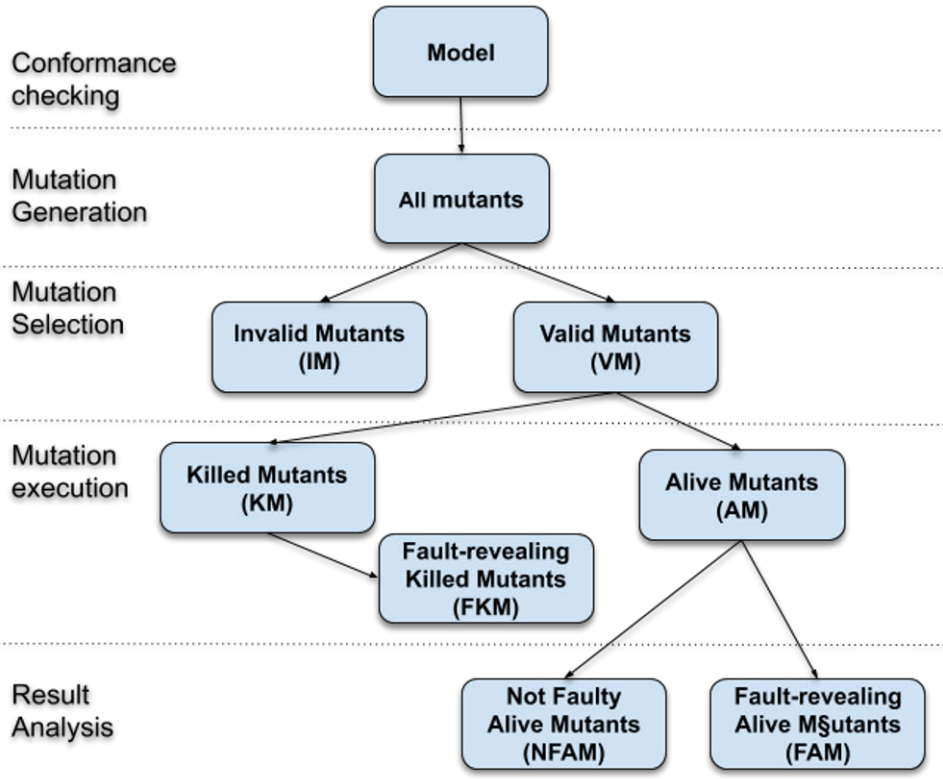


Figure 3.7: Classification of mutations in this thesis

- A **killed** mutant model is a mutant which does not conform to the behavior of the SUT resulting in a failed or inconclusive verdict during testing.
- An **alive** mutant is a valid mutant that generates faulty behavior that cannot be observed or distinguished from the behavior of the SUT.
- If a mutant is killed but also reveals an anomaly in the SUT, then we count it as a **fault-revealing killed** mutant.

Since infection criterion ensures that the mutants are not equivalent to the original model, we do not have any equivalent mutants. Analyzing alive mutants classifies them as follow:

- If the behavior of a mutants is the same as the behavior of the SUT, then there is a fault in the implementation of the SUT. Such mutant is called **fault-revealing alive** mutant.

- If a mutant's behavior does not harm the SUT and does not cause any change in the state of the SUT, then it is known as a **not faulty alive** mutant. Distinguishing between such mutant and fault-revealing alive mutants should be checked by manual inspection.

3.4 Defining Selection Criteria for Model Mutations to Improve our Testing Approach

Even though mutation testing has shown to be stronger than other testing criteria, it suffers from being expensive and cumbersome. Due to the vast number of mutants which mostly are not suitable for testing, the effort of generating mutants with a limited number of valuable mutants is high.

The unsuitable mutants either have identical behavior to their original model (known as *equivalent mutants*) or are not syntactically correct. The problem of equivalency of mutants is a known problem in code-based mutation testing, and there are many techniques to minimize the problem, as illustrated in [83]. Selecting suitable mutants which can be used for generating test cases is also another challenge that has been studied in various ways.

As the fourth contribution of this thesis, we introduce a useful technique that reduces the effort of mutation generation. Our mutation-selection technique originates from the main principle of mutation testing: reachability, infection, and propagation (RIP) for killing mutants.

To kill a mutant:

1. It must be **reachable**, which means that the mutated part of the model is accessible at some point of test execution,

2. It must cause **infection** on the state of the mutant model after the mutation is visited,

3. It can **propagate** the mutation through the model (i.e., the difference in the behavior of the mutant is observable).

If the reachability and infection are satisfied by a mutant, it is called *weak mutant*, and if all conditions are met, it is called *strong mutant*.

In our approach, during mutant selection, we check whether each mutation is reachable and potentially infects the SUT during test execution. These conditions help to eliminate unfit mutants. In UTA, reachability, liveness, and deadlock-free properties can be defined on states and paths. We define reachability criteria based on the model elements where the mutation is applied.

RIP Condition for Mutants in UTA

In our MBMT approach, during mutant selection, we check whether each mutation is reachable and potentially infects the SUT during the test execution. These criteria improve the computation and execution time of test generation by eliminating unfit mutants. In UTA, reachability, liveness, and deadlock-free properties can be defined on states and paths. We define reachability criteria based on the model elements where the mutation is applied.

Reachability

In the context of UTA, if a mutation applied on an edge, guard, or update, the **reachability** for *symbolic computation step* is defined using a global variable in the model and update its value on the action the mutation is applied to. Let $m \in \overline{v}$, be a boolean variable that is initialized to zero and action a is a mutated action. A mutated action is reachable iff in $l, m == 0$ and in $l', m == 1$, and $(l, \overline{v}) \xrightarrow{a} (l', \overline{v}')$ and
$D' = \{\overline{v}''|(l, \overline{v}) \xrightarrow{a} (l', \overline{v}') \wedge (l, \overline{v}') \xrightarrow{d} (l'', \overline{v}'') \wedge \overline{v} \in D \wedge m = 1\}$

Infection

After a mutation is visited and if input/output actions are different from the original input/output actions, it causes an **infection**. One way of detecting the infection is to apply *bisimulation* relation, which compares traces of the original model and its mutants and checks whether they are equivalent. We used this technique in [97]. The mutants that pass the reachability and infection conditions will be considered as valid mutants.

Propagation

Faulty test cases are generated from the valid mutants by online testing tool TRON. The **propagation** condition can be checked during the test execution by comparing the observable behavior of the mutants with the behavior of the SUT. Since UPPAAL TRON evaluates the test results based on *rtioco* relation, it is a proper tool for detecting the propagation of the mutations at runtime.

If the mutation causes a change in observable input/output or the delays, and it can be detected during the test generation, then it is considered as killed. Otherwise, if the SUT provides test outputs to the mutated test stimuli, then the mutated test case will not be detected and the mutant will be alive.

3.5 Developing a Supporting Tool for Model-Based Mutation Testing

The last contribution of this thesis is a supporting tool that serves as part of the testing toolchain. We presented the μUTA testing tool that extends online model-based testing approach offered by UPPAAL TRON and enables model-based mutation testing. Figure 3.8 shows an overview of μUTA that uses an UPPAAL model as the input and creates mutants that generate mutated inputs against the SUT.

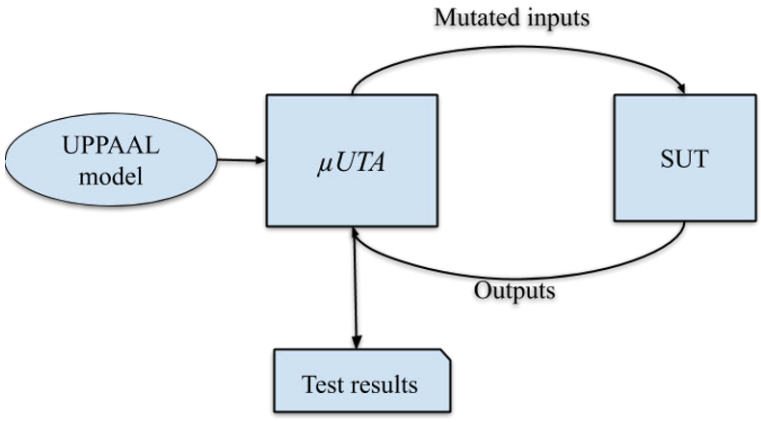


Figure 3.8: An overview of the μUTA tool

Figure 3.9 shows more detailed processes inside μUTA and the types of mutants that are generated at each stage. The tool consists of three processes: Mutation Generator, Valid Mutant Selector, and Mutation Test runner.

The Mutation Generator implements a set of mutation operators for UTA. The mutation operators are adopted from the studied in [3] and [1] and three new mutation operators that we defined in [100]. Each mutation operator generates several unique mutants.

From all generated mutants, we aim to select those that are suitable for test generation and create artificial invalid inputs. This task is done with Valid Mutant Selector, which defines verification properties for each mutation and verifies them. If the mutant satisfies the verification properties, then it is categorized as *valid* mutant, otherwise *invalid*. This process increases the efficiency of the MBMT approach by removing trivial and equivalent mutants and reduces the number of mutants that are used for the test generation. We evaluated the process by running an experiment on a case study with and without this process. The result shows a significant reduction in time of testing, as described in [100].

The invalid mutants are generally discarded, and from the valid mutants Mutation Test Runner executes tests against the SUT and categorizes the mutants into *killed* and *alive* based on the test results. Both killed, and alive mutants can reveal hidden faults in the SUT, however, detecting them still requires manual analysis (as it is shown in Figure 3.7). Killed mutants indicate that the SUT can detect their invalid inputs, while alive mutants indicate that the SUT could not detect any anomaly in the invalid inputs. It is worth noting that not all alive mutants indicate hidden faults in the SUT. In many cases, an alive mutant is a subset of the original model, or its mutation does not create any invalid inputs. Distinguishing fault-revealing mutants requires knowledge about the expected behavior of the SUT.

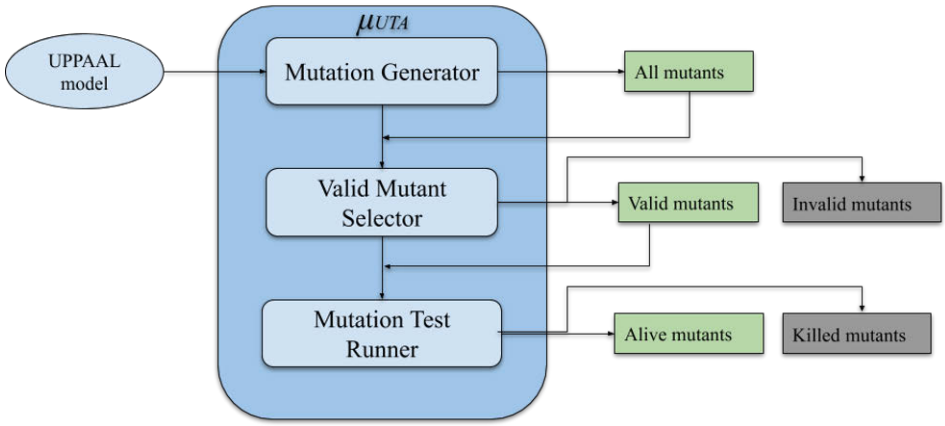


Figure 3.9: Processes within the μUTA tool and their outputs

Results of Applying MBMT Approach On Case Studies

From the UTA model of `Hotel Booking`, from a total of 1346 generated mutants, 393 were found to be valid mutants that were suitable for testing. After running the test, 40 mutants generated by two operators: CS and CT were found to identify three hidden faults that were not detected by the MBT conformance testing approach. The experiment indicates that the approach of specification mutation testing was adequate to reveal inconsistency between the specification and the SUT. The details of the test results and the faults are presented in Publication IV.

By running the MBMT approach for the `Blog` case study, in total, 2962 mutants were generated, whereas only 138 were valid mutants. From 138 valid mutants, 119 mutants were killed during the test execution, eight mutants caused a crash in the SUT during the test, and by investigating them, we found that all of the crashes belong to a single bug in the implementation.

Eleven mutants passed the test execution and remained as alive mutants. From eleven alive mutants, two CGV mutants were able to reveal two distinct vulnerabilities in the SUT. In total, three different defects have been found, one of them is revealed during the test execution, and two others have been found during the analysis. The details of the result and fault analysis are presented in Publication VI.

Chapter 4

Conclusions and Future Work

The primary focus of this thesis was related to model-based verification and testing interaction-intensive systems such as web services. Due to the increasing popularity of web services, they are expected to be highly accessible, reliable, and robust. As the first challenge, we presented an integrated model-based testing approach to design, verify, and validate composite web services.

We evaluated the integration and functionality of web service compositions with the model-based testing approach. With the help of requirements traceability mechanism, we traced requirements to UML models and, via the UML→UTA transformation to timed automata models. The reachability of the requirements was verified using Uppaal model-checker. The designed Uppaal models are used for online test generation. The use of online MBT proved beneficial as the system under test exhibits non-deterministic behavior due to concurrency and real-time constraints. Linking requirements to generated tests enabled tracing the requirements throughout the verification and validation processes.

Since web services play the main role in enabling communications among various systems, it is essential to confirm their correctness in different conditions. One of the main objectives of the research was to construct realistic test scenarios in which interactions of web services with their environment used as part of the test model. To be able to understand the role of environment in testing such systems, we systematically collected and reviewed a set of related work in the literature for getting guidelines of using environment modeling in model-based testing. By identifying the main characteristics of the environment models and application domains, we utilized environment models for examining web services based on user scenarios. The user scenarios were transformed from UML sequence diagrams into UTA (SD→UTA)

and used for test generation.

The next challenge we have addressed was the evaluation of the robustness of web services with model-based mutations. To this end, we combined the model-based testing technique with mutation testing concept to create mutant models. These models simulate unexpected and invalid input data to test the robustness of web services. The mutations are created from a set of well-defined mutation operators, each alters a specific element of the original test model and generates unique mutants.

Another challenge that we examined was the improvement in the efficiency of tests that much depends on the computation cost, such as the total time of mutation generation and test execution. We reduced the time of test execution by employing a verification process that discards equivalent, redundant (i.e., similar mutants) or trivial mutants and select valuable mutants during the mutation generation. The results show that applying the verification process reduced the total time of the test generation to less than half.

From a technical viewpoint, the trade-off between abstraction and the cost of computing is yet one the most challenging problem in designing software systems. On the one hand, eliminating the details of a system has the advantage of focusing on its general behavior. Adding the details will cause the state explosion problem. One solution to this challenge is to invest in designing only critical parts of a system and rigorously verifying them. In this thesis, we focused on the critical parts of our case studies. In the `Hotel Booking` case study, the goal of testing was the integration of the system. Therefore, our main concern about modeling, verification, and testing was on the integration of the system. Whereas in the `Blog` case study, our goal was to model multi-user access control over shared resources and finding security vulnerabilities in the web service. The mutation-selection process is integrated into the μUTA tool. The tool is built upon UPPAAL and TRON tools and contains three main parts: mutation generation, valid mutant selection, and mutation execution.

The contributions of this thesis have shown that the presented testing approach is promising, and the results have illuminated some research directions for further studies. The testing approach and mutation operators are not limited to web services but applicable to different types of real-time systems. In this thesis, we focused on web services as popular and widely used systems. The results of the experiments are mainly extracted from two case studies of web services. Generalizing the results for other application domains can be studied as future work. To this aim, the presented approach should be applied to different systems and in various application domains. Some improvements can reduce the test execution time while increasing the probability of finding faults. As the results of the experiment show, some of the mutation operators were more efficient than others in generating mu-

tants that can reveal faults. The result of mutation testing indicates that an intelligent choice of the mutation operators can attain high mutation efficiency scores while reducing the time of testing. From the results, it can be concluded that additional studies need to be conducted to improve the mutation operators.

Another direction on future work would be to combine the primary mutation operators and design higher-order mutations. Higher-order mutations can simulate more sophisticated behavior of the systems and are beneficial in identifying more complex defects. The presented approach does not adjust the mutant generation based on the knowledge of the previous test results. Therefore, another intriguing direction in research would be to use learning algorithms for test generation and test execution. The tester can start from a set of initial test cases and executes them against the SUT. Based on the test results from the initial tests, the tester can decide what type of mutants should be generated further.

More extensive studies are needed to investigate how model-based mutation can be applied in larger case studies preferably industrial-sized web services. Finally, to generalize what type of mutants correlate to what sort of faults, a broad range of case studies and different web services should be used as a benchmark. Thus, the current work can be extended for a comprehensive benchmark.

Bibliography

[1] M. Aboutrab et al. Specification mutation analysis for validating timed testing approaches based on timed automata. In *36th Annual IEEE Computer Software and Applications Conference, COMPSAC 2012, Izmir, Turkey, July 16-20, 2012*, pages 660–669, 2012.

[2] W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky. Validation, verification, and testing of computer software. *ACM Computing Surveys (CSUR)*, 14(2):159–192, 1982.

[3] B. Aichering et al. Time for Mutants – Model– Based Mutation Testing with Timed Automata. In *Tests and Proofs*, pages 20–38. Springer, 2013.

[4] R. Alur et al. Model-checking for real-time systems. In *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 414–425. IEEE, 1990.

[5] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[6] C. Armstrong. Modeling Web Services with UML. In *Talk given at the OMG Web Services Workshop*, pages 134–141, 2002.

[7] G. Behrmann, A. David, and K. G. Larsen. A tutorial on uppaal 4.0.

[8] F. Belli et al. Model-based mutation testing - approach and case studies. *Sci. Comput. Program.*, 120:25–48, 2016.

[9] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Advanced Course on Petri Nets*, pages 87–124. Springer, 2003.

[10] M. Bozkurt, M. Harman, Y. Hassoun, et al. Testing web services: A survey. *Department of Computer Science, King's College London, Tech. Rep. TR-10-01*, 2010.

[11] T. Bray et al. The javascript object notation (json) data interchange format. Technical report, 2014.

[12] T. A. Budd and A. S. Gopal. Program testing by specification mutation. *Computer languages*, 10(1):63–73, 1985.

[13] M.-E. Cambronero, G. Díaz, V. Valero, and E. Martínez. Validation and verification of web services choreographies by using timed automata. *J. Log. Algebr. Program.*, 80(1):25–49, 2011.

[14] J. Carlson, J. Håkansson, and P. Pettersson. Saveccm: An analysable component model for real-time systems. *Electronic Notes in Theoretical Computer Science*, 160:127 – 140, 2006. Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005).

[15] E. Cerami. *Web Services Essentials.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 2002.

[16] S. Chemaa, R. Elmansouri, and A. Chaoui. Web services modeling and composition approach using object-oriented petri nets. *International Journal of Computer Science Issues (IJCSI)*, 9(4):37, 2012.

[17] S. M. Cho and J. W. Lee. Lightweight specification-based testing of memory cards: A case study. *Electronic Notes in Theoretical Computer Science*, 111:73–91, 2005.

[18] L. Desmet, B. Jacobs, F. Piessens, and W. Joosen. Threat modelling for web services based web applications. In *Communications and multimedia security*, pages 131–144. Springer, 2005.

[19] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos. A survey on model-based testing approaches: a systematic review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, pages 31–36. ACM, 2007.

[20] G. Diaz, J.-J. Pardo, M.-E. Cambronero, V. Valero, and F. Cuartero. Verification of web services with timed automata. *Electronic Notes in Theoretical Computer Science*, 157(2):19 – 34, 2006. Proceedings of the International Workshop on Automated Specification and Verification of Web Sites (WWV 2005).

[21] E. W. Dijkstra. Notes on structured programming, 1970.

[22] J. S. Dong, Y. Liu, J. Sun, and X. Zhang. Verification of computation orchestration via timed automata. In *International Conference on Formal Engineering Methods*, pages 226–245. Springer, 2006.

[23] N. Dragoni and F. Massacci. Security-by-contract for web services. In *Proceedings of the 2007 ACM Workshop on Secure Web Services*, SWS '07, pages 90–98, New York, NY, USA, 2007. ACM.

[24] J. Edvardsson. A survey on automatic test data generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering*, pages 21–28, 1999.

[25] T. Erl. *Service-oriented architecture: concepts, technology, and design.* Pearson Education India, 2005.

[26] R. T. Fielding. *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Doctoral dissertation, 2000.

[27] G. Fischer. The importance of models in making complex systems comprehensible. In *Human Factors in Information Technology*, volume 2, pages 3–36. Elsevier, 1991.

[28] G. Fraser and F. Wotawa. Test-Case Generation and Coverage Analysis for Nondeterministic Systems Using Model-Checkers. In *International Conference on Software Engineering Advances*, pages 45–45, Aug 2007.

[29] G. Fraser, F. Wotawa, and P. E. Ammann. Testing with model checkers: a survey. *Software Testing, Verification and Reliability*, 19(3):215–261, 2009.

[30] C. Fu, B. G. Ryder, A. Milanova, and D. Wonnacott. Testing of java web services for robustness. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 23–34. ACM, 2004.

[31] X. Fu, T. Bultan, and J. Su. Analysis of interacting bpel web services. In *Proceedings of the 13th international conference on World Wide Web*, pages 621–630. ACM, 2004.

[32] Functional safety of electrical/electronic/programmable electronic safety-related systems. https://webstore.iec.ch/publication/5517, 2018. [https://webstore.iec.ch/publication/5517 Online; accessed 11-Feb-2020].

[33] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong. Specification and design of embedded systems. 1994.

[34] A. Gorbenko, A. Romanovsky, V. Kharchenko, and A. Mikhaylichenko. Experimenting with exception propagation mechanisms in service-oriented architecture. In *Proceedings of the 4th international workshop on Exception handling*, pages 1–7. ACM, 2008.

[35] M. Grinberg. *Flask Web Development: Developing Web Applications with Python*. " O'Reilly Media, Inc.", 2014.

[36] R. Gronmo, D. Skogan, I. Solheim, and J. Oldevik. Model-driven web services development. In *e-Technology, e-Commerce and e-Service, 2004. EEE'04. 2004 IEEE International Conference on*, pages 42–45. IEEE, 2004.

[37] R. Hamadi and B. Benatallah. A petri net-based model for web service composition. In *Proceedings of the 14th Australasian database conference-Volume 17*, pages 191–200. Australian Computer Society, Inc., 2003.

[38] E. R. Harold. *XML in a Nutshell*, volume 3.

[39] G. T. Heineman and W. T. Councill. Component-based software engineering. *Putting the Pieces Together, Addison-Westley*, 2001.

[40] M. Heisel, D. Hatebur, T. Santen, and D. Seifen. Using uml environment models for test case generation. In *Software Engineering (Workshops)*, pages 399–406, 2008.

[41] M. Heisel, D. Hatebur, T. Santen, and D. Seifert. Testing Against Requirements Using UML Environment Models. *Softwaretechnik-Trends*, 28(3), 2008.

[42] T. A. Henzinger. *The theory of hybrid automata*. Springer, 2000.

[43] A. Hessel et al. *Testing Real-Time Systems Using UPPAAL*, pages 77–117. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[44] A. Hessel, K. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou. Testing Real-Time systems using UPPAAL. In *Formal Methods and Testing*, pages 77–117. 2008.

[45] M. Iqbal, A. Arcuri, and L. Briand. Environment Modeling with UML/MARTE to Support Black-Box System Testing for Real-Time Embedded Systems: Methodology and Industrial Case Studies. In D. Petriu, N. Rouquette, and Ø. Haugen, editors, *Model Driven Engineering Languages and Systems*, LNCS, pages 286–300. Springer Berlin Heidelberg, 2010.

[46] M. Z. Iqbal, A. Arcuri, and L. Briand. Empirical Investigation of Search Algorithms for Environment Model-based Testing of Real-time Embedded Software. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA, pages 199–209, New York, NY, USA, 2012. ACM.

[47] I. ISO. Systems and software engineering–vocabulary. *ISO/IEC/IEEE 24765: 2010 (E), ed*, pages 1–418, 2010.

[48] ISOIEC. Information security management systems – guidelines for information security risk management, 2017. [https://www.enisa.europa.eu/topics/threat-risk-management/risk-management/current-risk/laws-regulation/rm-ra-standards/bs-7799-3 Online; accessed 11-Feb-2020].

[49] E. Jahier, S. Djoko-Djoko, C. Maiza, and E. Lafont. Environment-model based testing of control systems: Case studies. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 636–650. Springer, 2014.

[50] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, 2011.

[51] M. Johns. Code-injection vulnerabilities in web applications—exemplified at cross-site scripting. *It-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik*, 53(5):256–260, 2011.

[52] D. Kaye. *Loosely Coupled: The Missing Pieces of Web Services*. RDS Press, 2003.

[53] M. Kim, Y. Kim, and H. Kim. A comparative study of software model checkers as unit testing tools: An industrial case study. *Software Engineering, IEEE Transactions on*, 37(2):146–160, 2011.

[54] G. Kramler, E. Kapsammer, W. Retschitzegger, and G. Kappel. Towards using uml 2 for modelling web service collaboration protocols. In *Interoperability of Enterprise Software and Applications*, pages 227–238. Springer, 2006.

[55] I. Kruger. Service-oriented software and systems engineering-a vision for the automotive domain. In *Formal Methods and Models for Co-Design, 2005. MEMOCODE'05. Proceedings. Third ACM and IEEE International Conference on*, page 150. IEEE, 2005.

[56] B. Kumar, B. Czybik, and J. Jasperneite. Model based TTCN-3 testing of industrial automation systems; First results. In *Conference on Emerging Technologies Factory Automation*, pages 1–4, 2011.

[57] N. Laranjeiro, M. Vieira, and H. Madeira. Improving web services robustness. In *2009 IEEE International Conference on Web Services*, pages 397–404. IEEE, 2009.

[58] N. Laranjeiro, M. Vieira, and H. Madeira. A robustness testing approach for soap web services. *Journal of Internet Services and Applications*, 3(2):215–232, 2012.

[59] K. G. Larsen et al. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct. 1997.

[60] K. G. Larsen, F. Lorber, B. Nielsen, and U. M. Nyman. Mutation-based test-case generation with ecdar. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 319–328, March 2017.

[61] P. J. Leach, T. Berners-Lee, J. C. Mogul, L. Masinter, R. T. Fielding, and J. Gettys. Hypertext transfer protocol–http/1.1. 1999.

[62] S. Lee et al. Automatic mutation testing and simulation on owl-s specified web services. In *Simulation Symposium, 2008. ANSS 2008. 41st Annual*, pages 149–156. IEEE, 2008.

[63] S. C. Lee and J. Offutt. Generating test cases for XML-based Web component interactions using mutation analysis. In *Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on*, pages 200–209, Nov 2001.

[64] F. Leymann, D. Roller, and M. T. Schmidt. Web services and business process management. *IBM Systems Journal*, 41(2):198–211, 2002.

[65] J.-h. Li et al. Mutation analysis for testing finite state machines. In *Electronic Commerce and Security, 2009. ISECS'09. Second International Symposium on*, volume 1, pages 620–624. IEEE, 2009.

[66] B. Lindström, J. Offutt, D. Sundmark, S. F. Andler, and P. Pettersson. Using mutation to design tests for aspect-oriented models. *Information and Software Technology*, 81:112 – 130, 2017.

[67] F. Liu, A. Narayanan, and Q. Bai. Real-time systems. 2000.

[68] N. Lohmann. A feature-complete petri net semantics for ws-bpel 2.0. In *International Workshop on Web Services and Formal Methods*, pages 77–91. Springer, 2007.

[69] N. Looker, M. Munro, and J. Xu. Simulating errors in web services.

[70] N. A. Lynch. *Distributed algorithms*. Elsevier, 1996.

[71] E. Marcos, V. de Castro, and B. Vela. Representing web services with uml: A case study. In *International Conference on Service-Oriented Computing*, pages 17–27. Springer, 2003.

[72] R. Marinescu, C. Seceleanu, H. Le Guen, and P. Pettersson. A research overview of tool-supported model-based testing of requirements-based designs. In *Advances in Computers*, volume 98, pages 89–140. Elsevier, 2015.

[73] W. Mayer and M. Stumptner. Model-based debugging – state of the art and future challenges. *Electronic Notes in Theoretical Computer Science*, 174(4):61 – 82, 2007. Proceedings of the Workshop on Verification and Debugging (V&D 2006).

[74] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[75] Mega@MaRt^2. MegaModelling at Runtime - scalable model-based framework for continuous development and runtime validation of complex systems, 2017. https://megamart2-ecsel.eu/ Online; accessed 11-Feb-2020.

[76] D. Miljković. Fault detection methods: A literature survey. pages 750–755, 05 2011.

[77] S. Murugesan, Y. Deshpande, S. Hansen, and A. Ginige. *Web Engineering: a New Discipline for Development of Web-Based Systems*, pages 3–13. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[78] S. Myagmar, A. J. Lee, and W. Yurcik. Threat modeling as a basis for security requirements. In *Symposium on requirements engineering for information security (SREIS)*, volume 2005, pages 1–8. Citeseer, 2005.

[79] S. Narayanan and S. A. McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of the 11th international conference on World Wide Web*, pages 77–88. ACM, 2002.

[80] E. Newcomer and G. Lomow. *Understanding SOA with Web services*. Addison-Wesley, 2005.

[81] R. Nilsson, J. Offutt, and J. Mellin. Test case generation for mutation-based testing of timeliness. *Electronic Notes in Theoretical Computer*

Science, 164(4):97 – 114, 2006. Proceedings of the Second Workshop on Model Based Testing (MBT 2006).

[82] PAM. Practical Applications of Model based Technologies to continuous integration and Testing method, 2013. https://research.it.abo.fi/projects/PAM Online; accessed 11-Feb-2020.

[83] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman. Mutation testing advances: an analysis and survey. *Advances in Computers*, 2017.

[84] R. Patton. *Software testing*. Pearson Education India, 2006.

[85] M. Prasanna, S. Sivanandam, R. Venkatesan, and R. Sundarrajan. A survey on automatic test case generation. 15(2), 2005.

[86] I. Rauf, F. Siavashi, D. Truscan, and I. Porres. An integrated approach for designing and validating REST web service compositions. In *WEBIST 2014 - Proceedings of the 10th International Conference on Web Information Systems and Technologies*, volume 1, pages 104–115, Barcelona, Spain, 2014. SCITEPRESS Digital Library.

[87] I. Rauf, F. Siavashi, D. Truscan, and I. Porres. Scenario-based design and validation of REST web service compositions. In *Web Information Systems and Technologies – Revised Selected Papers*, pages 145–160. Springer International Publishing, 2015.

[88] A. P. Ravn, J. Srba, and S. Vighio. Modelling and verification of web services business activity protocol. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 357–371. Springer, 2011.

[89] L. Richardson and S. Ruby. *RESTful web services*. " O'Reilly Media, Inc.", 2008.

[90] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education, 2004.

[91] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education, 2004.

[92] H. M. Rusli and S. Ibrahim. Testing web services composition: a mapping study. *Communications of the IBIMA*, 2011.

[93] M. Salas and E. Martins. Security testing methodology for vulnerabilities detection of xss in web services and ws-security. *Electronic Notes in Theoretical Computer Science*, 302(Supplement C):133 – 154, 2014. Proceedings of the XXXIX Latin American Computing Conference (CLEI 2013).

[94] M. Salas and E. Martins. Security testing methodology for vulnerabilities detection of xss in web services and ws-security. *Electronic Notes in Theoretical Computer Science*, 302:133–154, 2014.

[95] H. Schlingloff, A. Martens, and K. Schmidt. Modeling and model checking web services. *Electronic Notes in Theoretical Computer Science*, 126:3–26, 2005.

[96] K. Schmidt and C. Stahl. A Petri net semantic for BPEL4WS-validation and application. In *Proceedings of 11th Workshop on Algorithms and Tools for Petri Nets*. Citeseer, 2004.

[97] F. Siavashi, J. Iqbal, D. Truscan, and J. Vain. Testing web services with model-based mutation. In *Software Technologies – Revised Selected Papers*, pages 45–67. Springer International Publishing, 2017.

[98] F. Siavashi and D. Truscan. Environment Modeling in Model-based Testing: Concepts, Prospects and Research Challenges: A Systematic Literature Review. In *EASE 2015 - Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–6, Nanjing, China, 2015. ACM.

[99] F. Siavashi, D. Truscan, I. Rauf, and J. Vain. On Mutating UPPAAL Timed Automata to Assess Robustness of Web Services. In *ICSOFT 2016 – Proceedings of the 11th International Joint Conference on Software Technologies*, volume 1, pages 15–26, Lisbon, Portugal, 2016. SCITEPRESS - Science and Technology Publications.

[100] F. Siavashi, D. Truscan, and J. Vain. Vulnerability assessment of web services with model-based mutation testing. In *QRS 2018 – IEEE 18th International Conference on Software Quality, Reliability, and Security*, pages 301–312, Lisbon, Portugal, 2018. IEEE Computer Society Conference Publishing Services.

[101] D. Skogan, R. Gronmo, and I. Solheim. Web service composition in uml. In *Proceedings of the Enterprise Distributed Object Computing Conference, Eighth IEEE International*, EDOC '04, pages 47–57, Washington, DC, USA, 2004. IEEE Computer Society.

[102] J. Snell, D. Tidwell, and P. Kulchenko. *Programming web services with SOAP: building distributed applications.* " O'Reilly Media, Inc.", 2001.

[103] E. A. Strunk, M. A. Aiello, and J. C. Knight. A survey of tools for model checking and model-based development. 2006.

[104] F. Swiderski and W. Snyder. *Threat Modeling (Microsoft Professional)*, volume 7. Microsoft Press, 2004.

[105] The Open Web Application Security Project (OWASP). https://www.owasp.org/, 2018. [Online; accessed 11-Feb-2020].

[106] The World Wide Web Consortium (W3C). Web Services Architecture, 2011. https://www.w3.org/TR/ws-arch/ Online; accessed 23-Feb-2020.

[107] J. P. Thomas, M. Thomas, and G. Ghinea. Modeling of web services flow. In *EEE International Conference on E-Commerce, 2003. CEC 2003.*, pages 391–398, June 2003.

[108] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[109] R. Vidgen, D. Francis, P. Powell, and M. Woerndl. Web service business transformation: collaborative commerce opportunities in smes. *Journal of Enterprise Information Management*, 17(5):372–381, 2004.

[110] M. Vieira, N. Laranjeiro, and H. Madeira. Assessing robustness of web-services infrastructures. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 131–136, June 2007.

[111] W. Visser, M. B. Dwyer, and M. Whalen. The hidden models of model checking. *Software & Systems Modeling*, 11(4):541–555, 2012.

[112] J. M. Voas and G. McGraw. *Software fault injection: inoculating programs against errors.* John Wiley & Sons, Inc., 1997.

[113] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, Aug 2016.

[114] X. Yi and K. J. Kochut. A cp-nets-based design and verification framework for web services composition. In *Proceedings. IEEE International Conference on Web Services, 2004.*, pages 756–760, July 2004.

[115] W. Yichen and W. Yikun. Model-based simulation testing for embedded software. In *2011 Third International Conference on Communications and Mobile Computing*, pages 103–109, 2011.

[116] D. Zubrow. Ieee standard classification for software anomalies. *IEEE Computer Society*, 2009.

Part II

Original Publications

Publication I

An Integrated Approach for Designing and Validating REST Web Service Compositions

Irum Rauf and Faezeh Siavashi and Dragos Truscan and Ivan Porres (2014). In *WEBIST 2014 - Proceedings of the 10th International Conference on Web Information Systems and Technologies*, pages 104–115. SCITEPRESS Digital Library

An Integrated Approach for Designing and Validating REST Web Service Compositions

Irum Rauf, Faezeh Siavashi, Dragos Truscan and Ivan Porres
Åbo Akademi University, Dept. of Information Technologies, Turku, Finland

Keywords: REST, Web Service Composition, Model-based Testing, UPPAAL, TRON.

Abstract: We present an integrated approach to design and validate RESTful composite web services. We use the Unified Modeling Language (UML) to specify the requirements, behavior and published resources of each web service. In our approach, a service can invoke other services and exhibit complex and timed behavior while still complying with the REST architectural style. We show how to transform service specifications into UPPAAL timed automata for verification and test generation. The service requirements are propagated to the UPPAAL timed automata during the transformation. Their reachability is verified in UPPAAL and they are used for computing coverage level during test generation. We validate our approach with a case study of a holiday booking web service.

1 INTRODUCTION

Web services have machine-readable interfaces that automate the task of communicating information between machines and reduce time and human efforts. They are being increasingly used in the industry in order to automate different tasks and offer services to a bigger audience. REST architectural style aims at producing scalable and extensible web services using technologies that play well with the existing tools and infrastructure of the internet. This has encouraged notable enterprizes to use REST web services to meet their needs. REST interfaces offer a CRUD interface (create, retrieve, update and delete) to its users via a set of standard HTTP methods. They offer stateless behavior that facilitates scalability and requires that no hidden session or state information be carried between method calls.

Different web services published over the internet can be composed to form a composite web service such that the composed web service fulfills new service goals using the functionality of partner web services. Automated systems, for example hotel reservation systems, are often built as stateful composite services that require a certain sequence of method invocations that must be followed in order to fulfill service goals. Designing and developing such stateful composite services with REST features is not a trivial task and requires rigorous approaches that are capable of creating reliable web services.

Thus, with the use of web services in businesses and critical applications, there is an increasing need for a) design approaches to develop web service compositions that support complex scenarios and timed behavior while complying with the REST architectural style and b) validation approaches to effectively and efficiently detect faults in specifications and implementations of such services.

In this article, we present a design and validation approach that facilitates the service developer to create reliable, timed and stateful composite REST web services. As a first contribution, we introduce an approach in which we use the Unified Modeling Language (UML) (UML, 2009) to model our service specifications via an extension of our previous work in (Porres and Rauf, 2011). We use UML since it has emerged as a standard modeling language at industrial level (Budgen et al., 2011) and has sophisticated tools due to large user base. This can make adoption of the approach easier in the industry.

The service design models and their implementations should be validated for their correct behavior in order to build trust on the service functionality. We have used the model checking approach for this purpose. Model checking is a way to exhaustively and automatically check if a finite-state model of a program satisfies its specifications (Clarke et al., 1994). The UML-based service design models represent the system graphically and are comprehensible for a human user. In order to make the models amenable

Rauf I., Siavashi F., Truscan D. and Porres I..
An Integrated Approach for Designing and Validating REST Web Service Compositions.
DOI: 10.5220/0004949601040115
In *Proceedings of the 10th International Conference on Web Information Systems and Technologies* (WEBIST-2014), pages 104-115
ISBN: 978-989-758-023-9

for model checking we suggest, as a second contribution, a set of reversible mechanized steps for translating UML-based service specifications into UPPAAL timed automata (UPTA) (Larsen et al., 2009). UPPAAL is a commonly used model checking tool for verifying real time systems through modeling and simulation (Larsen et al., 1997). We verify basic properties of our design models such as reachability, liveness and safety using the UPPAAL model-checker tool. UPTA are updated (if needed) based on the verification results and transformed back to UML. From the UML models, a skeleton of the composite service is generated automatically in the Django web development framework (Holovaty and Kaplan-Moss, 2009) using our partial code generation tool.

In our approach, a service can invoke other services and exhibit complex and timed behavior while still complying with the REST architectural style. We need to check if the service implementation is functioning correctly along with partner services and if the service goals and timed constraints are being fulfilled. Thus, as a third contribution, we show how we validate the implementation of the RESTful web service composition with a model-based testing (MBT) approach using the UPPAAL TRON tool (Larsen et al., 2009). By using MBT, test cases can be automatically generated with an increased probability of test coverage and with an ease of test case maintenance.

Requirements traceability is an important component of our integrated approach. The requirements of the composition are included in the UML specifications and then propagated to UPTA specifications. They are used for both verifying the reachability of those model elements implementing them and for reasoning about their coverage after the tests are executed. Upon detecting failures, traced requirements can be used to localize the errors either in the models or in the specifications.

We exemplify and validate our integrated approach with a relatively complex case study of a holiday booking composite REST web service from industrial context. The case study shows how stateful and timed web services offering complex scenarios and involving other web services can be constructed using our approach.

The paper is organized as follows: Section II gives an overview of our approach, while tool support is discussed in section III. The case study is presented in section IV and the evaluation of the approach is presented in section V. The related work is discussed in section VI and the section VII concludes the paper.

2 OUR APPROACH

An overview of our integrated design and validation approach is given in Figure 1. The left side of the figure shows our previous work, whereas the right side shows the contribution of this paper and how it is connected to the previous work.

In our previous work (Figure 1–left), we designed behavioral interfaces for web services that were RESTful by construction (Porres and Rauf, 2011). These design models were implemented in the Django web framework using our partial code generation tool (Rauf and Porres, 2011) which generated code skeletons with pre- and post-conditions for every service method. The design models were also analyzed for their consistency (Rauf et al., 2012).

In the current work, we focus on designing, verifying and testing composite REST web service (Figure 1–right) as follows:

Design: First of all we extend our design approach to create composite REST web services with UML. Our approach takes as input the behavioral interface specifications of the partner REST web services and the business requirements. With these inputs, we construct models for composite web service using our design approach.

Verification: We then provide verification of the design models by reasoning on the basic properties of models like deadlock, liveness, reachability and safety with the UPPAAL model-checker. To achieve this, we transform the service design models to UPTA, which are simulated and verified. Based on verification results, the UML-based service design models are updated.

Transformation: Transformation step generates two types of automata from service design models. One of the types corresponds to service design models and the other type represents the environment model. The environment model simulates the behavior of service user to invoke the interface service methods in order to facilitate test generation.

Code Generation: The code skeletons are generated from UML service design models to Python-based Django web framework using our code generation tool (Rauf and Porres, 2011) that are completed manually by the developer.

Testing: For model-based testing of the service implementation, we have used online conformance testing tool UPPAL-TRON that validates the service implementation against its UPTA specification models at runtime.

Evaluation: In the end, we have evaluated our validation approach for its efficiency using mutation testing and benchmarked.

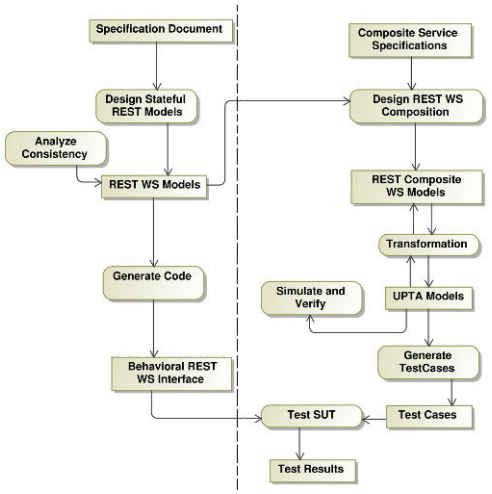


Figure 1: Activity Diagram of Design and Validation Approach for REST CWS.

2.1 REST Composition Models

We require that a composite REST web service interface should exhibit the REST interface features, i.e. addressability, connectivity, statelessness and uniform interface. We have modeled our composite REST web service interfaces with a *resource model*, a *behavioral model* and a *domain model* that exhibit these features.

This work extends our previous work on creating behavioral interface specifications of individual REST web services (Porres and Rauf, 2011). For the details of the design approach, readers are referred to (Porres and Rauf, 2011).

The concept of resource is central to the structure of REST web service. It represents a piece of information (Richardson and Ruby, 2008). We respresent the static structure of REST web service with resource model which is modeled with a UML class diagram. Each class represents a *resource definition*. We have used the term *resource definition* to define a resource entity such that its instances are called resources. This is analogue to the relationship between a *class* and its *objects* in the object-oriented paradigm.

The direction of the association between *resource definitions* gives the navigability direction between them while their role names give the relative URI of resources (addressability). The *collection resource definiton* without the incoming transitions is termed as *root* such that every *resource definition* in the resource model should be reachable via *root* and the graph formed should be connected (connectivity).

A behavioral model represents the dynamic structure of the service and it is modeled by a UML State Machine (SM). Each state represents the service state and the trigger methods of transitions are restricted to the side-effect methods of HTTP, i.e., PUT, POST and DELETE (uniform interface). The statelessness feature of the REST interface is preserved while building stateful REST web service by defining state invariants as boolean expression of states of different resources. The state of a resource is given by its representation retrieved by invoking a GET on it. We are thus able to define service states as predicates over the resources without maintaining any hidden session or state information (statelessness). The state invariants in the SM are written as Object Constrain Language (OCL) expressions. OCL is commonly used to define constraints in UML models, including state invariants (Birgit Demuth, 2009).

For modeling a service composition, the models are required to represent method invocations on the partner services. The service invocations to partner services are modeled as effects on the transitions. The composite web service requirements, inferred from the specification document, are added as UML *comments* on the transitions that satisfy them. These requirements should be met by the implementation of the service in order to fulfill the service goals.

The *domain model* of the composite service is represented with a class diagram. It represents interfaces between the composite service and its partner services. The required and provided interface methods between the composite and its partner services are modeled with required and provided interfaces in the domain model, respectively.

2.2 Verification

Model verification is a process of determining whether the models are designed correctly and represent the developer's conceptual descriptions and specifications. Model checking is one of the ways to exhaustively check the models automatically. The service design models of composite REST web service should be verified for their basic properties in order to build confidence of the service designer on the models before implementing them. This allows one to eliminate design errors that can be expensive to detect and correct at later stage of the development cycle.

UPPAAL model-checker is used for modeling, simulation and verification of real-time systems (Larsen et al., 1997). It consists of set of timed automata (TA), clocks, channels that synchronize the systems (automata), variables and additional elements. A real-time system is modeled as a closed network of TA. Each automaton in the network is speci-

fied via a *template*, which can be instantiated as *process*. A template in UPTA is composed of *locations*, *edges*, *clocks* and *variables* representing all properties of the system. Synchronization between different processes can be provided using *channels*. Two edges in different automata can synchronize if one is emitting (denoted as *channel_name!*) and the other is receiving (denoted as *channel_name?*) on the same channel. *Guards* are the conditions that enable a transition when they are satisfied. Similarly, the conditions associated to locations, called *invariant*, specify that the system can stay in the location if and only if the invariant is satisfiable.

The query language used in UPPAAL is a simplified version of TCTL (Alur et al., 1990) that consists of state formulae and path formulae. State formulae (φ) is an expression that describes an individual state, while path formulae can be classified into reachability, safety and liveness properties. Deadlock is expressed using state formulae. The syntax of TCTL path formulae that are used in UPPAAL is defined as follows:

- A $\Box$ φ - for all paths, the property φ is always satisfiable.
- A $\Diamond$ φ- for all paths, the property φ is eventually satisfiable.
- E $\Box$ φ - there is at least a path in the automata such that property φ is always satisfiable.
- E$\Diamond$ φ - there is at least a path in the automata such that property φ is eventually satisfiable.
- $\varphi \rightsquigarrow \phi$ - when φ holds, ϕ must hold.

If there is a location in the model that has no outgoing transition, then the model is said to be in a deadlock. Reachability properties validate the basic behavior of the model by checking whether a certain property is possible in the model with the given paths. The safety property checks that something bad will never happen and the liveness property is verified to determine that something will eventually happen.

However, before using UPPAAL model-checker to verify these properties we need to give our service design models in UML formal foundations that are understandable by the verification tool. This has to be done in an automated manner to avoid extra efforts from the service developer. In section 3, we present our tool support for this and explain in detail the transformation from UML to UPTA.

2.3 Model-Based Test Generation

Model-based testing (MBT) is a method that provides an abstract model of a system under test (SUT) and preforms automatic test case generation based on the specifications of the SUT (Utting and Legeard, 2007). In MBT, modeling the environment of a system is important since the environment generates test cases from whole or some parts of the model to satisfy the test criteria. Environment models help in automation of testing in three ways: the automation of test case generation from a simulated environment, the selection of test cases, and the evaluation of their test results. Our UML to UPTA transformation tool generates both the SM of SUT and the environment model.

We provide automatic test generation using UPPAAL TRON, which is an extension of UPPAAL for online model-based black-box conformance testing (Larsen et al., 2009). During test generation, the environment model randomly selects test cases and communicates to the test adapter.

A test adapter is used by UPPAAL TRON to expose the observable I/O communication between the test environment model and the SUT model, as shown in Figure 2. Our adapter implements the communication with the SUT by converting abstract test inputs into HTTP request messages and HTTP response messages into abstract test outputs. The TRON tool generates tests via symbolic execution of the specifications using randomized choice of inputs. Based on the timed sequence of input actions from the simulation, the adapter preforms input actions to Implementation Under Test (IUT) and waits for the response. Output from IUT is monitored and generated as output actions for the simulation. The conformance testing is achieved by comparing outputs of IUT to the behavior of the simulation.

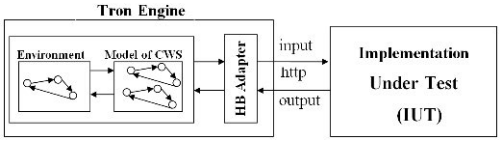


Figure 2: UPPAAL TRON test setup.

2.4 Requirements Traceability

Service requirements can be inferred from the specification document and they serve as service goals. A service should be checked for its service goals in order to validate that the service does what it is required to do. By catering to the service requirements at the design phase and propagating them to the validation stage, we provide a mechanism by which a service requirement can be validated for its goals and the unfulfilled requirements can be traced back to the design phase to find faults in the design. Service requirements are generally domain-specific since they are inferred from the specifications. We infer functional and temporal requirements from the specification document into a table and number them. These

requirements are attached to the SM as *comments* on the transitions and are propagated to UPTA such that the links between requirements and the model elements are preserved. These requirements are included in all the models and traced throughout the process, i.e., at UML, UPTA and test level, respectively.

The requirements are formulated as reachability properties in UPTA with the purpose of verifying them during simulation. Each requirement label is translated into a boolean variable (initialized to False) and attached to the corresponding edge in the UPTA. This mapping is explained in more detail in the Section 3 on the UML to UPTA transformation.

We require that our testing approach must validate that these requirements are met by IUT, in order to build confidence of the developer that the system is doing what it is required to do. Thus, their coverage level is monitored during test generation and execution. Once the test report is available, we can check which requirements have been validated and which have failed.

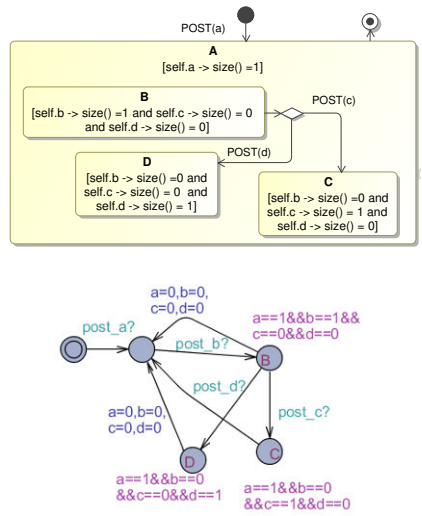


Figure 3: (Left) Composite State in UML State Machine. (Right) Flattened locations in timed automaton.

3 TOOL SUPPORT

Modeling in UML. The design models are modeled using MagicDraw (Mag, 2013). Static validation of models is done via OCL using the validation engine of Magic Draw. We rely on predefined validation suites for UML contained in MagicDraw for the basic validation of the model. These validation suites contain rules that check that the designed UML model conforms to UML meta-model specifications and prevent the developer from doing basic modeling mistakes.

Code Generation. The code-skeleton of the updated service design models of REST composite web service can be generated using our tool presented in (Rauf and Porres, 2011). The tool generates code skeleton for design models in Django that is a high level Python web framework (Holovaty and Kaplan-Moss, 2009). The generated code also has behavioral information such that the pre and post conditions for each method are included and the developer just has to write the implementation of the operations.

UML→UPTA Transformation. The transformation from UML design models to UPTA is an extension of our approach presented in (Nobakht and Truscan, 2013). The extension of transformation generates several artifacts: UPTA model, test environment model and a skeleton for test adapter depicting the testable interfaces of the composite web service.

Resource Model. In UPTA the resource model is represented as a template. The *resource definitions* in the resource model are specified as variables with 1 or 0 value, specifying if a resource exists or not, respectively. The attributes of *resource definitions* are inspected and for each integer attribute, an integer variable is declared in the UPPAAL model. Similarly, the boolean attributes are declared as integer arrays of 0 and 1.

Domain Model. The domain model shows set of operations offered and required by the composite web service and its partner web services. The corresponding communication between templates in UPPAAL is represented by channel synchronizations. Two edges in different automata in UPPAAL can synchronize if one is emitting and the other is receiving on the same channel. The operations in an interface are thus translated into a binary synchronization channel in UPPAAL. The template of the service that realizes the interfaces acts as the receiving automaton and the sending automaton is specified by the template of the service that uses the interface.

Behavioral Model. The SM of the REST web service is encoded to TA that are represented by templates, which are instantiated as processes. Figure 3 shows an example of transformation from the SM to TA.

States. A state is mapped to a location in UPTA, and a state invariant is mapped to corresponding location invariant. The subclauses of the state invariant are translated to variables corresponding to the respective resource definition. For example, in Figure 3, $self.a- > size() = 1$ is translated as $a = 1$ and $self.b- > size() = 0$ as $b = 0$. The initial state corresponds to the initial location. The final states are translated by having an edge from the corresponding

location to initial location and updating all the variables to their initial values, as shown in Figure 3. The choice state in the SM is replaced by two edges in the TA model that are originating from the same source location to different target locations.

State Hierarchy. The SM may contain composite states for better representation of specifications. UPTA, however, does not support the notion of location hierarchy. We flatten the composite states into several simple states by including the state invariant of super states in the contained states that are then mapped to the respective locations in UPTA. For example, in Figure 3, the top figure contains a SM with a composite state that is flattened to UPTA model shown at the bottom. States B, C and D in the SM correspond to the locations B, C and D of UPTA, respectively. Note that all the locations contain the state invariant of superstate A in the SM.

Transitions. A transition in the SM is mapped to an edge in UPTA and guards on the transition are mapped to guards on the corresponding edge in UPTA. In Figure 4, we shows how the transitions in the SM (top) are translated to UPTA (bottom right). The locations $L1$ and $L5$ correspond to states $S1$ and $S2$ of SM, respectively, and locations $L0$, $L2$, $L3$, and $L4$ are the extra locations created during the transformation process as explained below. The state invariants are translated to location invariants and represented as boolean functions for the purpose of diagram clarity. The transition between states S1 and S2 is triggered by POST(b) after 10 minutes as specified in the guard. In UPTA, this is represented as guard over the clock variable *cl*.

Trigger Methods. The trigger methods from the SM are translated in to receiving channels in UPTA. This receiving channel is in sync either with the automaton of the partner service or with the environment model.

Time Events. The time events in behavioral diagram are replaced by clocks in UPTA. The clock is reset in the incoming edge to the location (L1) and is also included in the location invariant. Thus, the guard *after(10m)* is translated to $cl > 10$ on the corresponding UPTA edge.

Effects. The effect on the transition, i.e., $POST(c)$ shows invocation to the partner service. The communication between two web services is established by using a unique channel synchronization. For instance, emitting a request from a web service to the other one can be replaced by synchronizing a channel in an UPPAAL process, and the other process is the receiver of the synchronization. The effect of the transition that invokes a remote service is represented with two edges and an urgent location (marked with U in the circle) in between, i.e., edges *e2* and *e3* and *urgent* location *L3*. An urgent location in UPTA does not allow any delays (Larsen et al., 1997). Thus, the first edge (e1) is synchronized with the environment model and the second edge (e2) synchronizes with the partner automaton. The third edge (e3) is synchronized to receive acknowledgment response from the partner (as we model asynchronous service) and the sending channel on the fourth edge (e4) is synchronized with the environment to indicate the completion of transition.

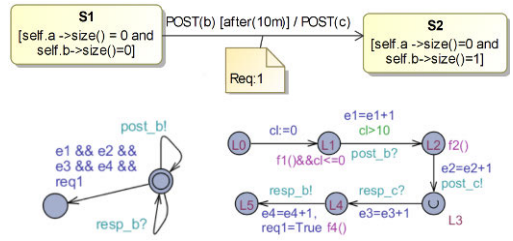


Figure 4: Example of SM (top) Corresponding Environment Model (bottom left) and Flattened TA (bottom right).

Requirements. The requirement on the transitions are translated into a boolean variable (initialized to *False*) and attached to the corresponding edge which updates it to *True*. This is shown in Figure 4 with *Req1= True* on edge e4. This implies that whenever this edge would be traversed, this requirement will be met. This can be formulated as reachability properties to attain requirement coverage and tracked during test generation and execution.

Environment Model. The environment model in UPTA has sending channels that are received by the composite web service automaton as inputs to trigger the process. This is similar to interface method calls in the SM. All the interface methods of the service specified in the state machine are mapped to the sending channels in the environment model and the response of successful transition is received from the composite web service via receiving channels. This is also shown in Figure 4: the environment model initiates the automaton (bottom right) by sending channel *post_b*! and the process completes when the channel *resp_b*? is received.

A Python script is currently used to create the environment model, from a given UPTA model by analyzing the channels observable from the environment. The original idea has been discussed in (Hessel et al., 2008). This will be merged in the final version of the UML→UPTA transformation script.

Test Coverage Information. In order to enable rigorous test coverage in UPPAAL TRON, a second Python script (discussed in more detail in (Koskinen et al., 2013)) is used to automatically add *counter*

variables for each edge of a given automaton in a UPTA model and a corresponding update of the given variable on the corresponding edge. Whenever the edge is visited during the simulation or execution, the variable is incremented, allowing thus to track which edges have been visited and how many times. This enables one to track coverage level wrt. e.g., edge coverage or edge pair coverage. This script will also be integrated in the final version of the UML→UPTA transformation script.

4 CASE STUDY

Our case study is a Holiday Booking (HB) composite REST web service that is built on inspiration from the *housetrip.com* service, with the purpose of having a case study similar in complexity to real services. This service is a holiday rental online booking site, where one can search and book an apartment in the destination country.

The user of the service searches for a room in a hotel from the list of available hotels at HB before travel. He books the room (if it is available) and that booking is reserved by HB with the hotel for 24 hours. The user must pay for the booking within 24 hours. If the user does not pay within this time then the booking is canceled. If the booking is paid, then the HB service invokes a credit card verification service and waits for the payment confirmation. When the payment is confirmed, HB invokes the hotel service to confirm the booking of the room. If the hotel does not respond within 1 day or it does not confirm at all, the booking is canceled and the user is refunded. If the hotel service confirms, then a booking is made with the hotel. The payment is not released to the hotel until the user checks in. When the user checks in, HB releases the money to the hotel and the booking is marked by the hotel as paid. Due to space limitation, we only show some of the models and information here. The detailed case study is available at (Rauf et al., 2013)

Design Models. The design of HB composite REST web service is modeled with resource, behavioral and domain models. The state machine of HB composite service is shown in Figure 5.

Requirements Traceability. We have inferred functional and temporal requirements from specification document for our case study. Table 1 shows the requirements for *Booking* and *Payment Release*. These requirements should be fulfilled by the IUT in order to satisfy the service goals. They are added as comments to the model in Figure 5.

Verification. The design models of Holiday Booking (HB) composite REST web service are translated to UPTA with the help of transformation tool. Here, we only show an excerpt of UPTA in Figure 6. The detailed model and the specifications of the partner web services are available in (Rauf et al., 2013).

Table 1: Requirements of Holiday Booking CWS (excerpt).

Req	Sub-Requirements
1- Booking	1.1 - A booking should be paid 1.1.1 - A booking should be paid within 24 hours of the booking. 1.1.2 - If a booking is not paid within 24 hours of the booking, then it is canceled by the system 1.1.3 - A confirmed paid booking, waits for user check in
2- ...	2.1 - ...
4- Payment Release	4.1 - If the user checks in then the payment must be released to the hotel. 4.2 - When the payment is released to the hotel, HB CWS must notify the hotel about release of the payment

The verification properties are specialized for our case study and some of them are mentioned below.

Deadlock Freeness. The HB Service, the hotel service and the payment service models are all deadlock free. This means that the composite service is never reach to a state that cannot preform a transition (i.e., $A[]\ not\ deadlock$). Note that the following queries are made for complete model and only some of them can be traced in Figure 6.

Reachability. All the locations in the HB service are reachable. This means that the model receives and sends messages to the partner services smoothly and the model is validated for its basic behavior (i.e., $E \Diamond CompService.r$), where r is the last location in the TA model and indicates that all processes for certain booking is completed.

Safety. Some of the safety properties in our model are: a) Payment should be released iff the user has checked in, i.e., ($E \Box CompService.h2$ imply $CardService.c2$), where $c2$ is the location after check-in and $h2$ is the location after payment release, b) If the payment is released by the HB service then the Hotel service is paid, i.e., ($E \Box CompService.h2$ imply $HotelService.p$), where p is the location in Hotel service model for hotel payment.

Liveness. Some of the liveness properties in the model are: a) When the payment is not paid within 24 hours, the booking is canceled (i.e., $CompService.c$ and $compService.cl > 24 \rightsquigarrow CompService.b1$), where c indicates waiting for the payment, cl indicates clock of the model and $b1$ indicates the booking request is going to cancel due to the delay, b) If the Hotel Service does not confirm within 3 days then the booking is considered not confirmed (i.e., $CompService.o$ and $CompService.cl > 3 \rightsquigarrow CompService.n$), where o is the location for waiting for the hotel response and n is the location for canceling.

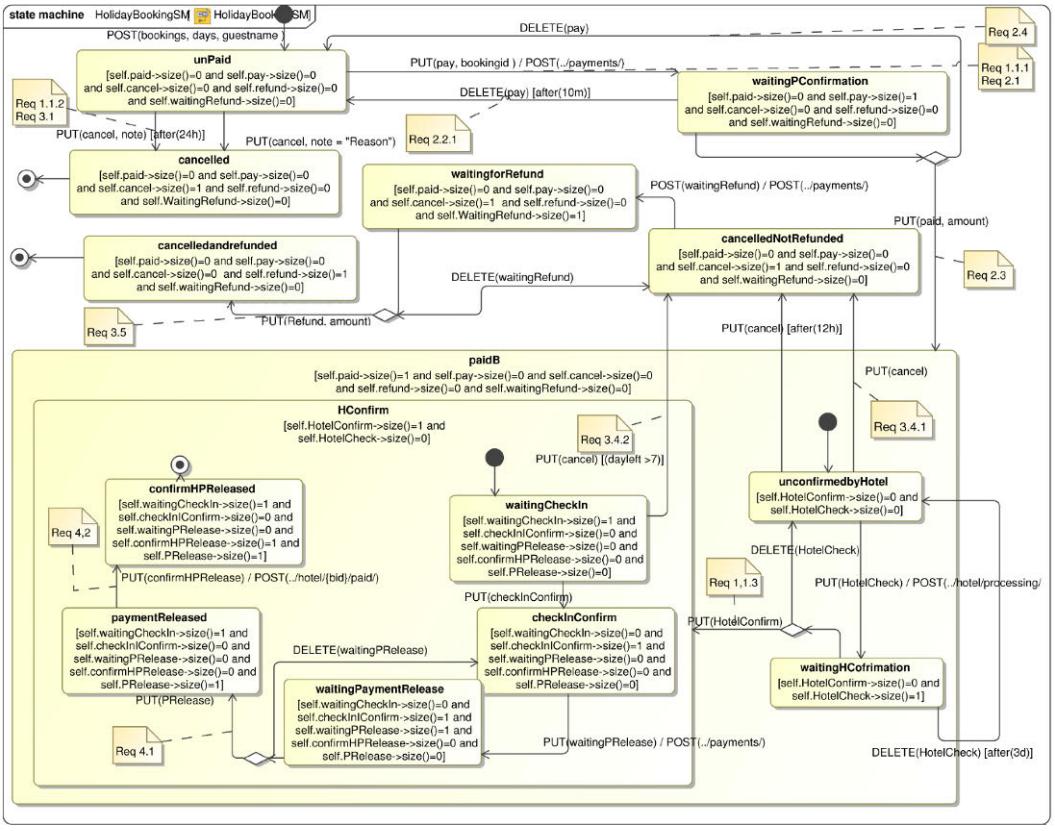


Figure 5: UML State Machine of Holiday Booking Composite REST Web Service.

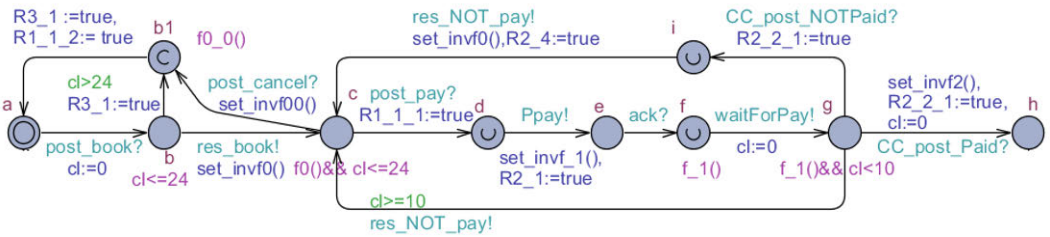


Figure 6: Excerpt of UPTA model of Holiday Booking Composite REST Web Service.

Test Environment. The environment model specifies the user actions, such as booking, canceling a reservation, requesting for the payment, paying, refunding and checking in. These are created from the observable channel synchronizations of the composite web service. The automaton in Figure 7 shows the environment model satisfying edge and requirements coverage. In Figure 7 they are encoded in the guard as a `verdict()` boolean function in the form: $r_1 \&\& \ldots r_n \&\& e_1 \ldots \&\& e_m$ where r_i and e_j are variables corresponding to requirements and, respectively, to edges of the composite web service in Figure 6. Whenever the verdict function evaluates to TRUE environment model can go to the final location.

Test Setup. Similar to Figure 2, the test setup comprises the TRON engine, the adapter, and the IUT. The IUT is a web service composition of three web services: Holiday Booking, Hotel and Payment Services. The test adapter composed of a set of test cases which satisfy the test requirements that are listed in Table1.

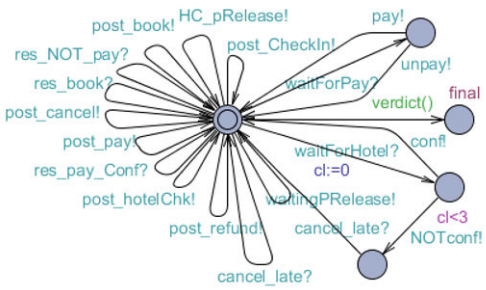


Figure 7: Environment model.

5 EVALUATION

The UML state machines of the HB composite REST web service had 14 states and 25 transitions. These were translated into a UPTA model with 34 locations and 46 edges. Similarly, the state machines of Payment service had 3 states and 4 transitions which transformed in to a UPTA model with 5 locations and 6 edges. The Hotel service had 4 states and 5 transitions that were translated into 7 locations and 9 edges. In addition, the environment model created had 4 locations and 13 edges.

One issue with using formal tools like UPPAAL for verification and test generation, is the scalability of the approach, due to the state space explosion. In contrast to offline test generation, where the entire state space has to be computed, in online test generation only the symbolic states following the current symbolic states have to computed. This reduces drastically the number of symbolic states making the test generation less prone to space explosion and thus more scalable. For instance, the number of explored symbolic states when generating, with the `verifyta` tool, traces satisfying complete edge coverage (i.e., $\&e_1 \ldots \&\&e_m$, where e_j are tracking variables corresponding to all m edges of the HBS models) was 974. In the contrast, the maximum number of symbolic states reported by TRON during a test session achieving complete edge coverage was 12 (see Figure11).

For benchmarking the verification process, we have used the `verifyta` command line utility of UPPAAL for verification of the specified 5 properties. We have used the `memtime` tool to measure the time and memory needed for verification. The result showed in average 0.20 seconds and 54996 KB of memory being used. Although the memory utilization depends heavily on the symbolic state space, it shows that the current size models leave room for scalability of the approach. A known limitation of UPPAAL is that the maximum memory size it can use is close to 4GB due to its 32-bit architecture. Figure 11 plots the evolution of the number of symbolic states for 10000 model time units (20 seconds).

In order to evaluate the efficiency of our approach, we compared the specification coverage with the code coverage yielded by a given test run. Since we had access to the source code of the IUT, we used the *coverage* tool for Python (pyt, 2013) to report the code coverage for each test session. The Table 2 lists results of several measurements:

Table 2: Correspondence between code coverage and edge coverage.

Run	Edge Coverage	Code Coverage
1	64 %	55%
2	80%	67%
3	100%	78%

Although many of the errors were caused by modeling mistakes, testing revealed some errors in the implementation as well. For instance, in the HB service, there was an error in sending *cancel* request and another error found in the POST header in *refund* request. Also in the Hotel service, the confirmation was sent by the wrong method, so it was rejected by Holiday Booking service.

In order to evaluate the fault detection capabilities of our approach, we have manually created 30 mutated versions of the original HB service program code. Each mutation had one fault seeded in the code, for instance replacing POST with DELETE, removing one line of the source code, change of logical conditions, etc. The faults were always seeded in those parts of the code that is covered when achieving 100% edge coverage of the model. We assumed that the original version of the composite web service is the correct one, as we were able to run the 100 test sessions in TRON against it. For each mutated version of the composite web service, we set the TRON to execute 100 test sessions against it. When a fault was discovered, the mutant was considered as *killed*. If the mutated statement has been covered by the test runs but no failure was detected, we mark it as *alive*. Out of the 30 mutated programs, 28 mutants were killed and 2 were alive. This resulted into a mutation score of 93.3%.

6 RELATED WORK

There is already a large body of work on using model checking techniques for validation and verification of web service compositions. Overviews of such works

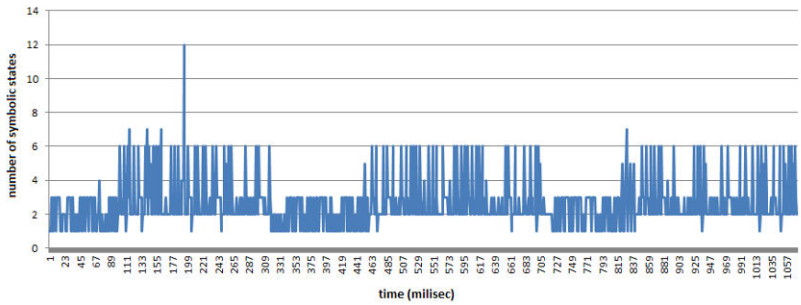


Figure 8: Evolution of symbolic states.

can be found in (Rusli et al., 2011) and (Bozkurt and other, 2010). Mostly authors have used web service specific specification languages as their starting point and converted the specifications to an intermediate language that is accepted by model checking tools. Then, by taking advantage of the model checking tool capabilities they performed simulation, verification or test generation via model-checking. Most of these works use the selected model-checking tool only for simulation and verification; only a handful generate abstract tests from the verification conditions, and in most cases it is not clear how the abstract test cases (i.e., the counterexample traces) are transformed into executable ones and executed. In the following, we will revisit those works which are most similar to ours.

We can distinguish roughly two approach categories: those that target the PROMELA language (Part and Peschke, 2003) which is the input language for the SPIN model-checker (Holzmann, 1997), and those that target the UPPAAL timed automata which is the input language for the UPPAAL model-checker (Behrmann et al.,).

In the first category, the vast majority of approaches have used BPEL or OWL-S(Martin et al., 2004) for the specification of the web service composition. For instance, Garcia (García-Fanjul et al., 2006) generates test cases using test case specifications created from counterexamples that are obtained from model checking. The transition coverage criterion is used to identify transitions in BPEL specification that define the test requirements for producing test cases. These transitions are mapped to the model and expressed in terms of LTL property expressions. Transition coverage is obtained by repeatedly executing the tool with each previously identified transition.

Fu. et al. (Fu et al., 2005) provide framework for analyzing, designing and verifying web service compositions. Their work provides both bottom-up and top-down approach to analyze the interaction between web services. In top-down approach, the desired conversation of a web service is specified as guarded automaton that are converted to PROMELA and used as input to SPIN model-checker. The bottom-up approach translates BPEL to guarded automaton and then used with SPIN model-checker after translating guarded automaton to PROMELA. The web service conversations are analyzed for synchronization in order to verify their compatibility.

One distinct approach is given by Huang et al. (Huang et al., 2005). They automatically translate OWL-S specification of composite web service into a C-like specification language and PDDL through an integrated process. These can be processed with the BLAST model-checker which can generate positive and negative test cases during model checking of a particular formula and test the web service using the test cases.

These works focus on BPEL processes and OWL-S, this makes them dependent on specific execution languages for SOAP based services whereas our work is not dependent on implementation and supports REST architectural style. In addition, their work does not support requirement traceability and is not clear how tests are generated and executed. Furthermore, the works that use the PROMELA language for specification do not address real-time properties, due to the limited support for time in PROMELA.

In the second category, researchers have targeted timed automata specifications. In (Cambronero et al., 2011), Cambronero et al. verify and validate web services choreography by translating a subset of WS-CDL into a network of timed automata and then use UPPAAL tool for validation and verification of the described system. They also capture capture requirements by extending KAOS goal model and implement them. The work is supported by WST tool that provides model transformation of timed composite web services (Cambronero et al., 2012). In (Dıaz et al., 2007), Diaz et. al also provide a translation from WS-BPEL to UPPAAL timed automata. Time properties are specified in WS-BPEL and translated to UPPAAL.

However, requirements are not traced explicitly, while verification and testing are not discussed.

Ibrahim and Al-Ani (Ibrahim and Al Ani, 2013) transform BPEL specification to UPAAAL. The specification includes safety and security non-functional properties which are later formulated into guards in the UPPAAL model which could be similar to our verification of requirements. They do not consider neither real-time properties nor test generation.

In (Guermouche and Godart, 2009), Nawal and Godart deal with checking the compatibility of web service choreography supporting asynchronous timed communications using model checking based approach. They use model-checker UPPAAL and present compatibility checking distinguishing between full and partial compatibility and full incompatibility of web services. Our work is somewhat similar to their work as we support time critical stateful REST webs service compositions using UPPAAL, however, in addition to verification we use UPPAAL with TRON to validate the implementation of the web services.

Zhang (Zhang et al., 2011) suggest the use of the temporal logic XYZ/ADL language (Zhu and Tang, 2003) for specifying web server compositions. They transform the specifications into a timed asynchronous communication model (TACM) which are verified in UPPAAL.

In (Lallali et al., 2008), uses BPEL specifications as a reference specification and transform them to an Intermediate Format (IF) based on timed automata and then propose an algorithm to generate test cases. Similar to our approach, tests are generated via simulation in a custom tool, where the exploration is guided by test purposes. One noticeable difference is that time properties are added manually to the IF specification, while we specify them at UML level.

These works provide approaches to verify and validate the service specifications by checking the properties of interest using UPPAAL tool, however our work, in addition to model checking the properties also performs conformance testing of the service composition via online model-based testing with the TRON tool and provides requirement traceability for non-deterministic systems.

7 CONCLUSION

We have presented an integrated approach to design and validate RESTful composite web services. In our approach, a service can invoke other services and exhibit complex and timed behavior, while still complying with the REST architectural style. We showed how to model the service composition in UML, including time properties. We modeled communicating web services and explicitly define the service invocations and receiving service calls.

We use model checking approach with UPPAAL model-checker to verify and validate our design models. From the verified specification, we generate tests using an online model-based testing tool. The use of online MBT proved beneficial as our system under test exhibits non-deterministic behavior due to concurrency and real-time domain.

With the help of requirements traceability mechanism we traced requirements to UML models and, via the UML→UPTA transformation to timed automata models. Their reachability is verified in UPPAAL and they are used as test goals during test generation. Linking requirements to generated tests allowed us to quickly see which requirements have been validated and which have not. In addition, it allows us to identify from which parts of the specification/implementation the detected error has originated.

We exemplified our approach with a relatively complex case study of a holiday booking web service and we provided preliminary evaluation results.

REFERENCES

(2013). Code coverage measurement for Python – coverage, v. 3.6. Online at https://pypi.python.org/pypi/coverage. retrieved: 20.08.2013.

(2013). Nomagic MagicDraw webpage at http://www.nomagic.com/products/magicdraw/.

Alur, R. et al. (1990). Model-checking for real-time systems. In *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 414–425. IEEE.

Behrmann, G. et al. Uppaal 4.0. In *QEST '06 Proceedings of the 3rd international conference on the Quantitative Evaluation of Systems*, pages 125 – 126. IEEE Computer Society Washington, DC, USA.

Birgit Demuth, C. W. (2009). Model and Object Verification by Using Dresden OCL. In *Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice,*, pages 81–89.

Bozkurt, M. and other (2010). Testing web services: A survey. *Department of Computer Science, Kings College London, Tech. Rep. TR-10-01*.

Budgen, D., Burn, A. J., Brereton, O. P., Kitchenham, B. A., and Pretorius, R. (2011). Empirical evidence about the uml: a systematic literature review. *Software: Practice and Experience*, 41(4):363–392.

Cambronero, M. et al. (2012). Wst: a tool supporting timed composite web services model transformation. *Simulation*, 88(3):349–364.

Cambronero, M. E. et al. (2011). Validation and verification of web services choreographies by using timed automata. *Journal of Logic and Algebraic Programming*, 80(1):25–49.

Clarke, E. M., Grumberg, O., and Long, D. E. (1994). Model checking and abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542.

Dıaz, G. et al. (2007). Model checking techniques applied to the design of web services. *CLEI Electronic Journal*, 10(2).

Fu, X. et al. (2005). Synchronizability of conversations among web services. *Software Engineering, IEEE Transactions on*, 31(12):1042–1055.

García-Fanjul, J. et al. (2006). Generating test cases specifications for BPEL compositions of web services using SPIN. In *International Workshop on Web Services–Modeling and Testing (WS-MaTe 2006)*, page 83.

Guermouche, N. and Godart, C. (2009). Timed model checking based approach for web services analysis. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 213–221. IEEE.

Hessel, A. et al. (2008). Testing Real-Time systems using UPPAAL. In *Formal Methods and Testing*, pages 77–117. Springer-Verlag.

Holovaty, A. and Kaplan-Moss, J. (2009). *The definitive guide to Django: Web development done right*. Apress.

Holzmann, G. J. (1997). The model checker SPIN. *Software Engineering, IEEE Transactions on*, 23(5):279–295.

Huang, H. et al. (2005). Automated model checking and testing for composite web services. In *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*, pages 300–307. IEEE.

Ibrahim, N. and Al Ani, I. (2013). Beyond functional verification of web services compositions. *Journal of Emerging Trends in Computing and Information Sciences*, 4, Special Issue:25–30.

Koskinen, M. et al. (2013). Combining Model-based Testing and Continuous Integration. In *Proceddings of the International Conference on Software Engineering Advances (ICSEA 2013)*. IARIA. TO APPEAR.

Lallali, M. et al. (2008). Automatic timed test case generation for web services composition. In *on Web Services, 2008. ECOWS'08. IEEE Sixth European Conference*, pages 53–62. IEEE.

Larsen, K. G. et al. (1997). UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152.

Larsen, K. G. et al. (2009). UPPAAL Tron user manual. *CISS, BRICS, Aalborg University, Aalborg, Denmark*.

Martin, D. et al. (2004). OWL-S: Semantic markup for web services. *W3C member submission*, 22:2007–04.

Nobakht, M. and Truscan, D. (2013). An Approach for Validation, Verification, and Model-Based Testing of UML-Based Real-Time Systems. In Lavazza, L., Oberhauser, R., Martin, A., Hassine, J., Gebhart, M., and Jntti, M., editors, *ICSEA 2013, The Eighth International Conference on Software Engineering Advances*, pages 79–85. IARIA.

Part, I. and Peschke, M. (2003). Design and validation of computer protocols.

Porres, I. and Rauf, I. (2011). Modeling behavioral RESTful web service interfaces in UML. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1598–1605. ACM.

Rauf, I. et al. (2012). Analyzing Consistency of Behavioral REST Web Service Interfaces. In Silva, J. and Tiezzi, F., editors, *The 8th International Workshop on Automated Specification and Verification of Web Systems*, Electronic Proceedings in Theoretical Computer Science, page 115. EPTCS.

Rauf, I. and Porres, I. (2011). Beyond CRUD. pages 117–135.

Rauf, I., Siavashi, F., Truscan, D., and Porres, I. (2013). An Integrated Approach to Design and Validate REST Web Service Compositions. Technical Report 1097.

Richardson, L. and Ruby, S. (2008). *RESTful web services*. O'Reilly.

Rusli, H. M. et al. (2011). Testing Web services composition: a mapping study. *Communications of the IBIMA*, 2007:34–48.

UML, O. (2009). 2.2 Superstructure Specification. *OMG ed*. http://www.omg.org/spec/UML/2.2/.

Utting, M. and Legeard, B. (2007). *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Zhang, G. et al. (2011). Model checking for asynchronous web service composition based on xyz/adl. In *Web Information Systems and Mining*, pages 428–435. Springer.

Zhu, X.-Y. and Tang, Z.-S. (2003). A temporal logic-based software architecture description language xyz/adl. *Journal of Software*, 14(4):713–720.

Publication II

Environment Modeling in Model-based Testing: Concepts, Prospects and Research Challenges: A Systematic Literature Review

Siavashi, Faezeh and Truscan, Dragos (2015). In *EASE 2015 - Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–6. ACM

Environment Modeling in Model-Based Testing: Concepts, Prospects and Research Challenges

A Systematic Literature Review

Faezeh Siavashi
Åbo Akademi University
Joukahainengatan 3-5
20520 Turku, Finland
faezeh.siavashi@abo.fi

Dragos Truscan
Åbo Akademi University
Joukahainengatan 3-5
20520 Turku, Finland
dragos.truscan@abo.fi

ABSTRACT

In this paper, we describe a systematic literature review (SLR) on the use of environment models in model-based testing (MBT). By applying selection criteria, we narrowed down the identified studies from two hundred ninety seven papers to sixty one papers which are used in this analysis. The results show that environment models are especially useful in testing systems with high complexity and non-deterministic behaviors in terms of facilitating automatic test generation. However, building environment models is not a trivial task due to the lack of a systematic methodology and of supporting tools for automation.

Categories and Subject Descriptors

I.6.7 [**Simulation and Modeling**]: Simulation Support Systems—*Environments*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*

Keywords

environment model, software testing, model-based testing, systematic literature review

1. INTRODUCTION

Model-Based Testing (MBT) is a black-box testing technique that generates tests from abstract behavioral models [23]. The models can represent either the expected behavior of the system under test (SUT) or of its environment, or in some cases of both. In this context, *abstraction* is beneficial in hiding unnecessary details of the implementation and reducing the complexity of testing. Nevertheless, it is also essential that a test model is *detailed* enough in order to generate effective test cases. Finding the right level of abstraction for the test model is one of the challenges in MBT [23].

EASE '15, April 27 - 29, 2015, Nanjing, China

http://dx.doi.org/10.1145/2745802.2745830.

MBT can be used for both online and offline testing. In online testing, the test inputs are generated and executed on-the-fly, whereas in offline testing, test inputs are first generated and later on executed as a batch [22].

In complex computer systems, which operate in environments with large numbers of events and different timings, testing leads to a large number of test cases to cover all possible states of the system. Executing all possible test cases becomes time consuming and unfeasible. Therefore, more advanced methods are required in MBT in order to optimize the number of test cases and reduce the complexity of testing [9].

Environment modeling is an activity that specifies a part of the real world, in which the system is integrated. The process of environment modeling results into an *environment model*, which captures all relevant assumptions and contains all interactions with the SUT [11]. Environment modeling can help addressing the problem of testing complex systems, since one can use environment models to generate automatic test cases for a particular behavior of the SUT.

The main objective of this SLR is to understand how an environment model can enable MBT and what are the current problems and research challenges. In this paper, we do not attempt to compare different approaches, instead, we extract the information as presented by the authors of the available literature in order to present a complete picture of the research done on this topic. To our best of knowledge, this is the first systematic literature review on different approaches in environment modeling.

The remainder of this paper proceeds as follows: in Section 2, we define the research method, provide research questions, and describe the material selection process based on the defined selection criteria. In Section 3, we answer the research questions and present the data analysis from our findings. In Section 4, we discuss validity threats, while in Section 5, we provide a discussion and conclusions.

2. RESEARCH METHOD

In this work, we follow the research method suggested by Kitchenham and Brereton [13] for conducting a systematic literature review. However, we describe a summary of the process here, while deferring more details to [20].

Research questions: The following research questions are addressed in this paper:

- *RQ1:* What are the characteristics of the environment models used for MBT?
- *RQ2:* What are the advantages of using environment models in MBT?
- *RQ3:* What formalism and tools have been used for creating environment models in MBT?
- *RQ4:* What problems and challenges have been observed by researchers using environment models in MBT?

Search terms: First, we selected a set of keywords from the research questions and then defined the search term:

("environment model" OR "environment behavior model" OR "environmental model" OR "environmental modelling" OR "environment modelling" OR "environment modeling" OR "environmental modeling") AND ("model-based testing" OR "model based testing"OR testing OR test OR "software testing")

Sources of studies: The electronic libraries that we used for searching are: ACM digital library, IEEE Explore, Science Direct, Springer, and Google Scholar. The reason for using Google Scholar is to ensure that we covered all available and relevant papers that are published by miscellaneous publishers or shared in other databases.

Selection criteria: A set of inclusion and exclusion criteria has been defined in order to collect relevant studies and filter out irrelevant ones. The inclusion criteria were:

- The objective of the study should be to discuss, apply or investigate the environment model methodologies for the purpose of testing.
- The studies must be written in English.
- The study should be published in a journal or conference proceedings.
- The study should answer at least one of the research questions.
- The study should be published between the years 2000 and 2014 (September).

The exclusion criteria are:

- The studies for which only extended abstracts were available.
- The papers that are about environmental engineering or biological studies or other studies outside the scope of software engineering/testing.
- Master's theses and Doctoral monographs. We assumed that these works have been previously reported and presented as conference or journal publications.

The inclusion and exclusion criteria were applied during the selection process in parallel with reading the full papers.

Procedure of selecting primary studies:

Step 1. 297 studies were identified by using the search terms in the electronic libraries.

Step2. We reviewed *title* and *abstract* of the identified papers and selected 120 studies.

Step 3. We read the *content* of the selected studies and applied the selection criteria. In parallel, we made a data extraction form in our Excel spread sheet and recorded details of each study, such as authors, year of publication, etc. In this step, we reduced the number of studies to 63.

Step 4. We added all relevant references that we found in 63 papers and applied Steps 1-3 on them (*snowballing* [10]). From the references, we selected 5 more studies, so the total number from this step reached to 68.

Step 5: We found that 7 papers were redundant, so we removed them and 61 studies remained.

In this paper, we report the findings that we retrieved by studying 61 studies, which we refer to them as *primary studies*.

For each repository the number of selected papers in each step is shown in Table 1.

Table 1: The number of selected papers in each repository and in each step of SLR

Database	Step1	Step2	Step3	Step4	Step5	%
ACM	33	12	7	8	7	11%
IEEE Explore	125	48	25	25	24	39%
ScienceDirect	26	7	4	4	4	7%
Springer	72	36	20	22	18	30%
Google Scholar*	41	17	7	9	8	13%
Total	297	120	63	68	**61**	100%

* Only papers that are published in miscellaneous repositories

The last column in Table 1 shows that a large percentage of the publications, 39%, comes from IEEE Explore (24 studies), followed by 30% papers from Springer (18 studies). Google Scholar and ACM have 13% and 11% respectively (8 and 7 studies). Only 7% of the papers are selected from Science Direct (4 studies). Here, Google Scholar has a smaller percentage, since we removed the studies that were originally found in the other databases.

Figure 1 shows the number of primary studies from 2000-2014 by five years interval. It can be noticed that in recent years there has been increased attention towards environment modeling in MBT. This may be due to the growing rate of the complexity of computer systems and applications and subsequently the testing process is becoming more complex. Thus, using environment models as a technique for reducing the complexity is becoming more popular.

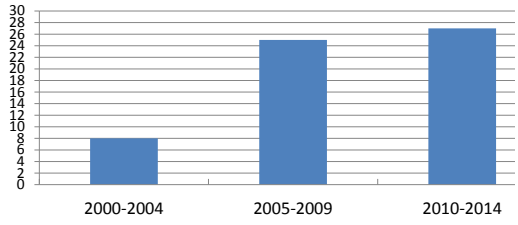


Figure 1: Number of primary papers by five years intervals

3. RESULTS

In this section, we present how the literature answers the research questions. We start with the information about the characteristics of environment models (RQ1), followed by an overview of the role of environment models in MBT (RQ2). Next, we present a list of modeling formalism, tools and methodologies that are based on environment models (RQ3). Finally, we look at the current limitations and challenges that are reported by the literature (RQ4). Citations of the selected papers are given in this section for further reading.

Out of 61 studies, 95% are empirical studies (i.e., experiments on case studies). 5% were theoretical studies which are based on providing concepts, formal definitions, methodology or references to other work.

Due to the lack of space, we only present a short overview of our findings. More details of the analysis and references to the primary studies are presented in [20].

3.1 RQ1 - What are the characteristics of environment models used for MBT?

All primary studies, either implicitly or explicitly, presented general characteristics of using an environment model in their testing approaches. Below, we present them in more detail:

- **Specific aspects of the SUT:** An environment model can be specified in a way that it covers only certain part(s) of the SUT in order to test those parts. Therefore, different parts of a system can be tested separately. Besides, environment models can be defined in a way that they contain different test scenarios to violate specific functionality of the SUT. Twelve out of 61 primary studies fall in this category.

- **Non-determinism:** Non-determinism is an important feature in modeling complex systems with unpredictable environments. It is not a trivial task to model a system which can accept and react to unpredictable conditions. Therefore, using environment models can help in defining the continuous and unpredictable interactions [7]. Non-deterministic environment models give more options to choose among enabled test inputs. We found that 3 primary studies present environment models with non-deterministic behavior.

- **Include multiple entities**: A SUT can have communications with different environment entities such as users, other systems, or a part of the actual environment (i.g. temperature or the sunlight). There are two ways in specifying multiple environments in a test model: it can be defined as a single model containing all environment interactions, or it can be defined as multiple environment components. In both cases, the environment model should capture all assumptions of the environment. Also it should control the interactions among the entities as well. Ten papers presented their work using different environment entities.

- **Dynamic and static behavior:** An environment model is able to support both static and dynamic behaviors of a SUT. Static behaviors mostly indicate what are the inputs of the environment model into the SUT and what type of data and properties are supported by the environment model. In change, dynamic behaviors specify the interactions of the environment model with the SUT, the timing properties and the order of test inputs based on the current outputs during test execution. Six papers used environment models to specify dynamic and static behavior.

- **Abstraction:** A model is an abstract specification of the real world. An abstract model can be defined by restricting the range of input values, omitting some functions or reducing the time span. Environment models can be modeled to only focus on the more abstract interactions. Six papers in our review emphasize on the importance of the abstraction level of the test model when using environment models.

- **Control of time:** Environment models generate timed input traces, which can occur in the real environment, to ensure that the system can satisfy specific timing properties. This characteristics can be beneficial in modeling and testing real-time systems. Nine studies show that environment models can be used for testing systems with timing properties, such as real-time systems.

- **Explicit behaviors:** Having separate models for the SUT and its environment has advantages modifying each of them separately. For instance, when environment models are used in test generation, they typically encode test goals. Whenever test requirements change, only the environment models should be changed. Six papers argue that environment models used in MBT have explicit nature depicting the expected behavior of the system.

- **Source of knowledge for modeling:** The source of knowledge about an environment can come from the requirements, or from the assumptions of the test designers. The requirements are a list of the specifications that a system must follow and need to be tested. When the system specifications are not available, assumptions of the environment can be observed from the actual environment and then formally defined. From the literature, we found that seventeen papers define their environment models from the requirements that are provided in the documentation of the SUT. Also, sixteen papers explicitly claimed that they define their environment via assumptions.

3.2 RQ2 - What are the advantages of using environment models in MBT?

Based on the MBT taxonomy illustrated by Utting et al. [23], testing of a system using MBT consists of three main dimensions: modeling, test generation, and test execution. Our findings from the primary studies show that applying environment modeling can be beneficial in all these dimensions. Environment modeling brings the following benefits:

- **Test oracle creation:** In MBT, a test oracle is usually encoded in the test model, and during test generation it is assigned to the generated test cases. In complex systems, in order to reduce the complexity of testing and focus on certain functionality, environment model can be used to model certain test oracles. Three of the primary studies discussed explicitly about test oracle generation using environment models.

- **Automated test generation:** In online testing, it is essential to automatically generate test cases. Automation prevents human errors, which might occur with manual testing, and reduces the time of generating test cases. A *Test harness* (automated testing framework) can be built by a set of test data to automatically run tests and monitor the outputs. Environment model can be used in automation of testing. Twelve papers present that environment models are used to generate test inputs for the SUT during testing.

- **Optimal test generation:** Optimized test case generation is discussed as a benefit of using environment models in testing, making the testing process more efficient. It is caused by having support for abstraction in environment modeling. This advantage is presented in five papers.

- **Reducing the size of the state space:** One of the main issues in executing and simulating complex models (or models with a wide range of inputs) is that the number of symbolic states that should be explored increases during test execution, which causes the system to run out of memory. This problem is known as state space explosion. Reducing the size of the state space can be done by using bounded data types, resetting clock variables, or defining model invariants which limit the enabled states at a given time. Five studies report that well-defined environment models significantly reduce the search space by constraining the ranges of certain test inputs.

- **Early validation of requirements:** Using explicit environment models can be helpful for validating the requirements at the early stage of the system design. Inconsistencies in specifications can be detected when building the models. In addition, they can be used to guide the simulation of early prototypes of the SUT. Two primary studies discuss this issue as an advantage of using environment models.

- **Re-usability:** Different SUTs or different versions (regression) of the same SUT can be tested using a single environment model (see for instance [4]). Generally, environment models will be changed relatively rarely unless some errors originating from requirements are discovered during testing. Therefore, the modeling efforts can be reduced by using the same models in different testing contexts. Five primary studies report this advantage.

- **Different testing types:** Our findings show that environment models can be applied in different testing approaches, such as *safety testing* (5 studies), *robustness testing* (2 studies) and *regression testing* (3 studies). Safety and robustness can be verified by creating erroneous test inputs to the SUT. Moreover, since environment models are able to test certain parts of the SUT, applying them in regression testing can improve the testing effort. Also, environment models have been studied in Aspect Oriented Modeling (AOM) [6] where are known as context models. In AOM, an environment consists of some smaller models, which communicate with each other and with the SUT.

- **Applicable into all testing levels:** The primary studies show that environment models can be applied at all testing levels: *system*, *integration*, and *unit*. The majority of the studies describe applying environment modeling at the system level (43 studies) and few number of them report using environment models at the integration level (3 studies) and the unit level (6 studies).

3.3 RQ3 - What formalisms and tools have been used for creating environment models in MBT?

We detected a large range of modeling languages and variety of modeling and testing tools from the primary studies. In this section, we provide the list of the most frequent languages and tools, their references, and briefly discuss some of the most referenced tools.

- **UML:** The majority of the studies use the Unified Modeling Language (UML) [19] as the modeling language. In our review, 20 primary studies are built on UML either by using its standard behavioral diagrams such as sequence and state diagrams, or UML profiles such as Fondue [16], MARTE [1], SysML [8] and MbRTE (Executable model-based robustness testing environment) [24]. The structure of the environment is a model that describes all various entities and their relationships (also known as a domain model in the literature) and consists of one or more environment components. The domain model provides the information on all relationships and properties between the components.

- **Timed-Automata:** Six primary studies present the MBT approaches using Timed Automata (TA) [3]. The tools used with TA are UPPAAL, its an online testing tool (UPPAAL TRON) and Maude. UPPAAL is a model-checker which allows simulation and verification of TA-based specifications. Environment models can be specified in UPPAAL as deterministic or non-deterministic. UPPAAL-TRON is an online testing tool that generates test cases from TA and executes them against the SUT [14]. Maude is a tool based on supporting equational logic and rewriting logic [15]. It represents model generation rules by applying rewriting theories, instead of describing a model directly. It can be applied for modifying the TA model.

- **AEG:** Six primary studies present their experiments on testing with Attributed Event Grammar (AEG) [4], which is used for testing real-time and embedded systems. Event grammars are text-based and are appropriate for specifying the dynamic environment with an arbitrary number of actors and events. Models based on event grammars can be designed either for the environment or for the environment and the SUT. They can also contain hazardous states to assess the safety of the SUT. The environment models can be used to automatically generate test cases.

- **Petri Nets:** Four primary studies are based on Petri Nets, using the TINA tool (TIme Petri Net Analyser) [5]. TINA is a software environment for the editing and analysis of Petri nets and Timed Petri nets. The environment models in TINA have the same properties as the models defined in UPPAAL. Similar, to UPPAAL, the environment model supports both non-deterministic and deterministic assumptions.

- **Lutin:** Three primary studies discuss testing with the Lutin language [18]. Lutin is a test-based language for specifying random reactive behaviors, specially developed for modeling and testing reactive systems. The Lurette test generator is used for random or guided test case generation.

- **BEG:** Two primary studies show how environment models can be designed in the Bandera Environment Generator (BEG) [21], a tool that automates the generation of environments for model-checking Java programs. The tool is able to decompose a given Java program into small modules and create the environment models out of it.

Table 2 shows all formalism and modeling tools that have been used for environment modeling.

Table 2: Formalism and tools for environment modeling

Formalism/ Languages	Tools	# Studies
UML	UML tools	10
UML/MARTE		5
UML Fondue		2
UML/SysML		1
ESML		1
MbRTE		1
Timed Automata	UPPAAL	4
	UPPAAL TRON	1
	Maude	1
Event Grammar	AEG	6
Petri nets	TINA	4
Lutin	Lurette	3
Java	BEG	2
QR	QR models	2
TSML	AUTOSAR	1
Esterel	Esterel	1
SPIN	Promela	1
TML	JUMBL	1
Markov model	Markov model	1
TTCN-3	TTCN-3	1
SLAM	SLAM	1
DoB	Degree-of-Belief(DoB)	1
BLAST	BLAST	1

3.4 RQ4 - What problems and challenges have been observed by researchers using environment models in MBT?

We identified several studies that describe problems in MBT using environment models. Also, they identify research areas in MBT for further investigations.

- **Lack of methodology for environment modeling:** Many of the identified studies use environment models for testing, but without discussing explicitly how they are created. Methodological aspects of creating environment models are only discussed in a limited number of papers (e.g., in [9] for UML models). Kishi and Noda emphasize the importance of having a strategy for defining environment model in aspect oriented approaches [12]. Dividing an environment model into several sub-models requires a well-defined methodology as well.

- **Test adaptation is manually implemented:** The studies show that although once an environment model is specified, then the test generation will be automatic. Yet creating the test adapter which can transform the model-level test inputs into executable test cases is manual and error prone process (e.g. in [17]).

- **Multiple test adaptations:** In systems with multiple environment entities, multiple test adaptations are required [2]. The reason is that the interactions among the environment entities as well as interactions between the environments and the SUT are usually complex.

- **Lack of extensive experiments:** The results of our findings show that environment modeling is still immature in some aspects of MBT. For instance, reports have shown that environment models are good choices in robustness testing [24] and regression testing. However, there are very few studies which applied the environment models in practice. Moreover, reusability of environment model can be investigated more and other advantages of using environment models can be studied in more details.

- **Complex specifications:** It is still a challenge to expand the environment modeling in complex systems and for more complicated environments. As it is noted by Auguston et al., more methods are required in order to evaluate environment modeling in large and complex SUTs with large number of test cases automated by the environment [4].

4. VALIDITY THREATS

There are four main threats related to our SLR. One is related to studies that we might have missed in our search. Despite the fact that we followed all the steps mentioned in the systematic review process, we cannot be certain that all of the approaches that use environment models in MBT have been identified. Some exclusions were made during reading titles and abstracts, which could have removed studies with relevant content. However, in the second round of the search (snowballing), we made the effort of finding all the studies that were we did not find (or excluded) in the initial round.

Next threat is that there might be some studies that can not be found in any of the selected repositories. We are aware that there are some repositories (e.g. Scopus) that may have more collections of studies. Nevertheless, we converged our search into those repositories to which we could have access and in addition we included Google Scholar to find additional works.

Another threat is that the measurements may not be reliable. This can be caused from lack of reliability in the searching databases, or from the lack of metrics of comparing and selecting the papers. We made all efforts to obtain all published studies that are available in the databases. For each resource, we recorded the details and the information about how and where we searched, in order to make the search repeatable in the future. Moreover, as mentioned in the search and selection process, we searched several different repositories as well as books, conference proceedings and journals, where the most updated works and tools are presented.

Moreover, judgmental errors may have happened during the classification of the papers. We followed the terminologies and classifications that are defined by the literature. Besides, for each classification, we provide the referenced definition, to prevent ambiguity. Based on the quality assessment that we presented in [20], more than 84% of the studies are evaluated as high or very high quality. Thus, the reliability of our measurements can be acknowledged.

5. DISCUSSION AND CONCLUSIONS

In this SLR, we defined research questions about environment modeling in MBT. We searched the keywords in different resources based on the defined inclusion and exclusion criteria. Sixty-one primary studies are found answering the research questions and the data are extracted and analyzed.

We identified the main characteristics of an environment model and provided a list of its advantages that are reported in the literature. From the characteristics and advantages, we clarified that using environment models can be helpful

in robustness testing, safety testing and regression testing. Also, we showed that in what modeling languages environment models have been studied.

The limitations and current challenges in testing with environment models were summarized as well. The studies report that although the environment modeling helps in the automation of test case generation, yet some case test cases are written manually. Also, the transformation from the symbolic test cases to test scripts is still a manual process.

More research is needed to develop some statistical methods to evaluate and analyze the applicability of environment models in MBT.

From the literature, we clearly conclude that there is still plenty of potential for investigating environment modeling and automating test generation specially w.r.t. non-functional testing approaches. Extensions of the current methodologies are needed to overcome these limitations.

6. REFERENCES

[1] The UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems. `http://www.omgmarte.org/`, 2013. [Online; accessed 22-December-2014].

[2] N. Adjir, P. De Saqui-Sannes, and K. Rahmouni. Testing Real-Time Systems Using TINA. In M. Núñez, P. Baker, and M. Merayo, editors, *Testing of Software and Communication Systems*, volume 5826 of *LNCS*, pages 1–15. Springer Berlin Heidelberg, 2009.

[3] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[4] M. Auguston, J. B. Michael, and M.-T. Shing. Environment behavior models for automation of testing and assessment of system safety . *Information and Software Technology*, 48(10):971 – 980, 2006. Advances in Model-based Testing.

[5] B. Berthomieu and F. Vernadat. Time petri nets analysis with TINA. In *Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on*, pages 123–124. IEEE, 2006.

[6] T. Elrad, O. Aldawud, and A. Bader. Aspect-oriented modeling: Bridging the gap between implementation and design. In *Generative Programming and Component Engineering*, pages 189–201. Springer, 2002.

[7] G. Fraser and F. Wotawa. Test-Case Generation and Coverage Analysis for Nondeterministic Systems Using Model-Checkers. In *International Conference on Software Engineering Advances*, pages 45–45, Aug 2007.

[8] S. Friedenthal, A. Moore, and R. Steiner. *A practical guide to SysML: the systems modeling language*. Elsevier, 2011.

[9] M. Iqbal, A. Arcuri, and L. Briand. Environment Modeling with UML/MARTE to Support Black-Box System Testing for Real-Time Embedded Systems: Methodology and Industrial Case Studies. In D. Petriu, N. Rouquette, and Ø. Haugen, editors, *Model Driven Engineering Languages and Systems*, LNCS, pages 286–300. Springer Berlin Heidelberg, 2010.

[10] S. Jalali and C. Wohlin. Systematic Literature Studies: Database Searches vs. Backward Snowballing. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM, pages 29–38, New York, NY, USA, 2012. ACM.

[11] G. Karsai, S. Neema, and D. Sharp. Model-driven architecture for embedded software: A synopsis and an example. *Science of Computer Programming*, 73(1):26 – 38, 2008. Special Issue on Foundations and Applications of Model Driven Architecture (MDA).

[12] T. Kishi and N. Noda. Aspect-oriented context modeling for embedded systems. *Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, page 69, 2004.

[13] B. Kitchenham, O. Pearl Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman. Systematic Literature Reviews in Software Engineering - A Systematic Literature Review. *Inf. Softw. Technol.*, 51(1):7–15, Jan. 2009.

[14] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou. Testing real-time embedded software using uppaal-tron: An industrial case study. In *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05, pages 299–306, New York, NY, USA, 2005. ACM.

[15] G. Li, S. Yuen, and M. Adachi. Environmental simulation of real-time systems with nested interrupts. In *Theoretical Aspects of Software Engineering, 2009. TASE 2009. Third IEEE International Symposium on*, pages 21–28, July 2009.

[16] L. Lucio, L. Pedro, and D. Buchs. A methodology and a framework for model-based testing. In *Rapid Integration of Software Engineering Techniques*, pages 57–70. Springer, 2005.

[17] M. Mews, J. Svacina, and S. Weissleder. From AUTOSAR Models to Co-simulation for MiL-Testing in the Automotive Domain. In *International Conference on Software Testing, Verification and Validation*, pages 519–528, April 2012.

[18] R. Pascal, R. Yvan, and J. Erwan. Lutin: A language for specifying and executing reactive scenarios. *EURASIP Journal on Embedded Systems*, 2008, 2008.

[19] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The*. Pearson Higher Education, 2004.

[20] F. Siavashi and D. Truscan. A systematic literature review on environment modeling techniques in model-based testing. Technical Report 1129, 2015, `http://tucs.fi/publications/view/?pub_id=tSiTr15a`.

[21] O. Tkachuk, M. Dwyer, and C. Pasareanu. Automated environment generation for software model checking. In *International Conference on Automated Software Engineering, 2003. Proceedings.* , pages 116–127, Oct 2003.

[22] M. Utting and B. Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2010.

[23] M. Utting, A. Pretschner, and B. Legeard. A Taxonomy of Model-based Testing Approaches. *Softw. Test. Verif. Reliab.*, 22(5):297–312, Aug. 2012.

[24] S. Yang, B. Liu, Shihai, and M. Lu. Model-based robustness testing for avionics-embedded software . *Chinese Journal of Aeronautics*, 26(3):730 – 740, 2013.

Publication III

Scenario-Based Design and Validation of REST Web Service Compositions

Irum Rauf and Faezeh Siavashi and Dragos Truscan and Ivan Porres (2015). In *Web Information Systems and Technologies – Revised Selected Papers*, pages 145–160. Springer International Publishing

Scenario-based Design and Validation of REST Web Service Compositions

Irum Rauf, Faezeh Siavashi, Dragos Truscan and Ivan Porres

Åbo Akademi University, Dept. of Information Technologies, Turku, Finland,
[irum.rauf, faezeh.siavashi, dragos.truscan, ivan.porres]@abo.fi

Abstract. We present an approach to design and validate RESTful composite web services based on user scenarios. We use the Unified Modeling Language (UML) to specify the requirements, behavior and published resources of each web service. In our approach, a service can invoke other services and exhibit complex and timed behavior while still complying with the REST architectural style. We specify user scenarios via UML Sequence Diagrams. The service specifications are transformed into UPPAAL timed automata for verification and test generation. The service requirements are propagated to the UPPAAL timed automata during the transformation. Their reachability is verified in UPPAAL and they are used for computing coverage level during test generation. We validate our approach with a case study of a holiday booking web service.

Keywords: REST, web service composition, model-based testing, UPPAAL, TRON

1 Introduction

REST (REpresentational State Transfer) web services are built on the principles of the REST architectural style [12] which aims at producing scalable and extensible web services. The REST interface offers a CRUD interface (create, retrieve, update and delete) to its users via a set of standard HTTP methods. In additions, REST offers stateless behavior that facilitates scalability.

Different web services published over the internet can be composed into new composite web services which fulfill new service goals using the functionality of partner web services. Automated systems, for example hotel reservation systems, are often built as stateful composite services that require a certain sequence of method invocations that must be followed in order to fulfill service goals. Creating such composite services with advanced scenarios and REST features requires rigorous development approaches that are capable of creating web services that can be trusted for their behavior.

With the rise in use of REST web services in different domains offering complex and timed scenarios, there is an increasing need for validation approaches to effectively and efficiently detect faults in the specifications and implementations of such services.

In this article, we present a scenario-based validation and verification approach that can help the service developer in improving the quality of service specifications and implementations. The approach supports the creation of timed and stateful behavior with the confidence that the service fulfills its advertised functionality. The Web Service Composition (WSC) is specified using the Unified Modeling Language (UML) starting from the requirements of the WSC. A code skeleton of the WSC is automatically generated and manually completed by the developer. In order to perform validation and verification of the composition, the UML specifications are transformed into UPPAAL timed automata (UPTA). We use the UPPAAL tool set [23] to simulate the specifications and to verify their properties via model-checking. We also use them to automatically generate tests in order to validate the implementation.

Requirements traceability is an important component of our approach. The requirements of the composition are included in the UML specifications and then propagated to UPTA. They are used for both verifying the reachability of those model elements implementing them and for reasoning about the coverage level of the tests generated. Upon detecting failures, the traced requirements are used to trace back errors either in the models or in the implementation.

We exemplify and validate our approach with a relatively complex example of a holiday booking composite REST web service extracted from an industrial application. The example shows how stateful and timed web services offering complex scenarios and involving other web services can be constructed efficiently using our approach.

The paper is organized as follows: Section 2 presents our approach and the tool support is discussed in Section 3. The case study is presented in Section 4, followed by the evaluation of the approach in Section 5. The related work is discussed in Section 6 and conclusions are drawn in Section 7.

2 Our Approach

Our scenario-driven approach to verify and test the composite REST web service is shown in Figure 1. We start by inferring service requirements in tabular format from specification document and the corresponding user scenarios from the specification document of the REST WSC. Each user scenario is detailed by one or several UML sequence diagrams. In addition, we build several perspectives of the WSC such as a resource, a behavioral and a domain model using UML class and state machine diagrams. This is an extension of our previous work, in which we designed behavioral interfaces for web services that were RESTful by construction [26]. We transform the service design models to UPTA, which are simulated and model-checked by reasoning the properties such as deadlock, liveness, reachability, and safety. If inconsistencies are found, the UML-based service design models are updated. These design models are used to implement the service in the Python-based Django web framework [16] using our partial code generation tool [26] which generates code skeletons with pre- and post-conditions for every service method. The skeleton is manually completed by the

service designer. The verified UPTA specifications are used for online model-based conformance testing of the implementation.

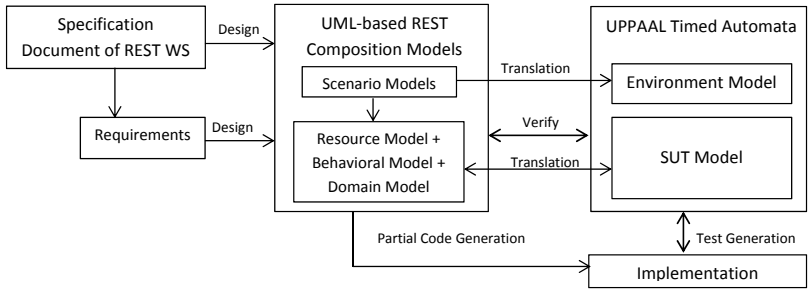


Fig. 1. Scenario-based V&V Approach for REST CWS

Requirements Traceability. Service requirements are inferred from the specification document and they serve as service goals. A service should be checked for its service goals to validate that the service does what it is required to do. By addressing the service requirements at the design phase and propagating them to the verification and validation stages, we provide a mechanism by which a service implementation can be validated for its goals and the unfulfilled requirements can be traced back to the design phase to find faults in the design.
Requirements Table: Service requirements are generally domain-specific since they are inferred from the specifications. We infer functional and temporal requirements from the specification document into a table and number them. These requirements are attached to the UML state machine (SM) as *comments* on the transitions and are propagated to UPTA such that the links between requirements and the model elements are preserved. These requirements are included in all the models and traced throughout the process, i.e., at UML, UPTA and test level, respectively. The requirements are formulated as reachability properties in UPTA with the purpose of verifying them during simulation. Each requirement label is translated into a boolean variable (initialized to *False*) and attached to the corresponding edge in UPTA.
Scenario Models: The behavioral requirements are elicited as scenario models using UML sequence diagrams. These scenario models are translated to environment model in UPTA since these scenarios define different conditions under which the composite service can be invoked.

We require that our testing approach must validate that the service requirements are met by IUT, and the service works correctly in different scenarios, in order to build confidence of the developer that the system is doing what it is required to do. Thus, the coverage level of scenarios and requirements is monitored during test generation and execution. Once the test report is available, we can check which requirements have been validated and which have failed. The main strength of using both the requirements table and scenario models in our approach is that the former helps in tracing the unfulfilled requirements to the

design models and locating the faults in the design of the service. On the other hand, the later helps in determining if the service works fine in different scenarios and identify under which conditions the service shows a faulty behavior.

REST Composition Models. The web service compositions that we build exhibit RESTful features such as addressability, connectivity, statelessness and uniform interface. Thus, we model several perspectives of a service composition:

Scenario models. Some of the behavioral requirements of the service are elicited into scenario models using UML sequence diagrams. These scenario models provide details of the interaction between composite service and its partners and also insights on how a certain scenario is realized. This information facilitates the development of the composition and they are also used later on to validate the service implementation.

Resource Model. The concept of resource is central to the structure of REST web service. It represents a piece of information [28]. We represent the static structure of REST web service with resource model which is modeled with a UML class diagram. Each class defines a *resource*. The direction of the associations specify navigability (connectivity) direction between resources, while their role names give the relative URI of resources (addressability). The *collection resources* without the incoming transitions are termed as *root* such that every *resource* defined in the resource model should be reachable via the *root* and the graph formed should be connected (connectivity).

Behavioral Model. The behavioral model represents the dynamic structure of the service using UML state machines. Each state represents the service state and the transition triggers are restricted to the side-effect methods of HTTP protocol, i.e., PUT, POST and DELETE (uniform interface). The statelessness feature of the REST interface is preserved while building stateful REST web service by defining state invariants as boolean predicates over the states of different resources. The state of a resource is given by its representation retrieved by invoking a HTTP GET method on it. We are thus able to define service states as predicates over the resources without maintaining any hidden session or state information (statelessness). The state invariants in the SM are written as Object Constraint Language (OCL) expressions. OCL is commonly used to define constraints in UML models, including state invariants [5]. For modeling a service composition, the models are required to represent method invocations on the partner services. The service invocations to partner services are modeled as effects on the transitions. The composite web service requirements, inferred from the specification document, are added as UML *comments* on the transitions that satisfy them.

Domain Model. The domain model of the composite service is represented with a UML class diagram. It represents interfaces between the composite service and its partner services. The required and provided interface methods between the composite and its partner services are modeled with required and provided interfaces in the domain model, respectively.

Transformation. In order to make the models amenable for simulation and model checking we employ a set of mechanized steps for translating UML-based service specifications into UPPAAL timed automata (UPTA) [23].

The transformation from UML design models to UPTA has been discussed in [27]. It takes as input the resource model, domain model, and behavioral model and generates two artifacts in UPTA: the SUT model specifying the behavior of the service and of its partner services (a generic example is presented in Figure 2) and an *environment model* which simulates the behavior of the service user. Two kinds of environment models are generated automatically: a canonical model which allows to simulate freely all possible behaviors of the SUT and a model used for testing different user scenarios.

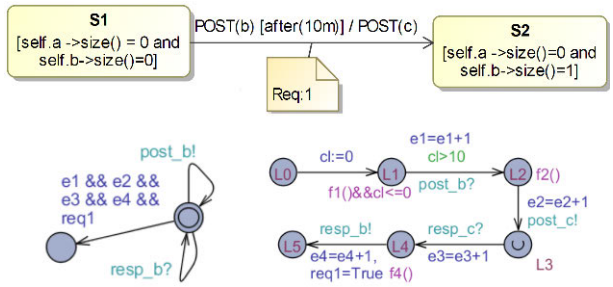


Fig. 2. Example of State Model (top), Corresponding Environment Model (bottom left), and Flattened TA (bottom right)

The transformation of the user scenarios from sequence diagrams to UPTA environment models is applicable to Sequence Diagrams(SD) with a restricted set of elements. The following generic steps are used by the transformation:

- Each SD has may have several lifelines, which are grouped into two groups: SUT and environment. The messages exchanged between the two groups will provide the testing interface.
- For each input message to the SUT group, we define an edge to a new location in UPTA. The edge is labeled by the name of the message and it has associated a sending channel(!).
- For each output message from the SUT group, we define a new edge to a new location with a receiving channel (?).
- For SD fragments (i.e., alt, loop, opt), based on the number of conditions in the fragment, we define several edges from a location and use the conditions as guards on the edges.
- Timing constraint and duration constraint are transformed into location invariants and edge guards in UPTA.
- Tracking variables are added to each scenario trace in UPTA. If a scenario has more than one exit points (alternatives) several variables are added. A tracking variable is an updated tuple *(sd_no = false, sd_no = true)* on the first edge in the scenario trace and respectively on the last edge in the trace.
- UPTA traces stemmed from different SDs are included in one single UPTA environment.

The resulting UPTA environment will have channel synchronizations matching the SUT model obtained in the first transformation.

Figure 3 shows an example of a SD with three lifelines (right) and its transformed environment model (left). Assuming *S1* as the environment, the UPTA environment model contains only emitting/receiving messages to/from *S1*. Response(b) should be received within 5 minutes ($cl < 5m$) and request(j) can be sent before 24 hours ($cl2 < 24$ hrs).These timing constraints are modeled as location invariants and guards in UPTA. For modeling before and after a deadline ($cl < 10$ and $cl >= 10$) in sequence diagram, we used **alt**, which is transformed into two different locations with their corresponding edges (c! and g?) in UPTA. The timing constraints in **alt** are translated as location invariant and edge guard in the model.

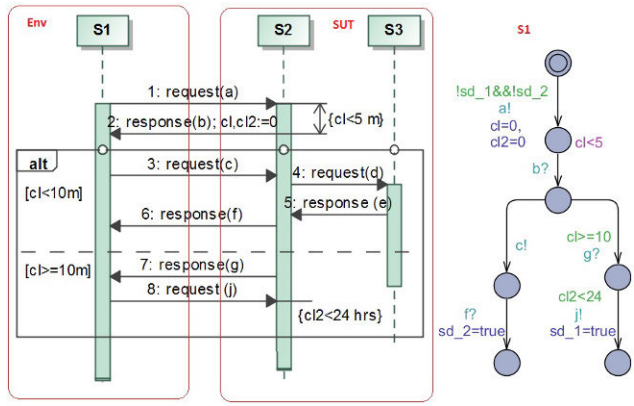


Fig. 3. Example of SD of three services (left) and the UPTA model of S1 as environment (right)

Verification. We use the UPPAAL model-checker [23] to verify basic properties of our design models such as reachability, liveness, and safety. In addition, we check wether the service user scenarios are satisfied. This allows one to eliminate design errors that can be otherwise expensive to detect and correct at later stages of the development cycle. If problems are found, updates are manually fixed in the UML design models.

Test Generation. A skeleton of the composite service is generated automatically in the Django web development framework [16] using our partial code generation tool. The implementation is manually completed by the service developer. In order to validate that the implementation of the composite service is functioning correctly along with its partner services and if the service goals and timed constraints are being fulfilled, we generate tests from the UPTA models and execute them online (on-the-fly) against the implementation.During the test execution we monitor how different test coverage criteria are fulfilled, how the requirements are covered, and whether the user scenarios are validated.

3 Tool Support

Modeling in UML. The design models are modeled using MagicDraw [2]. Static validation of models is done via OCL using the validation engine of Magic Draw. We rely on predefined validation suites for UML contained in MagicDraw for the basic validation of the model. These validation suites contain rules that check that the designed UML model conforms to UML meta-model specifications and prevent the developer from doing basic modeling mistakes.

Code generation. The code-skeleton of the updated service design models of REST composite web service can be generated using our tool presented in [26]. The tool generates code skeleton for design models in Django that is a high level Python web framework [16]. The generated code also has behavioral information such that *pre* and *post* conditions for each method are included and the developer just has to write the implementation of the operations.

UML→UPTA transformation. A Python script is used to automate the transformation.

Test generation. We generate tests using UPPAAL TRON, an extension of UPPAAL for online model-based black-box conformance testing [24]. A test adapter is used by UPPAAL TRON to expose the observable I/O communication between the test environment model and the SUT model. Our adapter implements the communication with the SUT by converting abstract test inputs into HTTP request messages and HTTP response messages into abstract test outputs. UPPAAL TRON generates tests via symbolic execution of the specifications using randomized choice of inputs. Based on the timed sequence of input actions from the simulation, the adapter preforms input actions to Implementation Under Test (IUT) and waits for the response. Output from IUT is monitored and generated as output actions for the simulation. The conformance testing is achieved by comparing outputs of IUT to the behavior of the simulation.

Test coverage information. In order to enable rigorous test coverage in UPPAAL TRON, a second Python script (discussed in more detail in [20]) is used to automatically add *tracking* variables (also referred to as *traps* in the UPPAAL community) for each edge of a given automaton in a UPTA model and a corresponding update of the given variable on the corresponding edge. Whenever the edge is visited during the simulation or execution, the variable is incremented, allowing thus to track which edges have been visited and how many times. This enables one to track coverage level wrt. e.g., edge coverage or edge pair coverage. This script will also be integrated in the final version of the UML→UPTA transformation script. W.r.t scenario-coverage each scenario will have its own tracking variable, changing value when the scenario is considered fulfilled (see for instance variables *Sc1* and *Sc2* in Figure 8 (left)).

4 Case Study

Our case study is a Holiday Booking (HB) composite REST web service that is built on inspiration from the *housetrip.com* service, with the purpose of having a case study similar in complexity to real services. This service is a holiday rental online booking site, where one can search and book an apartment in the destination country.

The user of the service searches for a room in a hotel from the list of available hotels at HB before travel. He books the room (if it is available) and that booking is reserved by HB with the hotel for 24 hours. The user must pay for the booking within 24 hours. If the user does not pay within this time then the booking is canceled. If the booking is paid, then the HB service invokes a credit card

verification service and waits for the payment confirmation. When the payment is confirmed, HB invokes the hotel service to confirm the booking of the room. If the hotel does not respond within 1 day or it does not confirm at all, the booking is canceled and the user is refunded. If the hotel service confirms, then a booking is made with the hotel. The payment is not released to the hotel until the user checks in. When the user checks in, HB releases the money to the hotel and the booking is marked by the hotel as paid. Due to space limitation, we only show some of the models in here while complete details are available at [26].

Requirements. We have inferred functional and temporal requirements from specification document for our case study. In total we specified 4 main requirements with their sub-requirements. Some of these requirements are accompanied by scenario models. For brevity, Table 1 shows only two of these requirements, *Payment* and *Cancel*. The scenario models in Figures 4 and 5 detail how their corresponding user scenarios are fulfilled by the composite service.

Req	Sub-Requirements
2- Payment	2.1 - When user pays for the booking, partner service should be invoked to process the payment 2.2 - If the partner service confirms the payment, the booking should be marked paid. ...
3- Cancel	3.1 - A paid booking can be canceled by the user 3.2 . A canceled booking must be refunded. ...

Table 1. Requirements of Holiday Booking CWS (excerpt)

Design Models. The design of HB composite REST web service is modeled with resource, behavioral and domain models. Due to space reasons only an excerpt of the state machine of HB composite service is shown (Figure 6). Service requirements are traced to the state machine by including them (and their sub-requirements) as comments linked to transitions.

UML→UPTA transformation. The timed automaton corresponding the the HB service from Figure 6 is given in Figure 7. The detailed model and the specifications of the partner web services are available in [26].

Figure 8 shows the two types of environment models produced by the transformation: one modeling the user scenarios in Figures 4 and 5, and a canonical model. Each scenario has associated a tracking variable (e.g., *Sc1*) which helps in performing the verification and monitoring test coverage.

Verification. The verification properties are specialized for our case study and some of them are mentioned below.

Deadlock Freeness. The HB Service, the hotel service and the payment service models are all deadlock free. This means that the composite service never reaches a state that cannot preform a transition (i.e., $A\,\Box\, not\ deadlock$). Note that the following queries are made for complete model and only some of them can be traced in Figure 7.

Reachability. All the locations in the HB service are reachable. This means that the model receives and sends messages to the partner services smoothly and the model is validated for its basic behavior (i.e., $E\,\Diamond\, CompService.r$), where r is

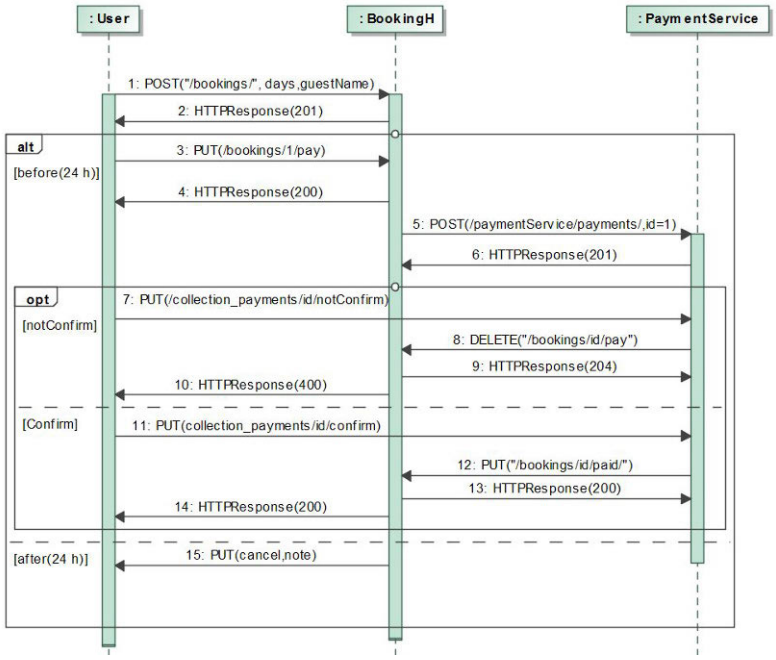


Fig. 4. Scenario Model for User Payment and Invoking Payment Service

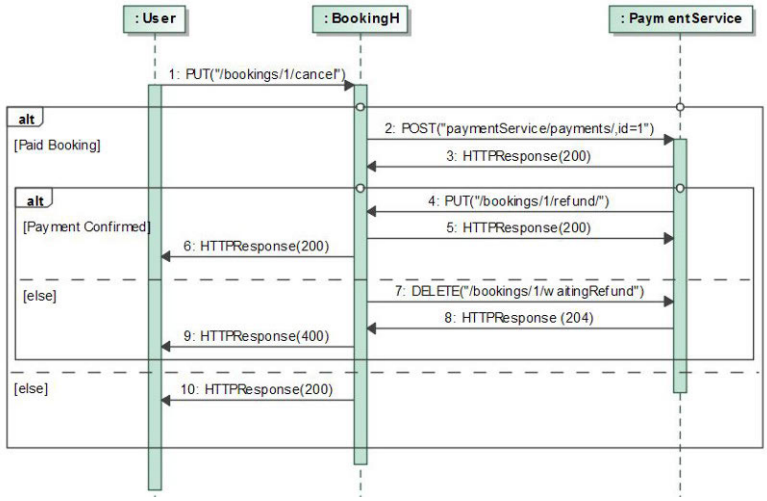


Fig. 5. Scenario Model to Cancel Booking

the last location in the TA model and indicates that all processes for a certain booking is completed.

Safety. Some of the safety properties in our model are: a) Payment should be released iff the user has checked in, i.e., ($E\Box CompService.h2$ imply $CardService.c2$),

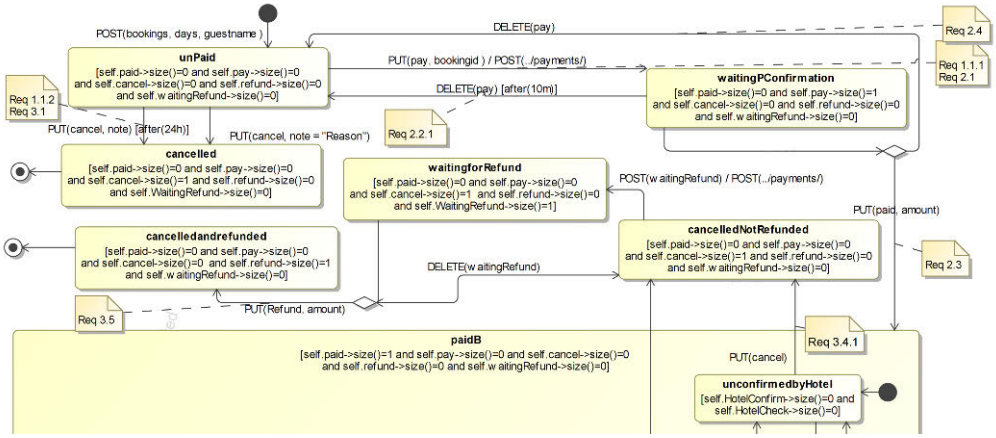


Fig. 6. Excerpt of UML State Machine of Holiday Booking Composite REST Web Service

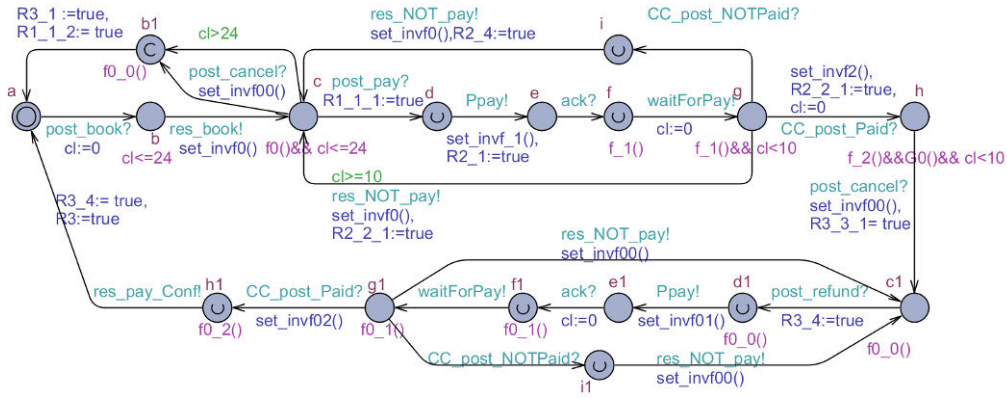


Fig. 7. Excerpt of UPTA model of Holiday Booking Composite REST Web Service

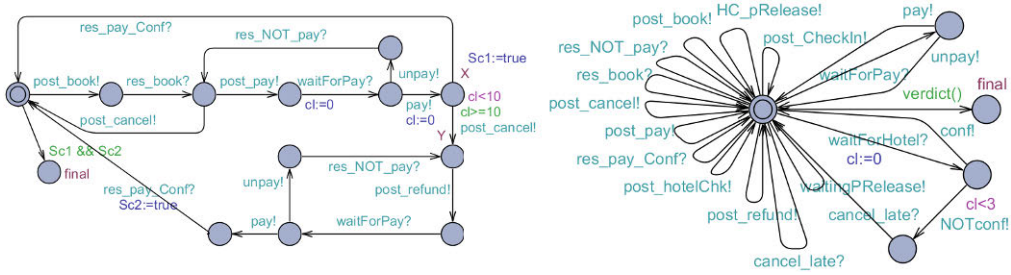


Fig. 8. Excerpt of Scenario-based environment (left) and canonical environment (right)

where *c2* is the location after check-in and *h2* is the location after payment release, b) If the payment is released by the HB service then the Hotel service is paid, i.e., ($E\Box CompService.h2$ imply $HotelService.p$), where p is the location in Hotel service model for hotel payment.
Liveness. Some of the liveness properties in the model are: a) When the payment is not paid within 24 hours, the booking is canceled (i.e., $CompService.c$ and $compService.cl > 24 \rightsquigarrow CompService.b1$), where c indicates waiting for the payment, cl indicates clock of the model and *b1* indicates the booking request is going to cancel due to the delay, b) If the Hotel Service does not confirm within 3 days then the booking is considered not confirmed (i.e., $CompService.o$ and $CompService.cl > 3 \rightsquigarrow CompService.n$), where o is the location for waiting for the hotel response and n is the location for canceling. For the scenario environment, we identified a boolean variable for each scenario. Initially, all variables are false, and at the end of each scenario the corresponding variable will be set to true. The verification rule shows that all scenarios are reachable ($E\Diamond SDEnv.Sc1$ and $SDEnv.Sc2$ and $SDEnv.Sc3$), where SDEnv indicates the environment model, and Sc1, Sc2 and Sc3 are the variables. Timing constraints in scenario environment is verified by checking if the user is waiting for the service payment confirmation more than 10 hours (i.e., in location X), then she can cancel the reservation (i.e., $SDEnv.X$ and $SDEnv.cl > 10 \rightsquigarrow SDEnv.Y$), Y and X are locations.

Testing. The test setup comprises the TRON engine, the test adapter, and the IUT. The IUT is a web service composition of three web services: Holiday Booking, Hotel and Payment Services, whereas the environment model is one of the models in Figure 8. Whenever all the tracking variables monitored by the environment models are *true*, e.g., scenario 1 and 2 are fulfilled or all edges of the SUT model are covered, the environment transitions to the final state. This approach is used as a stopping criterion for testing.

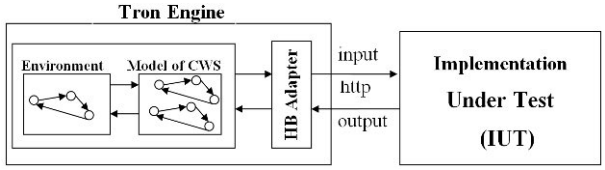


Fig. 9. UPPAAL TRON test setup

5 Evaluation

The UML state machines of the HB composite REST web service had 14 states and 25 transitions. These were translated into an UPTA model with 34 locations and 46 edges. Similarly, the state machines of the Payment service had 3 states and 4 transitions which were translated into an UPTA model with 5 locations

and 6 edges. The Hotel service had 4 states and 5 transitions that were translated into 7 locations and 9 edges. In addition, the environment model created had 4 locations and 13 edges.

Similarly, the two user scenarios discussed in this article (Figures 4 and 5) comprised of 15 and respectively 11 messages which were transformed into the automaton in Figure 8–left with 13 locations and 17 edges.

One issue with using formal tools like UPPAAL for verification and test generation, is the scalability of the approach, due to the state space explosion. In contrast to offline test generation, where the entire state space has to be computed, in online test generation only the symbolic states following the current symbolic states have to computed. This reduces drastically the number of symbolic states making the test generation less prone to space explosion and thus more scalable. For instance, the number of explored symbolic states when generating, with the `verifyta` tool, traces satisfying complete edge coverage (i.e., $e_1 \& \ldots \& e_j \& \ldots \& e_m$, where e_j are tracking variables corresponding to all m edges of the HBS models) was 974. In the contrast, the maximum number of symbolic states reported by TRON during a test session achieving complete edge coverage was 12.

For benchmarking the verification process, we have used the `verifyta` command line utility of UPPAAL for verification of the specified 5 properties. We have used the `memtime` tool to measure the time and memory needed for verification. The result showed in average 2 seconds and 54996 KB of memory being used. Although the memory utilization depends heavily on the symbolic state space, it shows that the current size models leave room for scalability of the approach.

In order to evaluate the efficiency of our approach, we compared the specification coverage with the code coverage yielded by a given test run. Since we had access to the source code of the IUT, we used the *coverage* tool for Python [1] to report the code coverage for each test session. Table 2 lists results of several measurements.

Table 2. Correspondence between code coverage and edge coverage

Run	Edge Coverage	Code Coverage
1	64 %	55%
2	80%	67%
3	100%	78%

Although many of the errors were caused by modeling mistakes, testing revealed some errors in the implementation as well. For instance, in the HB service, there was an error in sending *cancel* request and another error found in the POST header in *refund* request. Also in the Hotel service, the confirmation was sent by the wrong method, so it was rejected by Holiday Booking service. Similar errors were detected by applying Scenario-based environment model.

In order to evaluate the fault detection capabilities of our approach, we have manually created 30 mutated versions of the original HB service program code. Each mutation had one fault seeded in the code, for instance replacing POST with DELETE, removing one line of the source code, change of logical conditions, etc. The faults were always seeded in those parts of the code that is covered when achieving 100% edge coverage of the model. We assumed that the original version

of the composite web service is the correct one, as we were able to run the 100 test sessions in TRON against it. For each mutated version of the composite web service, we set the TRON to execute 100 test sessions against it. When a fault was discovered, the mutant was considered as *killed*. If the mutated statement has been covered by the test runs but no failure was detected, we mark it as *alive*. Out of the 30 mutated programs, 28 mutants were killed and 2 were alive, using the canonical test environment in Figure 8-left. This resulted into a mutation score of 93.3%.

6 Related Work

A large body of work on using model checking techniques for validation and verification of web service compositions has been done and overviews of works can be found in [29] and [7]. Mostly authors have used web service specific specification languages as starting point and converted specifications to models using model checking tools. Then, they performed simulation, verification or test generation via model-checking. Most of these works use the selected model-checking tool only for simulation and verification; only a handful generate abstract tests from the verification conditions. We can distinguish roughly two verification approaches: those that target the PROMELA language [25] which is the input language for the SPIN model-checker [17], and those that target the UPPAAL timed automata as modeling tool [4]. In the following, we will revisit those works which are most similar to ours.

Garcia [14] uses counterexamples to specify and generate test cases in model checking tool. The transitions in BPEL define the test requirements. The transitions are mapped to the model expressed in LTL properties. Fu et al. [13] provide a framework for both bottom-up and top-down approach analyzing web service compositions. In top-down, the conversation of a web service is specified as guarded automaton converted to PROMELA modeled in SPIN model-checker. The bottom-up approach translates BPEL to guarded automaton and used SPIN tool after translating guarded automaton to PROMELA. The synchronization of web service conversations are analyzed in order to verify the compatibility.

Huang et al. [18] present a work that automatically translate OWL-S specification of composite web service into a C-like specification language and PDDL. These can be processed with the BLAST model-checker which can generate positive and negative test cases of a particular formula.

These works focus on BPEL processes and OWL-S which make them dependent on specific execution languages for SOAP based services whereas our work is not dependent on implementation and supports REST architectural style. Besides, they do not support requirement traceability and is not clear how tests are generated and executed. Furthermore, the PROMELA language cannot address real-time properties, due to the limited support for time in PROMELA. Cambronero et al. verify and validate web services choreography by translating a subset of WS-CDL into a network of timed automata using UPPAAL tool[8]. They model the requirements by extending KAOS goal model. The work is sup-

ported by WST tool that provides model transformation of timed composite web services [9]. Diaz et. al also provide a translation from WS-BPEL to UPPAAL timed automata [10]. Time properties are specified in WS-BPEL and translated to UPPAAL. However, requirements are not traced explicitly, while verification and testing are not discussed.

Ibrahim and Al-Ani [19] specify safety and security non-functional properties in BPEL and later formulated into guards in the UPPAAL model. They do not consider neither real-time properties nor test generation. In [15], Nawal and Godart use UPPAAL to check compatibility of web service choreography supporting asynchronous timed communications. They distinguished between full and partial compatibility and full incompatibility of web services. Our work is somewhat similar to their work as we support time critical stateful REST webs service compositions using UPPAAL, however, in addition to verification we use UPPAAL with TRON to validate the implementation of the web services.

Zhang [30] suggest the use of the temporal logic XYZ/ADL language [31] for specifying web server compositions. They transform the specifications into a timed asynchronous communication model (TACM) which are verified in UPPAAL. In[21], uses BPEL as a reference specification and transform them to an Intermediate Format (IF) based on timed automata and then propose an algorithm to generate test cases. Similar to our approach, tests are generated via simulation in a custom tool, where the exploration is guided by test purposes. The time properties are added manually to the IF specification, while we specify them at UML level.

Biswal et al. present a test generation approach using UML activity diagram to define scenarios [6]. Arnold et al. provide a framework that supports automatic test generation from scenarios and also transforms them to test cases that can run on actual IUT [3]. Enoiu et al. presented an approach to generate test suites for PLC software using UPPAAL [11]. Larsen et al. presented an approach in which scenario-based requirements are translated to timed automata, reducing the problem of model consistency and verification effort [22]. These works provide approaches to verify and validate the service specifications by checking the properties of interest using UPPAAL. However, in our work, in addition to model checking the properties we also perform conformance testing of the service composition via online scenario-based testing with the TRON tool and we provide requirement traceability for non-deterministic systems.

7 Conclusion

We have presented a scenario-based approach to verify and validate RESTful composite web services. In our approach, a service can invoke other services and exhibit complex and timed behavior, while still complying with the REST architectural style. We showed how to model the service composition in UML, including time properties. We modeled communicating web services and explicitly define the service invocations and receiving service calls.

We use model checking approach with UPPAAL model-checker to verify and validate our design models w.r.t user scenarios. From the verified specification, we generate tests using an online model-based testing tool. The use of online model-based testing proved beneficial as our system under test exhibits non-deterministic behavior due to concurrency and real-time domain.

With the help of requirements traceability mechanism we traced requirements to UML models and, via the UML→UPTA transformation to timed automata models. Their reachability is verified in UPPAAL and they are used as test goals during test generation. Linking requirements to generated tests allowed us to quickly see which requirements have been validated and which have not. In addition, it allows us to identify from which parts of the specification/implementation the detected error has originated.

We exemplified our approach with a relatively complex case study of a holiday booking web service and we provided preliminary evaluation results.

References

1. Code coverage measurement for Python – coverage, v. 3.6. Online at https://pypi.python.org/pypi/coverage (2013), retrieved: 20.08.2013
2. Nomagic MagicDraw webpage at http://www.nomagic.com/products/magicdraw/ (August 2013)
3. Arnold, D., Corriveau, J.P., Shi, W.: A scenario-driven approach to model-based testing (2010)
4. Behrmann, G., et al.: Uppaal 4.0. In: QEST '06 Proceedings of the 3rd international conference on the Quantitative Evaluation of Systems. pp. 125 – 126. IEEE Computer Society Washington, DC, USA
5. Birgit Demuth, C.W.: Model and Object Verification by Using Dresden OCL. In: Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice,. pp. 81–89 (2009)
6. Biswal, B., Nanda, P., Mohapatra, D.: A novel approach for scenario-based test case generation. In: Information Technology, 2008. ICIT '08. International Conference on. pp. 244–247 (Dec 2008)
7. Bozkurt, M., other: Testing web services: A survey. Department of Computer Science, Kings College London, Tech. Rep. TR-10-01 (2010)
8. Cambronero, M.E., et al.: Validation and verification of web services choreographies by using timed automata. Journal of Logic and Algebraic Programming 80(1), 25–49 (2011)
9. Cambronero, M., et al.: Wst: a tool supporting timed composite web services model transformation. Simulation 88(3), 349–364 (2012)
10. Dıaz, G., et al.: Model checking techniques applied to the design of web services. CLEI Electronic Journal 10(2) (2007)
11. Enoiu, E.P., Sundmark, D., Pettersson, P.: Model-based test suite generation for function block diagrams using the uppaal model checker. In: Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops. pp. 158–167. ICSTW '13, IEEE Computer Society, Washington, DC, USA (2013), http://dx.doi.org/10.1109/ICSTW.2013.27
12. Fielding, R.T.: Architectural styles and the design of network-based software architectures. Ph.D. thesis, University of California (2000)

13. Fu, X., et al.: Synchronizability of conversations among web services. Software Engineering, IEEE Transactions on 31(12), 1042–1055 (2005)
14. García-Fanjul, J., et al.: Generating test cases specifications for BPEL compositions of web services using SPIN. In: International Workshop on Web Services–Modeling and Testing (WS-MaTe 2006). p. 83 (2006)
15. Guermouche, N., Godart, C.: Timed model checking based approach for web services analysis. In: Web Services, 2009. ICWS 2009. IEEE International Conference on. pp. 213–221. IEEE (2009)
16. Holovaty, A., Kaplan-Moss, J.: The definitive guide to Django: Web development done right. Apress (2009)
17. Holzmann, G.J.: The model checker SPIN. Software Engineering, IEEE Transactions on 23(5), 279–295 (1997)
18. Huang, H., et al.: Automated model checking and testing for composite web services. In: Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on. pp. 300–307. IEEE (2005)
19. Ibrahim, N., Al Ani, I.: Beyond functional verification of web services compositions. Journal of Emerging Trends in Computing and Information Sciences 4, Special Issue, 25–30 (2013)
20. Koskinen, M., et al.: Combining Model-based Testing and Continuous Integration. In: Proceddings of the International Conference on Software Engineering Advances (ICSEA 2013). IARIA (October 2013), tO APPEAR
21. Lallali, M., et al.: Automatic timed test case generation for web services composition. In: on Web Services, 2008. ECOWS'08. IEEE Sixth European Conference. pp. 53–62. IEEE (2008)
22. Larsen, K., Li, S., Nielsen, B., Pusinskas, S.: Scenario-based analysis and synthesis of real-time systems using uppaal. In: Design, Automation Test in Europe Conference Exhibition (DATE), 2010. pp. 447–452 (March 2010)
23. Larsen, K.G., et al.: UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer (STTT) 1(1), 134–152 (1997)
24. Larsen, K.G., et al.: UPPAAL Tron user manual. CISS, BRICS, Aalborg University, Aalborg, Denmark (2009)
25. Part, I., Peschke, M.: Design and validation of computer protocols (2003)
26. Rauf, I.: Design and Validation of Stateful Composite RESTful Web Services. Ph.D. thesis (2014)
27. Rauf, I., et al.: An integrated approach for designing and validating rest web service compositions. In: Monfort, V., Krempels, K.H. (eds.) 10th International Conference on Web Information Systems and Technologies. vol. 1, p. 104115. SCITEPRESS Digital Library (2014)
28. Richardson, L., Ruby, S.: RESTful web services. O'Reilly (2008)
29. Rusli, H.M., et al.: Testing Web services composition: a mapping study. Communications of the IBIMA 2007, 34–48 (2011)
30. Zhang, G., et al.: Model checking for asynchronous web service composition based on xyz/adl. In: Web Information Systems and Mining, pp. 428–435. Springer (2011)
31. Zhu, X.Y., Tang, Z.S.: A temporal logic-based software architecture description language xyz/adl. Journal of Software 14(4), 713–720 (2003)

Publication IV

IV

On Mutating UPPAAL Timed Automata to Assess Robustness of Web Services

Siavashi, Faezeh and Truscan, Dragos and Rauf, Irum and Vain, Juri (2016). In *ICSOFT 2016 – Proceedings of the 11th International Joint Conference on Software Technologies*, pages 15–26. SCITEPRESS - Science and Technology Publications

On Mutating UPPAAL Timed Automata to Assess Robustness of Web Services

Faezeh Siavashi[1], Dragos Truscan[1] and Jüri Vain[2]
[1]*Faculty of Science and Engineering, Åbo Akademi University, Vattenborgsvägen 3, 20500, Turku, Finland*
[2]*Department of Computer Science, Tallinn University of Technology, Akadeemia tee 15A, Tallinn, Estonia*

Keywords: Web Service Composition, Specification Mutation, Robustness Testing, Model-based Testing, UPPAAL, TRON.

Abstract: We present a model-based mutation technique for testing the robustness of Web service compositions. Specifications of a Web service composition is modeled by UPPAAL Timed Automata and the conformance between the model and the implementation is validated by online model-based testing with the UPPAAL TRON tool. By applying a set of well-defined mutation operators, we generated model mutations. We validate all generate mutants and exclude the invalid ones. The remaining mutants are used for online robustness testing providing invalid test inputs and revealing vulnerabilities of the implementation under test. We experimented our method on a Booking System web service composition. The results show that from a total of 1346 generated mutants, 393 are found suitable for online model-based testing. After running the tests, 40 of the mutants revealed 3 new errors in the implementation. The experiment shows that our approach of mutating specifications is effective in detecting errors that were not revealing in the conventional conformance testing methods.

1 INTRODUCTION

Recently, the popularity of web services has increased in the industry. Web services are software applications that support machine-to-machine interactions over the Internet. They are accessible via ubiquitous protocols while expressing a well-defined interface. This advantage opens the door to new business opportunities by making it easy to communicate with partner services and by covering a wider range of users. Web Service Composition (WSC) is the combination of different services to satisfy a new service. Examples of using WSC can be seen in many web applications that enhance their services by using utilities that are offered by famous companies such as Google, Amazon, and Facebook (Sheng et al., 2014).

One principle characteristics of a WSC is its distributed resources, where other services or client web applications access to information by message protocols. This kind of systems should be robust against erroneous inputs. In this context, testing WSCs plays an important role. Not only the expected behavior of the implementation under test (IUT) should be tested, but also the IUT should not contain any unexpected behavior. The functionality of the system can be checked by running test cases derived from the specification while finding unexpected behaviors of the system can be done by *robustness testing*, which executes invalid inputs and detects the vulnerabilities or unexpected behavior of the IUT.

Defining test inputs by modeling the specifications is preferred over the manually written test scripts since the machine can verify the correctness of the models and automatically generate the test inputs. Moreover, it supports more extensive and systematically constructed sets of test cases.

One way to create invalid test inputs is using *mutation testing*, where a set of well-defined mutation operators systematically create syntactic changes to the specifications and produce mutants. This concept was primarily applied for mutating the source code of a system, however, it has also been applied to different modeling languages as well (Budd and Gopal, 1985). Mutants generate invalid scenarios as test cases, which are executed against the IUT. If the IUT respects the mutation without raising an exception, it means that its behavior is inconsistent with its specification (i.e, the IUT accepts an unspecified sequence of inputs).

In this paper, we propose an approach for robustness testing of WSCs using UPPAAL Timed Automata (UTA). The conformance between the model and the

Siavashi, F., Truscan, D. and Vain, J.
On Mutating UPPAAL Timed Automata to Assess Robustness of Web Services.
DOI: 10.5220/0005970800150026
In *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 1: ICSOFT-EA*, pages 15-26
ISBN: 978-989-758-194-6

IUT is first checked via UPPAAL TRON, an online testing tool which supports both test generation and test execution. In online testing, only one test input is generated and executed on the IUT at a time, and based on the test output the next test input will be selected.

As a first contribution, we introduce a testing method, which derives mutants from the specification and executes them via online testing. We use a selection of the mutation operators that are defined in the literature and slightly change them to generate mutants that are suitable for our work.

As a second contribution, in our methodology, we add verification properties to mutated model segments to ensure reachability of the mutated elements at runtime. This step is supported by a mutation generator tool, which implements selected mutation operators and performs early verification of each mutant. If a mutant does not pass the verification properties, it cannot be used for online testing, hence, we eliminate them. Furthermore, to ensure that the mutated part will be executed during the testing process, we monitor whether the mutated elements are reached during test execution.

As a third contribution, we empirically evaluate which existing mutation operators for UPPAAL timed automata are applicable to online testing. We define two formulas to measure the efficiency of mutation operators as well as their rates of fault detection.

The remainder of this paper is organized as follows: In Section 2, we briefly review the background studies. We present the steps of our methodology on specification mutation testing in Section 3 and selection criteria for valid mutants. The experiment is presented in Section 4. The results are discussed and possible improvements are suggested in Section 6, and the threats of validity of the proposed method are discussed in Section 7. We review the literature for related work in Section 8. Finally, we conclude our study and present future work in Section 9.

2 Background

We first review UPPAAL tool set, and introduce the conformance testing with UPPAAL TRON and the concept of specification mutation testing.

2.1 UPPAAL Timed Automata (UTA)

UPPAAL is a model-checker tool for modeling, simulation, and verification of real-time systems using an extended version of timed automata called UPPAAL timed automata (UTA) (Beharmann et al., 2004). A

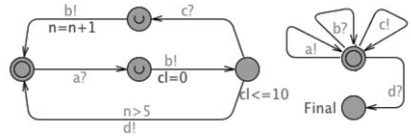


Figure 1: Example of an UTA model.

timed automaton is a finite state machine with locations, actions, and clocks.

In UPPAAL, a system is designed as a network of several such timed-automata called processes working in parallel. A process can be executed individually or in sync with another process. Synchronization of two processes is possible by using input/output actions (denoted as "!" for emitting and "?" for receiving synchronizations, respectively). The processes consist of locations and edges. The state of the system can be shown by the locations of all processes, their clock values, and their variable values. The edges between locations represent state transitions including clock resets. UPPAAL is extended further with global and local to some process variables that can be of type *integer*, *boolean*, and *clock*.

Transitions can be constrained by predicates (over the clocks or variables) known as *guards*, which defines when the corresponding edge is enabled. The state transitions are specified on edges as variable updates. A location can be restricted over the clock invariants, which specify how long the system can stay in that location. If there is more than one enabled edge at a time, then one of them will be randomly selected. This means that UPPAAL supports non-deterministic modeling, which gives more freedom to represent behaviors, especially in systems with random discrete events (Hessel et al., 2008).

An example of a UTA model is shown in Figure 1. The model consists of two automata modeling the behavior of a system under test and of its environment. The communication between the system and its environment is modeled using channel synchronizations and shared variables.

The UPPAAL model-checker uses a simplified version of TCTL (Alur et al., 1990), which enables to exhaustively verify the models w.r.t their specifications. The query language consists of state formulae and path formulae. State formulae (φ) is an expression that describes the properties of an individual state while path formulae can be used to specify which properties (like reachability, safety, and liveness) hold over a given path (Beharmann et al., 2004).

If there is a state in the model that has no enabled outgoing transitions, then the model is said to be in a deadlock. $A\ \Box$ *not deadlock* query, can be used to verify that for all paths in the model, there is no

deadlock state.

The safety property checks that "something bad will never happen". In UPPAAL it can be expressed in the form $A \square \varphi$ (φ should be true in all reachable states) and $E \square \varphi$ (there should exist a maximal path such that φ is always true).

The liveness property determines that "something will eventually happen" and it is shown by $A \Diamond \varphi$ (φ is eventually satisfied) and $\varphi \rightsquigarrow \phi$ (whenever φ is satisfied, then eventually ϕ will be satisfied).

Reachability properties validate the basic behavior of the model by checking whether a certain property is possible in the model with the given paths. The reachability can be expressed in the form of $E \Diamond \varphi$ (there is a path from the initial state, such that φ is eventually satisfied along that path).

2.2 Online Model-based Testing

There are two distinct approaches in testing: offline and online testing. In offline testing, the complete test scenarios and test oracle are created before the test execution, whereas online testing is a combination of test generation and execution: only one test input at a time is generated and executed and the next test input depends on the current test output (Larsen et al., 2005b). This continues until the test termination criteria are satisfied or an error occurs. Usually, the test stimulus is selected randomly from the enabled test inputs. In online testing, the state-explosion problem is reduced because only a portion of the state space is needed to be calculated and stored at each time. Also, the non-determinism of systems can be simulated on-the-fly by random selection of the tests.

In this study, we use the online Model-Based Testing (MBT) UPPAAL TRON, which is an input/output conformance testing tool for testing real-time systems based on the *rtioco* conformance relation (Larsen et al., 2005a). An UTA model typically consists of two partitions: a system partition and an environment partition. The abstract test inputs generated from the environment are translated into executable test inputs by using an *adapter*, which is an interface between TRON and the IUT. The outputs of the IUT also translated to model-level test outputs. Thus, the I/O conformance of the model and of the IUT is observed by TRON.

The result of online testing with TRON can be *passed*, *failed* or *inconclusive*. An inconclusive test result means that the environment model cannot be updated since the IUT output is unexpected or it has a delay in providing test output.

2.3 Specification Mutation Analysis

Specification mutation analysis is used to design tests to evaluate the correctness and consistency of the specification and the program (Budd and Gopal, 1985). When the mutation analysis is applied to the specification a set of *mutation operators* create slightly altered versions (mutants) of the specification. The tests will be generated from the mutated specification and used to assess whether the IUT is accepting the faulty tests.

In the literature (Belli et al., 2016) the following types of mutants are defined:

Killed: A mutant is said to be *killed* if tests generated from it fail against the implementation, under the precondition that the tests generated from the original model have passed.

Alive: A mutant is called *alive* if the IUT passes all test cases generated by the mutant. Alive mutants can be divided into two types:

Equivalent: An alive mutant is semantically equivalent if it manifests the same behavior as the original model, whereas they are syntactically different.

Non-equivalent: An alive mutant is known as *non-equivalent* if it does not have the same behavior as the original model, however, the differences cannot be detected during testing. These mutants indicate that the implementation is too permissive and is not able to detect the invalid inputs.

Our goal of using mutation for testing is to find the non-equivalent alive mutants since they show that there might be some inconsistencies between the specification and the implementation. Differing between non-equivalent alive mutants from equivalent mutants is done manually.

3 METHODOLOGY

An overview of our method is given in Figure 2. It is divided into five phases.

Design and Conformance Testing is based on our previous work on design and validation of WSCs (Rauf et al., 2014), where we presented an approach to design web services and their behavioral interfaces in UML. We transformed the design models from UML to UTA for verification and testing the implementation of a WSC.

The participating web services and the user behavior are modeled as distinct timed automata. The user behavior supports non-deterministic choices, as well as timing criteria.

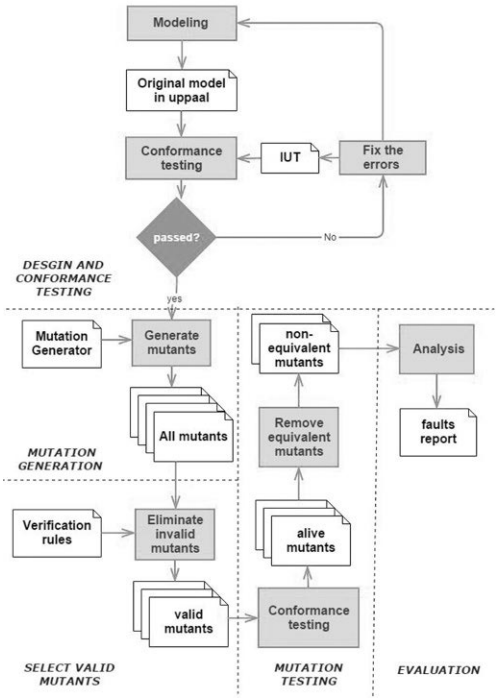


Figure 2: Our approach of Specification Mutation Testing.

The model is verified according to the criteria and timing constraints that are given in the requirements of the WSC. The verification is done using TCTL. For instance, we ensure that the model is deadlock-free and all states of the system are reachable meaning that the model can reach all test goals. These verification rules ensure that the model is usable for online testing.

With TRON, an online testing session is established and the conformance of the implementation is checked. External errors in IUT or in the model are fixed.

Mutation Generation: Mutation operators for TA have been formally defined and presented by two studies in the literature (Aboutrab et al., 2012; Aichering et al., 2013) and are shown in Table 1.

By summarizing Table 1, the following mutations can be applied to the different elements of TA.

- *Guard*: A guard over clock variables can be mutated in three ways: by widening, restricting, or shifting the time value. If the guard contains other variables than the clock variables, it can be mutated by negating the predicate.
- *Invariant*: An invariant can be changed by shifting it to a greater or smaller value. E.g., add/subtract value 1 to/from the value of the invariant.
- *Action*: Name of I/O actions can be changed to other defined actions. Also, changing their source and target locations will manipulate the behavior of the model and so can be used as a mutant.
- *Location*: A location can be made a sink location, which means that it accepts all other actions. It simulates a trap condition, where all actions in the process are accepted in the same location. Removing a location and adding a new location are other mutations that can be applied in TA.

Table 1: Mutation operators of timed-automata.

(Aboutrab et al., 2012)	(Aichering et al., 2013)
Restricting Timing Constraints (RTC)	Change guard
Widening Timing Constraints (WTC)	
Shifting Timing constraints (STC)	
-	Change invariant
Resetting a Clock (RC)	Invert reset
Not-Resetting a Clock (NRC)	
Exchanging Input Actions (EIA)	Change action
Exchanging Output Action (EOA)	
Transferring Destination Locations (TDL)	Change Target
-	Change source
-	Negate guard
-	Sink location

We have restrict some of the operators in such a way that they are suitable for online testing with TRON. As we mentioned earlier, the IUT and its environment (user, or other systems) are specified in separate automata and they communicate by synchronization of input/output transitions (actions). All transitions between the IUT and its environment are observable by TRON. Based on the type of the input or output, TRON controls which action can be executed at a time. The mutation operators for transitions without synchronizations (or internal transitions) will not be observed by TRON. Therefore, we restrict the mutation operators to only be applied to observable synchronizations.

Additionally, we adapt the mutation operators to be used for testing web services. For instance, for each HTTP request message to a web service, we have a corresponding HTTP response message and they are modeled as a pair of input/output actions. The requests are defined as input actions coming from the user (or the environment). One mutation option would be to change the name of the input actions, which mutates the sequence of the HTTP request messages. However, defining mutation for the HTTP response messages (i.e, output actions) cannot help in mutation analysis since the IUT generates them and we can only observe them. For instance, for a booking request, the WSC either accepts or rejects it and both of these responses cannot be mutated in the model-

level. Therefore, we limit the mutation operators to change the name of input actions only.

Finally, we do not change the direction of the synchronizations (i.e, "?" to "!") since, in our modeling approach, the requests from the users are modeled as input actions ("?"). Changing the inputs into output actions means that the requests should be changed into responses and it would not allow test generation at all.

Below we present a list of operators that we selected from Table 1 for our methodology.

1. **Change Name of Input Action (CNI)** replaces the name of an input action (denoted by "?") with the name of other actions. Thus, the expected sequence of the inputs to the implementation will be different.
2. **Change Target (CT)** changes the target of an action to other location. This operator can break the flow of test inputs and violate the state of the IUT. Both input and output actions can be mutated by this operator.
3. **Change Source (CS)** changes the source location of an action to other locations. Similar to CT, this operator gives a different I/O sequence.
4. **Change Guard (CG)** changes the clock constants in guards by a random value. It is effective for mutating the condition of enabling an action.
5. **Negate Guard (NG)** negates guards, which may result in omitting some paths of the test model.
6. **Change Invariant (CI)** shifts the values of invariant conditions to a different range, extending or restricting the constraints of the model. It can cause actions fire earlier (or later) that the expected time.
7. **Invert Reset (IR)** deletes the resetting of the clock and moves it to one action before or after. It means that the resetting is flipped one clock earlier or later.

Figure 3 shows the generated mutants of a model and sample mutants using the above operators. In our approach, we only apply first order mutation. That is, a mutated model contains only one mutated segment based on a single operator.

Select Valid Mutants: In our approach, we enforce that every time a mutated model is generated, we create a corresponding reachability rule to check whether it is a valid mutant for online testing or not.

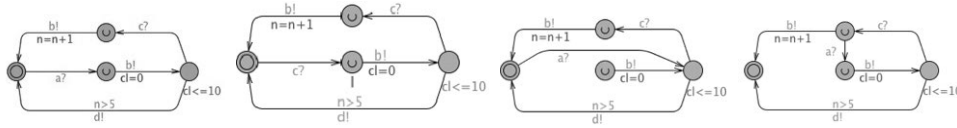
In UPPAAL, the reachability property is defined for locations, thus, when an action is mutated, we define the reachability property for the target location of that action. For instance, in Figure 3(b), the input action $a?$ is mutated into $c?$, hence, the reachability for this mutation should be defined for its target location (i.e, l). For example, in Figure 3(b), we have $E \diamond l$, which verifies that the mutation can be executed. An alternative to the reachability rule would be to define a *trap variable* (Gargantini and Heitmeyer, 1999) and set its initial value to false. For the mutated action, then, the variable will be updated to true, and so the reachability can be achieved by checking if the variable eventually will be set to true ($E \diamond trap == true$). One can use trap variables to ensure that the mutation part of the model will be reached during the test execution as well. In the case that the minimum repetitive execution of mutation is needed the boolean trap variable should be replaced by an integer counter variable *count* and the reachability condition with $E \diamond count >= const$. Those models that pass the verification process are considered as *valid mutants* and can be executed against the IUT.

Having verification rules offers two main advantages. First, it reduces the number of mutants used for testing by eliminating false negatives which cause semantic and syntactic errors. Secondly, it avoids having traps in the model, which may increase the size of the state space.

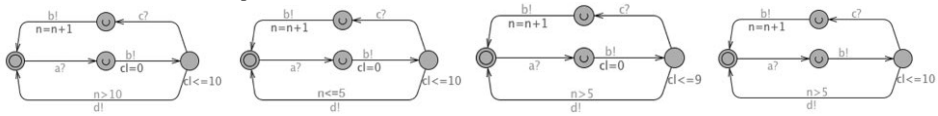
Mutation Testing: Each valid mutant model is executed in a testing session with UPPAAL TRON. The verdict of an online testing session with TRON can be *passed*, *failed*, or *inconclusive*. In TRON, an inconclusive verdict indicates that either the observed output from the IUT is not valid, or there is an unacceptable delay in sending inputs. We consider that the mutants that generate inconclusive test cases, exhibit different behavior than the original model and thus they are considered as *killed*. If the IUT passes the test, then two different scenarios are possible: either the mutant is an equivalent model to the original one (i.e, equivalent mutant), or not equivalent, but there is a defect in the implementation that allows mutated inputs (i.e, non-equivalent mutant). We defer automatic equivalence detection for future work. When executing the mutants we assume implicitly that these test runs are exhaustive w.r.t. the mutation, i.e. all mutations injected are also covered by these test runs.

Evaluation: The last phase of our methodology is to evaluate the result by reasoning about the unexpected behaviors that the IUT shows during testing. The non-equivalent mutants generate different invalid test inputs, thus, these test inputs are manually evaluated to find the correlations between them and the actual faulty behaviors.

Tool Support: We implemented the selected mutation operators as a tool in order to generate the mutants automatically. The tool uses UPPAAL TA XML



(a) The original model (b) CNI: Change the name of input action (c) CT: Change Target (d) CS: Change Source



(e) CG: Change Guard (f) NG: Negate Guard (g) CI: Change Invariant (h) IR: Invert Reset

Figure 3: A model with examples of mutants generated by the selected mutation operators.

format as input. From a given model, the tool generates mutants based with the selected mutation operators. In addition, it adds reachability and deadlock-freeness rules to the mutants and verifies them with the *verifyta* tool, which is a command-line verification tool for UPPAAL models.

4 EXPERIMENT

We exemplify our approach using the case study presented in (Rauf et al., 2014). In this section, first, we review the case study, and then we apply the specification mutation method.

4.1 Case Study

For evaluation, we used a WSC that is implemented in REpresentational State Transfer (REST) (Richardson and Ruby, 2008) architectural style. The composition of web services is based on a central service which orchestrates other services. This service synchronizes the execution of different methods on the web services participating in the composition and satisfies the specifications. The central web service (i.e, the composition service) can invoke other services while exhibits timed behaviors in a RESTful architecture.

The WSC offers a Hotel Booking System (HBS), including a Card service, a Hotel service, and a Booking service. This case study is specified, implemented and verified in our previous work in details in (Rauf et al., 2014). The Card service deals with payments and refunds for booking requests, whereas the Hotel service keeps track of the details of booking records such as name, the number of days and type of room, also giving access to the hotel manager for accepting or declining the booking requests. The Booking service is responsible for communications with customers, the Hotel, and the Card services. From the specification of HBS, we define the following scenarios:

Booking: A customer can search for a room in a hotel by accessing the booking service. He books the room (if it is available) and that booking is reserved by the Booking service for 24 hours.

Payment: If the user does not pay within 24 hours then the booking will be automatically canceled. If the booking is paid, then the Booking service invokes Card service and waits for the payment confirmation.

Hotel Confirmation: When the payment is confirmed, Booking service invokes the Hotel service to confirm the booking of the room. The Hotel service can confirm and assign a room for the customer, or it can reject the request.

Refund: If the Hotel service does not respond within 1 day, rejects the request, or does not confirm at all, the booking is canceled and the user is refunded.

Check-in: If the Hotel service confirms, then a booking is made with the hotel. The user now can check in to the hotel.

Hotel Payment Release: The payment is not released to the hotel until the user checks in. When the user checks in, the Booking service releases the money to the hotel and the booking is marked by the hotel as paid.

4.2 Model

From the above descriptions, we have specified the system as a UTA model which consists of four automata: three for the web services and one for the environment. Figure 4 shows the models of the case study and the interactions between the services and the environment. In this experiment, we mutate only the Booking service that is larger and handles the communications among other services and users. The Booking service model consists of 33 locations, 39 actions, 4 guards, and 4 clock invariants.

Table 2: Result of mutation testing.

Name	Generated	Valid	Killed	Alive
CNI	180	28	24	3
CT	567	314	242	72
CS	567	38	6	32
CG	12	6	6	0
NG	4	1	1	0
CI	12	4	4	0
IR	4	2	2	0
Total	1346	393	285	107

After verifying the model, we developed an adapter for translating the model-level inputs into HTTP requests which are sent to the IUT, and then, we generated tests using UPPAAL TRON. The use of online MBT proved beneficial as our implementation under test exhibits non-deterministic behavior. For instance, in the scenario of Hotel Confirmation, there are three possible cases from the hotel: confirmation, rejection, or no response. Any of these choices are given the same chance to be executed with non-deterministic modeling.

4.3 Generating Valid Mutants

Table 2 shows the numbers of mutants generated from each mutation operators. Since the Booking service represents the composition of different web services as well as communicating to the user, it is a good candidate to be mutated. The mutation generator provided 1346 mutants, from which 393 of mutants were valid (i.e, passed the verification rules). The total time for generation and validation of all mutants took 258 seconds in a 4 cores machine running the Ubuntu 14.4 Server operating system. As the numbers show, having verification in the early stage of testing would help in removing non-relevant mutants and hence the total time of the test execution will be considerably reduced.

As it can be seen in Table 2, a majority of 314 valid mutants are generated by the CT operator, in contrast with 38 valid mutants provided by the CS and 28 from the CNI. The other mutation operators have a small share of valid mutants.

4.4 Mutation Testing

We set the test session for executing tests 3 minutes for each mutant model covering all actions in the model ensuring that the mutated element was also covered at runtime. It roughly took 7 hours to complete running all valid mutants. The time was sufficient for covering all valid mutations of interest. Therefore, it was postulated that if no failure is detected during this time, and the test is passed, then the mutant is alive.

5 RESULTS

We check whether the alive mutants were able to show any fault in the behavior of the web services and which of the mutation operators generates more effective mutations in online testing.

We also present two formulas for the efficiency of mutation operators showing how many of the alive mutants address faults. We need, therefore, to separate the equivalent mutants from the alive mutants. The analysis is based on the reasoning why the mutated inputs could not be detected by the IUT.

Automatically detecting all equivalent mutants is an impossible task since they are undecidable (i.e, there is no possible solution to confirm that a mutant has equivalent behavior to its original program). Although there are several approaches to the detection of equivalent mutants, it still requires human effort. We manually distinguished the equivalent mutants by checking whether the mutants change the sequence of the test scenarios and how it affects the functionality of the IUT. It is done by checking if all the test scenarios can be covered by the mutants and where is the location of the mutation in the model.

It is worth noting that not all of the non-equivalent mutants cause violations in the functionality of the IUT. For example, in the model of Booking service, changing the target location of the action $post\_hotelChk$ to the location a does not cause an invalid test scenario. Despite the fact that such mutant does not cover all test scenarios, it will pass the test. The reasoning behind this is that from the initial location, a, any booking requests will be considered as a new booking request and will be a new booking record. Therefore, such non-equivalent mutants do not violate the functionality of the Booking service.

Since in the robustness testing the goal is to detect unexpected behaviors of the IUT, having more alive mutants indicates that the corresponding operators are more effective. Hence, we define the following formulas for analyzing the mutation operators:

Mutation Efficiency: For each mutation operator, we calculate how many mutants are alive. We calculate the efficiency of each mutation operator in generating alive mutants:

$$ME_i = \frac{A_i}{V_i}\,, \tag{1}$$

where A is the number of alive mutants, V is the number of valid mutants of operation i.

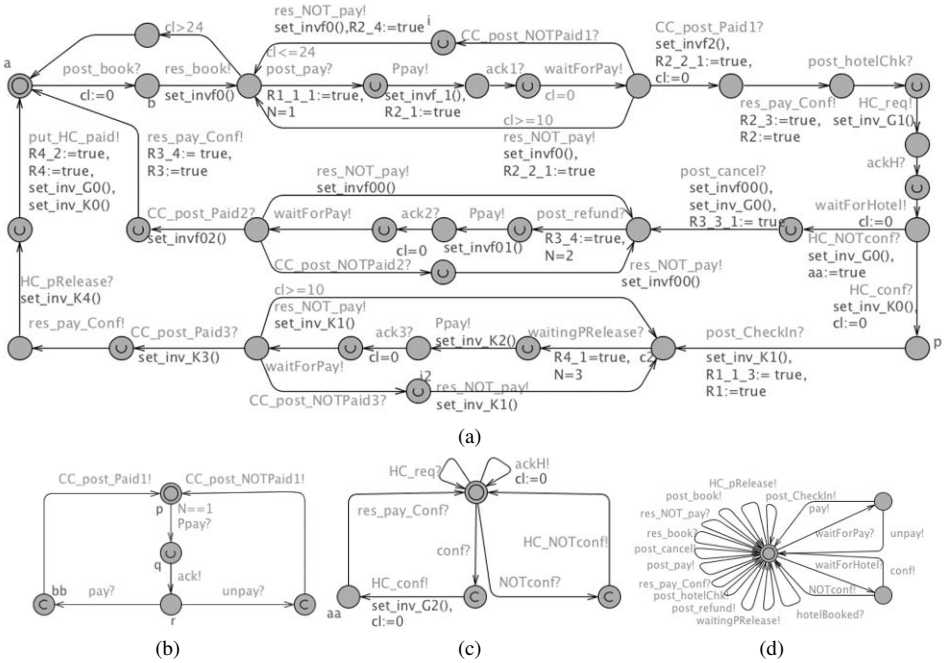


Figure 4: The model of HBS: (a) Booking service, (b) Card service, (c) Hotel service and (d) Environment.

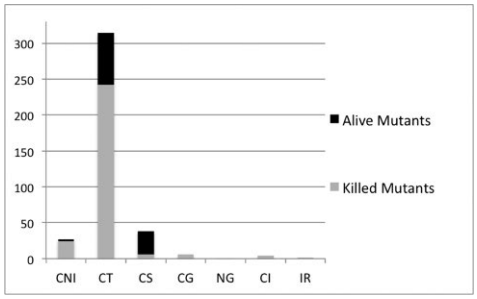


Figure 5: The proportion of alive and killed mutants for each mutation operator.

Mutation Fault Detection: After analyzing alive mutants and removing the equivalent mutants, we check which non-equivalent mutant corresponds to a fault. For each mutant that was able to show a fault in the IUT, we score the corresponding operator. For each operator, we measure the mutation fault detection with following formula:

$$MFD_i = \frac{NE_i}{T_i - E_i}\,, \qquad (2)$$

where NE is the number of non-equivalent mutants that reveal hidden faults, T is the number of total mutants and E is the number of equivalent mutants.

The primary result shows that the total alive mutations belong to three operators: CT, CS, and CNI with 72, 32 and 3 mutants respectively. The CT operator is a good candidate for mutation testing since it generates the highest number of alive mutants. It can be debated that changing the order of the test inputs may cause changing the state of the IUT and hence, the IUT may reach to an unknown state (i.e, unexpected condition). Moreover, as it can be observed from Figure 5 that the proportions for the total number of alive and killed mutants for each individual operator show that the mutation operators CT and CNI were the most suitable operators for our case study.

By using Formula 1, we calculated the efficiency of the operators CT, CS and CNI, which result in 22.9%, 84.2% and 10.7%, respectively (Table 3). This means that the CS operator is more effective in successfully generating alive mutations.

Table 3: Mutation efficiency and Mutation fault detection of the mutation operators.

	mutation efficiency	**fault detection**
CT	22.9%	62.5%
CS	84.2%	8,3%
CNI	10.7%	0

Analysis of the result shows that there are some faults in the implementation of the case study that

were not detected during the conformance testing. We found the following problems in the behavior of the implementation:

- Ten different mutants revealed the same fault in the *Hotel Confirmation* scenario. For example, one faulty scenario is: from a single booking, it is possible to send the confirmation request more than once. Nine of these mutants were generated by the CT operator and one by the CS operator.
- Seventeen mutants showed that there is a fault in the *payment* scenario of the IUT. After payment confirmation from Card service, a new payment for the same booking can be made. Also, for a single booking, there could be several payments. Seven of these mutants are generated by the CT operator and the rest 10 are from the CS operator.
- Thirteen mutants made faulty changes in the *refund* scenario, which could not be detected in the original testing. Four of them belong to the CT and 9 are from the CS.

From 40 different mutants, 3 hidden faults are revealed in the implementation.

Half of the mutants that revealed faults were from the CT operator and half were from the CS. We used Formula 2 to measure fault detection capability of each mutation operator. The result of the calculation is shown in Table 3 as well, showing that CT gets the best score in revealing faults.

Table 3 illustrates information on how the mutation operators are able to show some faults in the case study. The result in the first column shows how many alive mutants have remained after the mutation testing without having further information about the equivalent mutants.

Here, it seems that CS is a better operator than the others. However, after removing the equivalent mutants and calculating the fault detection ability of each operator, CT provides a better percentage. The second column in the table shows the result. All of the alive mutants generated by CNI were found equivalent and hence CNI is ranked 0 in fault detection.

6 DISCUSSION

Some improvements can reduce the test execution time while increasing the probability of finding faults. For instance, both CS and CT were able to reveal all three faults and since both of them have generated large numbers of mutants, selecting one of them can considerably reduce test generation and execution times. The result of mutation testing indicates that an intelligent choice of the mutation operators can attain high mutation efficiency scores while reducing the time of testing.

Another improvement could be done in the process of fault detection. Redundant work is done on detecting the same faults. This extra effort can be reduced by categorizing the alive mutants in such a way that all mutations of a certain location or action in the model will be in a category. As soon as any of the mutants in a category detects a fault, then the rest of the mutants on that group can be eliminated from the fault detection analysis. The idea behind this is that the locations and actions in a model represent actual states of the system under test and if there is a state which contains a fault, then any mutant from that state may be able to reveal that fault. However, more experiments are needed to show the correctness of this mutation reduction technique.

More extensive studies are needed in order to investigate how the specification mutation can be applied in larger case studies preferably industrial-sized web services. Besides, more experiment on larger scales would be helpful in finding whether there is any correlation between certain mutation operators and the real faults in design and implementation of web services.

It should be noted that the presented approach for robustness testing does not specifically designed for composite of web services, but any individual service can also be tested. We selected the WSC since it includes more communications and timing behaviors.

The main downside of model-based mutation testing comes from MBT: the process of design models from the specification, verifying them and writing the test adapter (to translate model-level test inputs into acceptable test script for the IUT and vice versa) is time consuming. We have reduced the design and verification time by reusing the same models from the previous research. The mutation testing does not add any overhead into MBT. The mutation generator tool automatically generates correct and valid mutations and thus, it reduces the mutant generation time.

7 THREATS TO VALIDITY

There are three main threats related to our study. One is related to the mutation operators. Despite the fact that we have followed the systematically and formally defined mutation operators and implemented them in our study, there might be some more effective mutation operators or combinations of operators that we have missed. We argue that the current number of mutation operators provides a large number of mu-

tants which can provide faulty test inputs which are close to the accepted inputs.

Another threat is that although the test model is designed and validated very carefully and the IUT is well-tested, there might be some mistakes in designing the test model. However, the probability of such mistakes is low since we have applied conformance testing and fixed the bugs prior to mutation analysis.

Judgmental errors may have happened during the classification between equivalent and non-equivalent mutants. For comparing the mutation models and the original one, we checked the alive mutants and applied formal verification rules.

8 RELATED WORK

A comprehensive analysis is done on all available mutation testing method presenting the current state of the art in this field and the open challenges (Jia and Harman, 2011) .

Lee and Offutt (Lee and Offutt, 2001) introduced an Interaction specification Model which formalize the interactions among Web components. They defined a set of mutation operators for XML data model in order to mutate the inputs of the Web components. Li and Miller (Li et al., 2009) presented mutation testing methods using XML schema to create invalid inputs. Mutation testing is extended to XML-based specification languages for Web services. Lee et al. presented an ontology based mutation operators on OWL-S, which is an XML-based language for specifying semantics on Web services(Lee et al., 2008). They mutate semantics of the specifications of their case study such as data mutation, condition mutation, etc. Wang and Huang presented a mutation testing approach based on OWL-S to validate the requirements of Web services (Wang and Huang, 2008). Also, Dominguez et al. presented a mutation generator tool for WS-BPEL.

We discuss those that are similar to our approach. Work has been done on using model checking techniques for validation and verification of WSC. There are two studies that review the literature on testing Web services (Rusli et al., 2011), (Bozkurt and other, 2010). Starting from specification languages for modeling Web services, researchers perform simulation, verification and test generation using model checking tools. Most of the works use model checking for specification and verification and only a group of them use the models for the test generation as well. We discuss those that are similar to our approach. Using TA models for mutation testing has been mostly studied on a real-time and embedded system. In (Aboutrab et al., 2012) and (Aichering et al., 2013) mutation operators for TA are presented. Aboutrab et al. proposed a set of mutation operators for timed automata to empirically compare priority-based testing with other testing approaches (Aboutrab et al., 2012). However, in their approach, the generation of mutations is done manually.

Aichernig at al. presented model-based mutation testing real-time system using UPPAAL (Aichering et al., 2013). The mutation operators that are defined in their work more detailed and some of them are implemented as a mutation on bounded model-checking and incremental SMT solving. They showed that using mutations for timed automata has potential on debugging and revealing the unexpected behavior of the IUT.

We applied/modified the mutation operators of TA presented by these studies for testing the robustness of WSC. Similar to (Aichering et al., 2013), we applied mutations on non-deterministic models, however, in their work, they use only the UTA model of the IUT and do not consider the environment. In our approach, however, each mutant is a closed model communicating with its environment as well as other systems. We check deadlockfreeness and reachability in order to reduce the number of invalid mutants. Also, we use different verification and test generation processes.

There are some works that target UTA as the specification language for Web services. In most of the works, the authors transformed the specification that is defined in their selected languages into UTA and then they investigated their research. For instance, in (Rauf et al., 2014), the specification of a WSC is defined initially in the form of UML and then transformed into UTA for an online testing purpose. In (Cambronero et al., 2011), Cambronero et al. verify web services by the UPPAAL tool for validation and verification of their described system that is transformed from WS-CDL into a network of TA. In (Dıaz et al., 2007), Diaz et al. also provide a translation from WS-BPEL to UTA. Time properties are specified in WS-BPEL and translated to UTA. However, requirements are not traced explicitly, while verification and testing are not discussed.

9 CONCLUSIONS AND FUTURE WORK

Due to the increasing popularity of combining different Web services as a new Web service, robustness of such systems gained attention in the recent years. We have presented a model-based mutation testing approach for Web service compositions using the UP-

PAAL TA.

Our method starts with the design model that is specified as UPPAAL TA, verified UPPAAL TRON applied for conformance testing thereafter.

We used our mutation generator tool which implements a set of mutation operators applied on the test model for the purpose of online testing. In order to reduce the number of trivial invalid models and also increase the efficiency of testing, we defined a set of verification rules for each mutant. We verified whether the generated mutants are deadlockfree and if the mutation part of each mutant is reachable. If both of these criteria are satisfied, then we select the mutant as a valid mutant. We used UPPAAL TRON for executing all of the mutation models against the system under test.

We presented our approach with an experimental study on Hotel Booking System as a case study. The Web services are implemented in REST architectural style and with timing constraints. Our hotel booking case study has been designed and validated with UPPAAL test model and also the testing evaluated with a series of mutation in the source code of the case study.

The results showed that from a total 1346 generated mutants, 393 were found to be valid mutants that were usable for testing. After running the test, 40 of the mutants were found to identify 3 hidden faults in the implementation of the IUT. The experiment indicates that our approach of specification mutation testing was effective to reveal inconsistency between the specification and the implementation under test.

The primary results of this study showed that our method in robustness testing a valid approach in improving the quality of web service implementations, by detecting faults not detected by the traditional MBT process.

Our experiments also showed that some of the existing mutation operators for time automata are more efficient than the others at finding faults.

There are some research directions that certainly improve the current approach. The next work will be running more experiments, on different case studies in different application domains. More experiments help us to gain more information about mutation operators and correlations between the type of the case study and the common faults.

Another improvement will be to investigate how to detect equivalent mutants. Automation of this process of the approach reduces the errors and increases the scalability of the target applications.

Moreover, we plan to apply mutation selection and mutation reduction techniques to increase the probability of fault detection. Defining new mutation operators, categorizing the mutants, etc., will be investigated in our future work.

ACKNOWLEDGMENTS

We would like to thank prof. Andreas Zeller from University of Saarland, Germany for his valuable comments and anonymous reviewers for their useful suggestions.

REFERENCES

Aboutrab, M. et al. (2012). Specification mutation analysis for validating timed testing approaches based on timed automata. In *36th Annual IEEE Computer Software and Applications Conference, COMPSAC 2012, Izmir, Turkey, July 16-20, 2012*, pages 660–669.

Aichering, B. et al. (2013). Time for MutantsModel-Based Mutation Testing with Timed Automata. In *Tests and Proofs*, pages 20–38. Springer.

Alur, R. et al. (1990). Model-checking for real-time systems. In *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 414–425. IEEE.

Beharmann, G. et al. (2004). A tutorial on uppaal. In *Formal methods for the design of real-time systems*, pages 200–236. Springer.

Belli, F. et al. (2016). Model-based mutation testingapproach and case studies. *Science of Computer Programming*, 120:25 – 48.

Bozkurt, M. and other (2010). Testing web services: A survey. *Department of Computer Science, King's College London, Tech. Rep. TR-10-01*.

Budd, T. A. and Gopal, A. S. (1985). Program testing by specification mutation. *Computer Languages*, 10(1):63 – 73.

Cambronero, M. E. et al. (2011). Validation and verification of web services choreographies by using timed automata. *Journal of Logic and Algebraic Programming*, 80(1):25–49.

Dıaz, G. et al. (2007). Model checking techniques applied to the design of web services. *CLEI Electronic Journal*, 10(2).

Gargantini, A. and Heitmeyer, C. (1999). Using model checking to generate tests from requirements specifications. In *Software EngineeringESEC/FSE99*, pages 146–162. Springer.

Hessel, A. et al. (2008). Testing Real-time Systems Using UPPAAL. In Hierons, R. M., Bowen, J. P., and Harman, M., editors, *Formal Methods and Testing*, pages 77–117. Springer-Verlag, Berlin, Heidelberg.

Jia, Y. and Harman, M. (2011). An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678.

Larsen, K. et al. (2005a). Testing real-time embedded software using UPPAAL-TRON: an industrial case study. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 299–306. ACM.

Larsen, K., Mikucionis, M., and Nielsen, B. (2005b). Online testing of real-time systems using uppaal. In Grabowski, J. and Nielsen, B., editors, *Formal Approaches to Software Testing*, volume 3395 of *Lecture Notes in Computer Science*, pages 79–94. Springer Berlin Heidelberg.

Lee, S. et al. (2008). Automatic Mutation Testing and Simulation on OWL-S Specified Web Services. In *Simulation Symposium, 2008. ANSS 2008. 41st Annual*, pages 149–156.

Lee, S. C. and Offutt, J. (2001). Generating test cases for XML-based Web component interactions using mutation analysis. In *Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on*, pages 200–209.

Li, J.-h., Dai, G.-x., and Li, H.-h. (2009). Mutation analysis for testing finite state machines. In *Electronic Commerce and Security, 2009. ISECS'09. Second International Symposium on*, volume 1, pages 620–624. IEEE.

Rauf, I. et al. (2014). An Integrated Approach for Designing and Validating REST Web Service Compositions. In Monfort, V. and Krempels, K.-H., editors, *10th International Conference on Web Information Systems and Technologies*, volume 1, page 104115. SCITEPRESS Digital Library.

Richardson, L. and Ruby, S. (2008). *RESTful web services*. O'Reilly.

Rusli, H. M. et al. (2011). Testing Web services composition: a mapping study. *Communications of the IBIMA*, 2007:34–48.

Sheng, Q. et al. (2014). Web services composition: A decades overview . *Information Sciences*, 280:218 – 238.

Wang, R. and Huang, N. (2008). Requirement Model-Based Mutation Testing for Web Service. In *Next Generation Web Services Practices, 2008. NWESP '08. 4th International Conference on*, pages 71–76.

Publication V

V

Testing Web Services with Model-Based Mutation

Siavashi, Faezeh and Iqbal, Junaid and Truscan, Dragos and Vain, Jüri (2017). In *Software Technologies – Revised Selected Papers*, pages 45–67. Springer International Publishing

Testing Web Services with Model-Based Mutation

Faezeh Siavashi[1(✉)], Junaid Iqbal[1], Dragos Truscan[1], and Jüri Vain[2]

[1] Faculty of Science and Engineering, Åbo Akademi University, Åbo, Finland
{faezeh.siavashi,junaid.iqbal,dragos.truscan}@abo.fi
[2] Department of Computer Science, Tallinn University of Technology, Tallinn, Estonia
juri.vain@ttu.ee

Abstract. One way of evaluating the robustness of a web service is to test it against invalid inputs. We introduce a model-based mutation technique which automatically generates faulty test inputs. From the specification of a Web service, a test model is designed using Uppaal Timed Automata and the conformance between the model and the implementation is validated via online model-based testing with the Uppaal Tron tool. A set of mutation operators is applied to the test model in order to generate mutant test models. We validate all generated mutants via verification rules and select those that are executable and introduce proper mutations. We employ bisimulation as a tool for detecting and eliminating equivalent mutants, that is those mutants which have identical input-output behavior with the original test model. The resulting mutants are used for online test generation against the service implementation in order to check whether the latter allows for unspecified behavior. We discuss tool support and present an experiment of applying our method for a case study of a blog web service with real-life properties. The experiment shows that the proposed approach of mutating the specifications is effective in detecting errors both in the system functionality and in the test model.

Keywords: Web service · Model-based mutation testing · Uppaal · Tron · Bisimulation of Uppaal Timed Automata

1 Introduction

Software applications that support machine-to-machine interactions over the Internet have heavily increased the role of web services. One main characteristic of web services is that they are accessed via clearly defined interfaces over the standard HTTP protocol. This kind of systems should be robust against erroneous inputs. This means that one needs to ensure that the web service implementation is tested with respect not only to its expected behavior, but also to its unexpected behavior. The former can be checked by running test cases derived from the specification, whereas the latter can be done via robustness testing, by executing invalid inputs.

E. Cabello et al. (Eds.): ICSOFT 2016, CCIS 743, pp. 45–67, 2017.
DOI: 10.1007/978-3-319-62569-0_3

When defining test inputs, the model-based specifications are preferred over manually written test scripts since the machine can verify the correctness of the models and automatically generate the test inputs from them. Moreover, test generation from models enables systematic construction of extensive test cases.

One way to create invalid test inputs is using *specification mutation*, where a set of well-defined mutation operators generate syntactic changes to the specifications and produce specification mutants. Although originally the mutations have been applied directly to source code, it has also been extended also to specification languages [1]. When applied to modeling languages, mutation used to create the models that generate invalid scenarios as test cases, which then are executed against the implementation under test (IUT). If the IUT conforms to the mutated specification (i.e., the IUT accepts an unspecified sequence of inputs), it means that its behavior is inconsistent with its original specification and it may have unspecified or incorrect behavior.

In this work, we propose a tool-supported approach for robustness testing of web service using Uppaal Timed Automata (UTA). The conformance between the model and the IUT is first checked via Uppaal Tron, an online conformance testing tool which supports both test generation and test execution. As a first contribution, we introduce a test generation method, which derives mutants from the specification and executes them via online testing. We use a selection of mutation operators that are previously defined in the literature and adapt them for the online testing process targeted in this work.

As a second contribution, in our method, we add verification properties to mutated model segments to ensure reachability of the mutated elements at runtime. If a mutant does not satisfy the verification properties, it cannot be used for online testing, hence, we eliminate it. Furthermore, to ensure that the mutated part will be executed during the testing process, we monitor whether the mutated elements are reached during test execution.

As a third contribution, we provide an approach for detecting and eliminating equivalent mutants, that is those mutant models which exhibit identical timed input-output behavior with the original test model, even if the two models are syntactically different. For this purpose, we verify the timed bisimilarity of the corresponding Uppaal timed automata models.

As a forth contribution, we empirically evaluate which of the existing mutation operators for Uppaal timed automata are effective for online testing of web services. For this purpose, we define two formulas to measure the efficiency of mutation operators as well as their fault detection rate.

Parts of this work have been originally presented in [2]. In this version, we extend previous work as follows: we provide a method for detecting equivalent mutants, we discuss tool support for the entire approach, we address a larger set of mutation operators, we use a different case study to complement the previous results and we provide a more detailed analysis of the results.

The remainder of this paper is organized as follows: In Sect. 2, we briefly revisit the background concepts behind Uppaal timed automata, conformance testing with Tron, and specification mutation analysis. Section 3 details the steps of our approach and its tool support. The case study and the experiments used to

validate our approach are presented in Sect. 4. The results are discussed in Sect. 4.6 and possible improvements are suggested in Sect. 6. Threats to validity of the proposed method are discussed in Sect. 7. We review the literature for related work in Sect. 8. Finally, we conclude our study and present future work in Sect. 9.

2 Background

We first review the UPPAAL tool set and we introduce the conformance testing with UPPAAL TRON , then we elaborate on the concept of specification mutation testing.

2.1 UPPAAL Timed Automata (UTA)

UPPAAL is a model-checker tool for modeling, simulation, and verification of real-time systems using an extended version of timed automata called UPPAAL timed automata (UTA) [3]. A timed automaton is a state machine with locations, actions, and clocks.

In UPPAAL, a system is designed as a network of several such timed-automata called processes working in parallel. A process can be executed individually or in sync with another process. Synchronization of two processes is possible by using input/output actions (denoted as "!" for emitting and "?" for receiving synchronizations, respectively). The processes consist of locations and edges. The state of the system can be shown by the locations of all processes, their clock value intervals, and their variable values. The edges between locations represent state transitions including clock resets. UPPAAL is extended further with global and local to some process variables that can be of type *integer*, *boolean*, *clock* and arrays of those.

Edges can be constrained by predicates (over the clocks or variables) known as *guards*, which defines when the corresponding edge is enabled. State transitions are specified on edges as variable updates. A location can be restricted over the clock invariants, which specify how long the system can stay in that location. If there is more than one enabled edge at a time, then one of them will be randomly selected. This gives more freedom to represent non-deterministic behaviour, especially in systems with random discrete events [4].

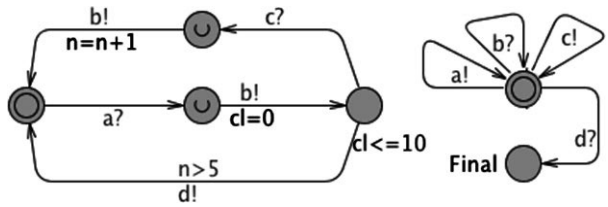


Fig. 1. Example of an UTA model [2].

An example of a UTA model consisting of two timed automata is shown in Fig. 1. The communication between automata is modeled using channel synchronizations (e.g., a, b, c, d) and variables (e.g., n). Time is modeled via the clock variable cl.

The Uppaal model-checker uses a simplified version of TCTL [5], which enables to exhaustively verify the models w.r.t their specifications. The query language consists of state formulae and path formulae. State formulae (φ) is an expression that describes the properties of an individual state while path formulae can be used to specify which properties (like reachability, safety, and liveness) hold over a given path [6].

If there is a state in the model that has no enabled outgoing transitions, then the model is said to be in a deadlock. $A\ \Box$ *not deadlock* query, can be used to verify that for all paths in the model, there is no deadlock state.

The safety property checks that "something bad will never happen". In Uppaal it can be expressed in the form $A\ \Box\ \varphi$ (φ should be true in all reachable states) and $E\ \Box\ \varphi$ (there should exist a maximal path such that φ is always true).

The liveness property determines that "something will eventually happen" and it is shown by $A\ \Diamond\ \varphi$ (φ is eventually satisfied) and $\varphi \rightsquigarrow\ \phi$ (whenever φ is satisfied, then eventually ϕ will be satisfied).

Reachability properties validate the basic behavior of the model by checking whether a certain property is possible in the model with the given paths. The reachability can be expressed in the form of $E\ \Diamond\ \varphi$ (there is a path from the initial state, such that φ is eventually satisfied along that path).

2.2 Online Model-Based Testing

Model-Based Testing (MBT) [3] is an approach which uses behavioral models of the system under test to generate tests. Based on how tests are generated and executed, there are two distinct approaches of MBT: offline and online testing. In *offline testing*, the complete test scenarios and test oracle are created before the test execution, whereas online testing is a combination of test generation and execution: only one test input at a time is generated and executed, then the next test input is generated based on the previous test output [7]. This continues until the test termination criteria are satisfied or an error occurs.

In this study, we use the online testing tool Uppaal Tron, which is an input/output conformance testing tool for testing real-time systems based on the *rtioco* conformance relation [8]. In Tron, the UTA model is divided in two partitions: a system partition and an environment partition, and the communication between the two is observed against the inputs and outputs of the IUT. Test stimuli are selected randomly from the enabled test inputs. A *test adapter* is used for converting abstract test cases to concrete inputs to the IUT and for converting concrete outputs into abstract outputs represented in the model. Via *online testing*, the state-explosion problem is reduced because only a portion of

the state space is needed to be calculated and stored at each step. Also, the non-determinism of systems can be simulated on-the-fly by random selection of the test inputs.

The result of online testing with TRON can be *passed*, *failed*, or *inconclusive*. An inconclusive test result means that the environment model cannot progress since the IUT output is unexpected or timeout occurred when waiting for test output.

2.3 Specification Mutation Analysis

Specification mutation analysis is an approach used to design tests to evaluate the correctness and consistency of the specification or of the program [1]. When the mutation analysis is applied to the specification, a set of *mutation operators* create slightly altered versions (mutants) of the specification. The tests will be generated from the mutated specification and used to assess whether the IUT is accepting the tests. The following types of mutants are defined in the literature [9]:

- **Killed:** the tests generated from a mutant specification fail against the implementation, under the precondition that the tests generated from the original specification have passed.
- **Alive:** all test cases generated from the mutant pass against the IUT. Alive mutants can be divided into two sub-types:
- **Equivalent:** The mutant manifests the same behavior as the original model, even if they are syntactically different.
- **Non-equivalent:** The mutant does not have the same behavior as the original model, however, all tests generated from the mutant pass against the IUT. These mutants indicate that the implementation is too permissive and is not able to detect the invalid inputs.

3 Method

In this section, a method which combines specification mutation testing and online model-based testing is presented. The outcome of the method is to generate and identify *non-equivalent alive mutants* which are used to show that there might be some inconsistencies between the specification and the implementation. The collected mutants are subject to further investigation to identify whether the source of the problem is in the specification or in the implementation. Figure 2 illustrates an overview of the method including six main phases (separated by dashed lines), as follows:

3.1 Design and Conformance Testing

From the given specifications of a system, e.g., a web service, a test model is designed. The test model consists of two partitions: the system under test and

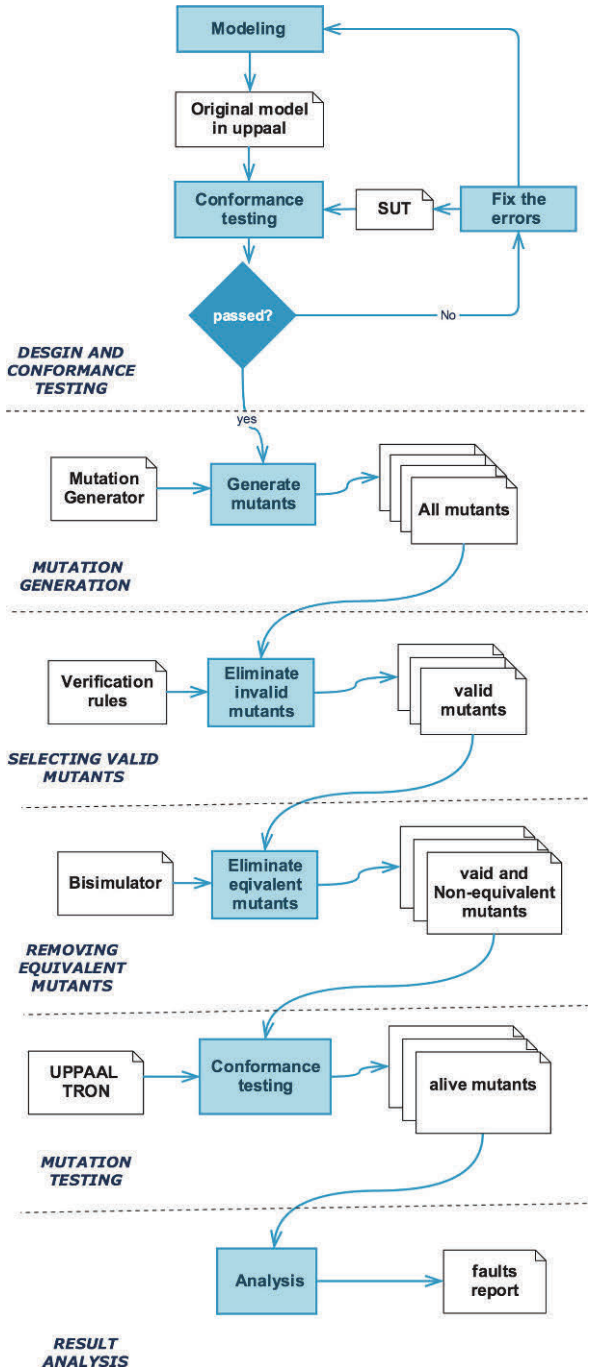


Fig. 2. Model-based mutation testing approach.

its environment. The former models the activities that a typical user performs against the web service, while the latter models how the system should respond to user activities. For instance, a flight booking server receives different HTTP requests (such as searching, checking in, etc.) and returns the corresponding HTTP responses. For each request and response, two edges are specified in the model. It should be noted that, failure responses are not modeled since adding all possible failure response types might make the model too complex.

In order to ensure that the model is correctly designed according to the specification, we verify it via model-checking. Deadlockfreeness and reachability properties are two common and essential properties that the model should satisfy. These properties ensure that the model can be used later on for online testing.

The test adapter is implemented to convert the observable test interface I/O actions into HTTP messages and vice-versa. The adapter is also used to check the status codes of different HTTP responses, before forwarding them to the tester. The TRON testing tool orchestrates the communications between the test model and the IUT, and check the I/O conformance between the two. During the online testing, the expected behavior of the IUT is validated and possible modeling errors or implementation bugs are resolved.

3.2 Mutation Generation

From a verified test model various modified versions are created. Each modified version of the original model is called *mutant model* (or simply *mutant*). Test generated from mutants will also exhibit a mutated behaviors compared to the original ones.

Mutation operators implement the rules that create systematic mutation of a given context. They are uniquely defined for a specific modeling language. For example, different modeling languages (UML, Petri Nets, UPPAAL, etc.) have different mutation operators. Mutation operators for UTA have been previously defined by Aboutrab et al. [10] and by Aichernig et al. [11], as summarized in Table 1. As one may notice, the two sets of operators are mostly similar in purpose, however they differ in the restrictions that are employed in each element. For instance, for a guard, three mutation operators (RTC, WTC and STC) are defined by Aboutrab et al., while, all three definitions are covered in one mutation operator (μCG) in the work of Aichernig et al.

Mutation operators that are used in this paper are selected from this list, combining the definition of the similar mutation operators in both studies. Only one of the mutation operators (sink location) was not selected in this study, since it will produce higher order mutations which are beyond the scope of this work.

We have restricted some of the operators to make them suitable for online testing with TRON. As we mentioned earlier, the IUT and its environment (user, or other systems) are specified in separate automata and they communicate via synchronization channels and global variables. All channels between the model of the system and its environment and the variables attached to those channels are observable by TRON. Based on the type of the input or output, TRON controls which action can be executed at a given time. Therefore, if there are multiple

Table 1. Mutation operators of timed-automata, [2].

Mutated elements	Aichering et al. [11]	Aboutrab et al. [10]	Informal definition
Guard	Change Guard (μCG)	Restricting Timing Constraints (RTC)	Restricts, expands or alters guards
		Widening Timing Constraints (WTC)	
		Shifting Timing constraints (STC)	
	Negate guard (μCg)	-	Guard will be replaced by its negation
Invariant	Change invariant (μCi)	-	Restricts, expands or change value of invartiants
Clock	Invert reset (μIr)	Resetting a Clock (RC)	Removes or adds clock resets
		Not-Resetting a Clock (NRC)	
Action	Change action (μCa)	Exchanging Input Actions (EIA)	Changes names of actions
		Exchanging Output Action (EOA)	
	Change source (μCs)	-	Changes source location of actions
	Change target (μCt)	Transferring Destination Locations (TDL)	Changes target locations of actions
Location	Sink location (μSl)	-	Makes a new locations and changes targets of all actions to the new location

processes in the model and some of the synchronizations among them are not defined in the IUT, then they cannot be observed by TRON. Thus, they will not be mutated either. It should be noted that in this study, only the system under test (SUT) partition is mutated and we limit our approach to partitions with only one timed automaton process.

Additionally, the mutation operator for changing the name of the actions (i.e. μCa) is only applied on input actions in the SUT model. The reason behind this is that the implementation of the web service is a black box and thus we cannot change them. The requests (inputs actions), on the other hand, come from outside of the SUT and can be manipulated. We select EIA mutation operator that will be effective since the sequence of requests will be mutated, whereas mutating the output actions does not make a suitable mutation.

Finally, the direction of synchronizations will not be changed (i.e., switching "?" to "!") since the requests from the environment are modeled as input actions ("?") and changing them into output actions indicates that the requests will be changed into responses, which is not applicable in web services. The web service is the receiver of the requests from the user and not the sender of the requests.

The mutation operators adapted from Table 1 and used in this paper are presented in the following.

1. **Change Name of Input Action (CNI).** This operator is same as EIA, which replaces the name of an input action (denoted by "?") with the name of other actions. Thus, the expected sequence of the inputs to the implementation will be different.
2. **Change Target (CT).** This operator is similar to TDL and μCT. As it name suggests, it changes the target of an interface action to other location. This operator can break the flow of test inputs and violate the state of the IUT. Both input and output actions can be mutated by this operator.
3. **Change Source (CS).** This operator is similar to μCS defined in [11] changes the source location of an action to other locations. Similar to CT, this operator gives a different I/O sequence.
4. **Change Guard (CG).** For this operator, we followed the definition of μCG, which changes the clock constants in guards by a random value. It is effective for mutating the enabling condition of an action.
5. **Negate Guard (NG).** It is the same operator as μNG which negates the guards and may cause some paths of the test model to become unreachable.
6. **Change Invariant (CI).** Similar to μCI, it shifts values of the invariants to a different range, extending or restricting the constraints of the model. It can cause actions fire earlier (or later) than expected by original model.
7. **Invert Reset (IR).** This operator is same as μIR which deletes the resetting of the clock and moves it to one action before or after. It means that the resetting is shifted one edge earlier or later.

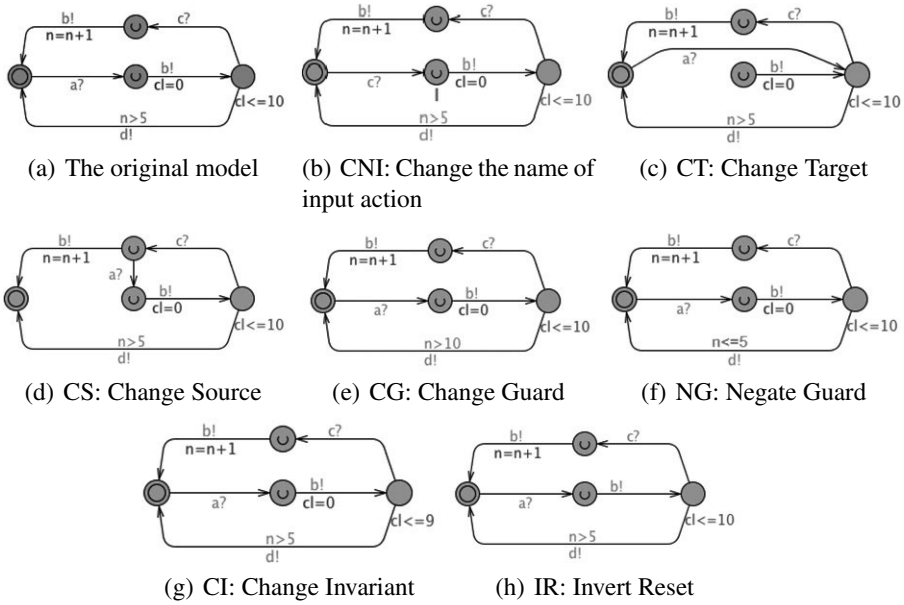
Figure 3 shows the mutants of a model and corresponding mutants using the above operators. In our approach, we only apply first order mutation. That is, a mutant model contains only one mutated segment based on a single operator.

3.3 Selecting Valid Mutants

In the context of this paper, we define a valid mutant as one which can be executed by TRON and in which the mutated part is reachable. To this end, we verify if all mutations are reachable and deadlock free.

In UPPAAL, the reachability property is defined for locations and the valuations of variable sets. When an action is mutated, we define the reachability property for the target location of that action. For instance, in Fig. 3(b), the input action $a?$ is mutated into $c?$, hence, the reachability for this mutation should be defined for its target location (i.e., l). For example, in Fig. 3(b), we have $E \diamond l$, which verifies if that the mutation can be reached and executed.

Nevertheless, for CT and CS that change target and source of actions, the above reachability is not suitable. Thus, we add an alternative method to define reachability in these circumstances. For a mutated action, we add a *trap variable* update [12] on its edge. The initial value of the variable is set to `false`. The variable will be updated to `true` whenever the mutated action is executed, and



(a) The original model (b) CNI: Change the name of input action (c) CT: Change Target

(d) CS: Change Source (e) CG: Change Guard (f) NG: Negate Guard

(g) CI: Change Invariant (h) IR: Invert Reset

Fig. 3. A model with examples of mutants generated by the selected mutation operators [2].

so the reachability can be achieved by checking if the variable eventually will be set to true ($E \diamond trap$).

One can use trap variables to ensure that the mutation part of the model will be reached during the test execution as well. In the case that the minimum repetitive execution of mutation is needed the boolean trap variable should be replaced by an integer counter variable *count* and the reachability condition with $E \diamond count >= const$. Those models that pass the verification process are considered as *valid mutants* and can be executed against the IUT.

Beside reachability, the deadlock-freeness property will also be verified. The deadlock freeness property can be expressed as $A\Box$ *not deadlock*, which indicates that for all existing path in the model there is no deadlock.

Early validation of the mutants reduces the number of final valid mutants by eliminating false negatives which cause semantic and syntactic errors.

3.4 Detecting and Removing Equivalent Mutants

In order to detect those mutants which have equivalent observable input-output behavior we employ bisimulation relation checks. Intuitively, two UTA are *bisimilar* if they accept the same timed language, i.e., they perform exactly the same observable action transitions and if they reach bisimilar states. In other words, each of the systems cannot be distinguished from the other by an external observer. Bisimulation relation is symmetric. Bisimulation for timed automata has been originally introduced by [13] and shown in [14] to be decidable for

parallel timed processes. In order to observe bisimilarity between the original model and one of its mutants, we follow these steps:

1. we compose a new UTA model containing both the original and a mutant model,
2. we add additional, side-effect free, synchronization channels between the models for the observable actions,
3. we verify that the complete model never deadlocks on all possible paths.

The mutants selected after the validation step described in Sect. 3.3 were examined for their bisimilarity.

Figure 4 depicts an example of a bisimulation model used to detect bisimilarity between two models. The observable channels and shared variables between environment and the SUT partition of the mutant process are renamed (using the *BISIM_* prefix) and added as a counterpart to observe the bisimilarity. A committed location between each channel and its counterpart to ensure that both observable actions take place at the same time. Thus, the deviation of the behavior of either process results in a deadlock which violates the condition of bisimilarity. The non-bisimilar models are good candidates for mutation testing due to their erroneous behavior. All mutants which are found to be bisimilar with the original model are considered equivalent, and consequently, are eliminated.

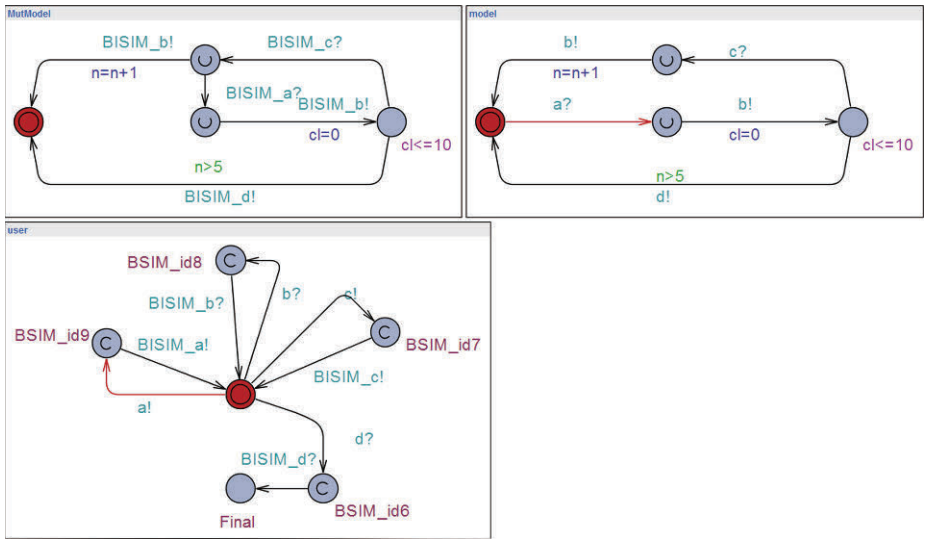


Fig. 4. UTA containing an original process (*model*), a mutated process (*MutModel*), and a shared environment process (*user*).

3.5 Mutation Testing

Each valid mutant model is executed in a testing session with Uppaal Tron. The verdict of an online testing session with Tron can be *passed*, *failed*, or

inconclusive. In TRON, an inconclusive verdict indicates that either the observed output from the IUT is not valid, or there is an unacceptable delay in receiving responses from the IUT. We consider that the mutants that generate inconclusive test cases exhibit different behavior than the original model and thus they are considered as *killed*. When executing the mutants we assume implicitly that these test runs are exhaustive w.r.t. the mutation, i.e. all mutations injected are also covered by these test runs.

3.6 Result Analysis

The last phase of our method is to evaluate the results by reasoning about the unexpected behaviors that the IUT shows during test execution. The focus of the analysis is on the non-equivalent mutants, which generate different invalid test inputs, thus, these test inputs are manually evaluated to find the correlations between them and the actual faulty behaviors.

3.7 Tool Support

Tool support has been implemented to automate several of the activities discussed in the previous section. The UPPAAL tool set is used for modeling and verification of the original model. Then the TRON tool and a test adapter is implemented to interface TRON with the IUT.

A prototype tool set, called *MuUTA*, has been implemented to support the generation of mutants based on the selected mutation operators, to automatically perform the verification of reachability and deadlock-freeness rules for each mutant via the *verifyta* utility of UPPAAL and run bisimulation checks. The tool also instantiates a test session for each mutant using TRON test adapter and IUT and eliminates those mutants that are killed. All generated mutants are stored based on their status in corresponding folders for further analysis.

4 Experiment

We exemplify our approach using the case study of a blog web service. We define the specifications of the web service and present different use cases that are satisfied.

4.1 Case Study

The case study represents a blog website that is implemented in REpresentational State Transfer (REST) [15] architectural style. The web service provides functionality for creating a user account, posting new articles, commenting, deleting/editing posts and comments, managing the user profile, etc. similar to other social networks. These main characteristics of a sample blog web service include authorization of users to access to features of the web service. The web service is implemented in Python using Flask web developing micro framework [16]. Figure 5 shows a use case diagram of the blog web service. Each use case is detailed below:

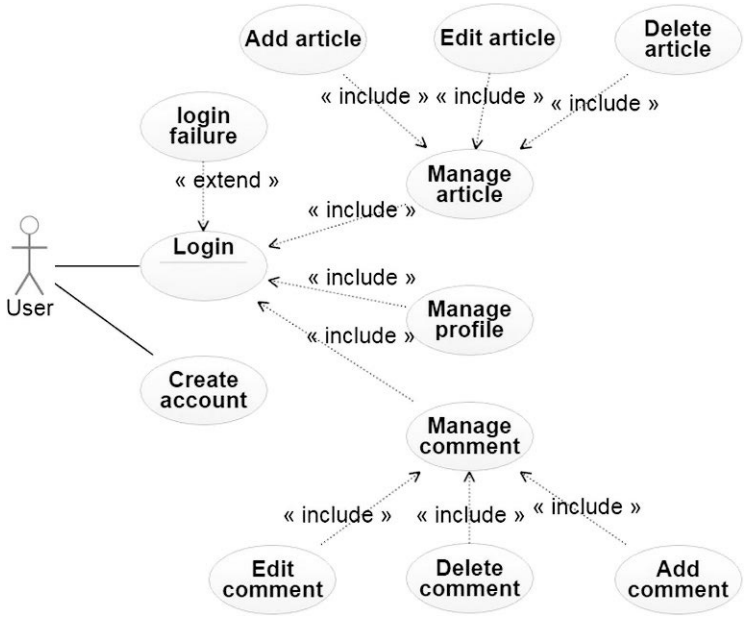


Fig. 5. Use case diagram of blog web service.

- **Create New Profile.** A new user can create a unique profile in blog service in order to use the features of the web service. It includes inserting a valid (and unique) username and a password.
- **Log In.** A user is able to log in with registered username and password. No two usernames are similar.
- **Manage Profile.** A signed up user can have access to his profile for further settings.
- **Delete Profile.** A user can delete his profile. This action logs him out from the blog as well as removes all of his posts and comments.
- **Post New Article.** A user can post new articles. Each article has a title and body.
- **View Articles.** Both user and reader (blog reader) are able to search throughout the blog and read the posted articles.
- **Comment Articles.** A user can comment on articles of the blog.
- **Edit/Delete Articles.** A user, who is owner of an article can edit/delete it.
- **Edit/Delete Comments.** An owner of the article can manage his comments.
- **Idle User.** There is a timer in the blog which checks whether a user is idle for more than 10 min. If so, then the service will automatically logs out the user.

4.2 Model

From the above descriptions, we have specified the system as a UTA model which consists of two automata: one for the blog web service and one for the environment (user). Figure 6 shows the models of the case study and the interactions between the service and the users.

From the specifications, we can define some use case scenarios which are designed in the model as well. For example, the above specification of edit/delete an article is designed in the model accordingly. In order to delete/edit an article, in the user and blog automata are synchronized as follows: the user sends a request *logged_in*, which is received in the blog by the same channel. The response from the blog will be either *loged_in* which changes the state of the model to the next location (id46 in blog) or $login\_failed$ which returns it to the initial location. This synchronization will continue by *manage_ar* (getting an specific article), which will be responded by either *access_ar* (access to the article) or *access_denied* (does not allow to have access to the article, or the article does not exists). If the response channel is *access_ar*, then there are two options for the next request from the user: *edit_ar* and *delete_ar* which will be responded by *edited_ar* and *deleted_ar* respectively. Similarly, the rest of the specifications are designed as UTA models as described above.

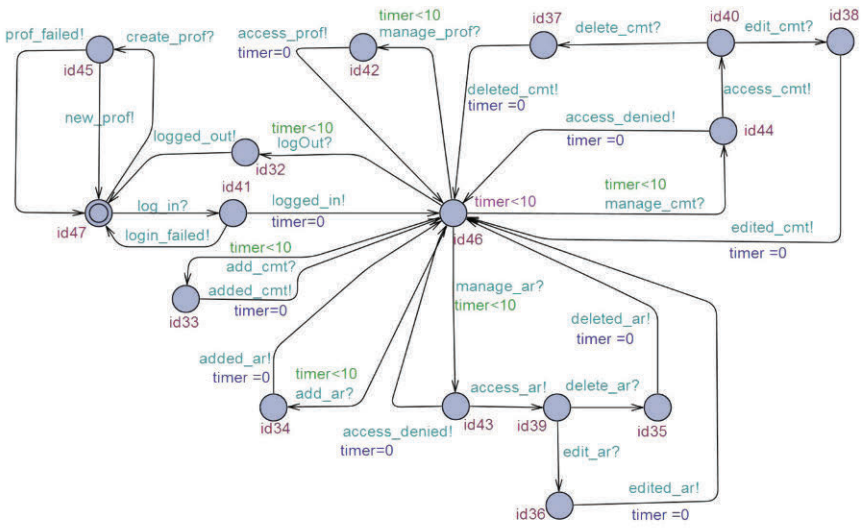
The model is verified before using it for test generation for deadlock-freeness and that the requirements are satisfied. For example, to ensure that editing article is possible we define a global boolean variable (e.g. a) and update its value to TRUE on *edited_ar*. Then we define a reachability property like $E \Diamond a == true$ which, if satisfied, indicates that there is at least a path in which a will be eventually true.

4.3 Generating Valid Mutants

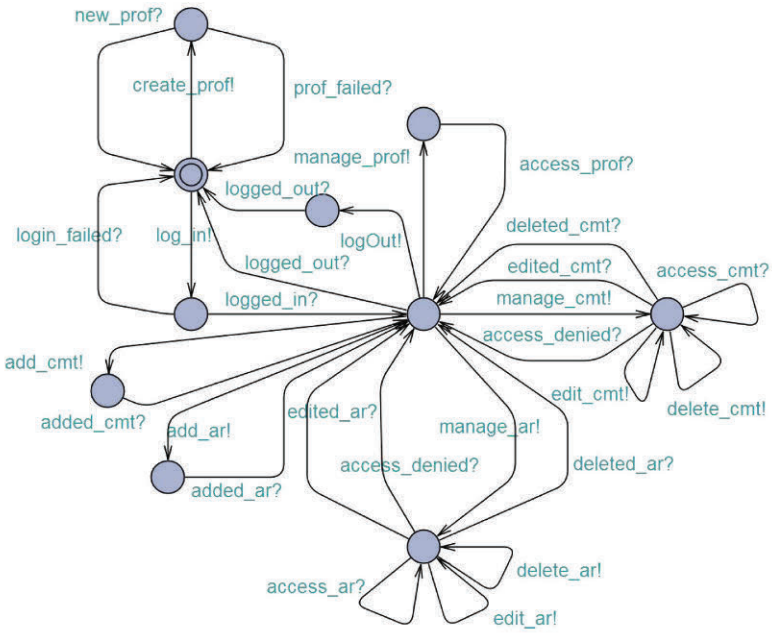
In this experiment, we mutate the blog automaton (Fig. 6(a)) to generate mutated models. The total time for generation and validation of all mutants took 246 s on a Windows 7 Enterprise 64-bit operating system, Intel quad-core CPU and 16 GB RAM. In total, the generator provided 1019 mutants, of which 470 mutants were valid (i.e., passed the verification properties). As the numbers suggest, early verification of the mutants is helpful in having only mutants applicable for testing and thus, reducing the time of the test execution. The majority of the valid mutants, 300, were generated by the CS operator, in contrast with 102 valid mutants provided by CNI and 32 by CT. All generated mutants by IR and NG passed the validation process, and other mutation operators have a small share of valid mutants.

4.4 Detecting and Removing Equivalent Mutants

After performing equivalence checking, a number of 31 mutants were eliminated as being bisimulation equivalent. We eliminated them from the valid mutants,



(a)



(b)

Fig. 6. Behavioral model of blog: (a) blog service, (b) blog user.

resulting a number of 439 valid non-equivalent mutants. The equivalence detection process took 286 s. Obviously, none of the mutants generated by CS, CT, CG and CNI were equivalent since they change the trace of the mutant which is clearly different than the original one. Only one mutant from NG is remained as valid non-equivalent. All mutants generated by IR and NG were equivalent and thus were eliminated.

4.5 Mutation Testing

We execute each mutant with TRON against the implementation of the blog web service. Each test session is set for 180 s in sequence, and we check that the mutated state is covered by the test and is also covered at runtime. It took roughly 8 h to execute the 439 non-equivalent mutants of which 436 have been killed.

4.6 Results Analysis

The resulting 33 alive mutants have been used for further analysis. In this step, we are interested in understanding why the tests generated from each alive mutant did not fail against the IUT. The process is done manually.

We detected 3 inconsistencies in the test model and the test adapter while no error in the code. In contrast, to the results presented in the previous paper [2], where the errors were detected on the implementation of the IUT, in this paper the detected errors were localized in the test adapter and the test model only.

The inconsistencies that are found are as follows:

- Mutation in some of alive mutants change the timing of which could not be killed by the IUT. Mutants by CG and NG are mainly addressing this inconsistency.
- For some mutants, although the IUT detects the mutation, the test adapter does not stop the test session. Some of alive mutants by CT revealed this problem.
- Some mutants generate the cases that regardless their difference from the original test cases, they are not erroneous. Alive mutants by CS, CN and CT revealed the same problem.

5 Analysis of Experimental Results

One of the questions that we wanted to answer in this paper was about the efficiency of the mutation operators used in our approach. Table 2 summarizes the results of the method at different steps, while Fig. 7 shows the 100% stacked bar chart showing the ratio of different types of mutants for different operators. For instance, IR and NG have the highest percentage of valid mutants in proportion to their generated mutants, however, majority of them were eliminated by the equivalence detection. The valid mutants in CS and CNI are more than half of

Table 2. Result of mutation testing.

Name	Generated	Valid	Non-equivalent	Alive
CNI	132	102	102	1
CT	420	32	32	8
CS	420	300	300	19
CG	12	4	4	4
NG	16	16	1	1
CI	3	0	0	0
IR	16	16	0	0
Total	1019	470	439	33

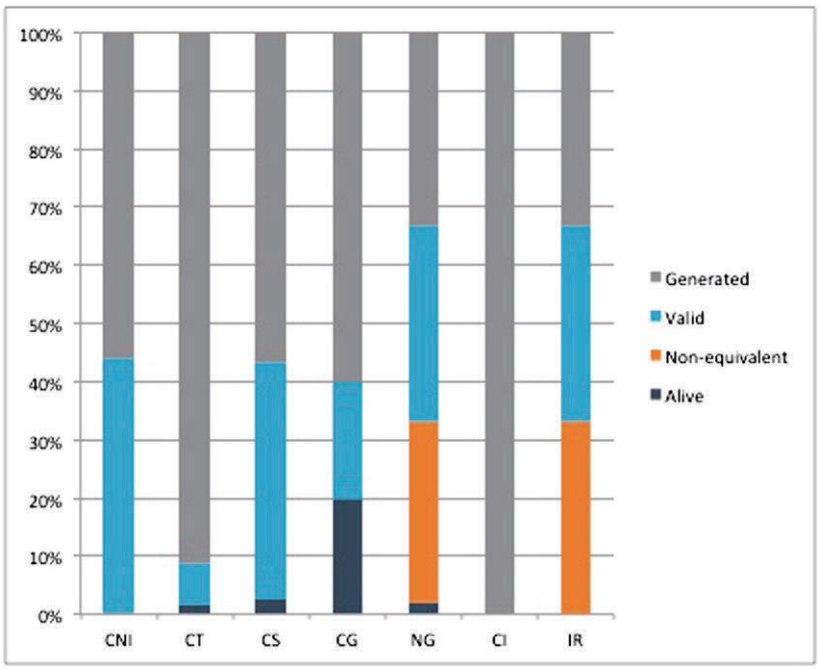


Fig. 7. The proportion of the result of each mutation operator and the total result of testing.

their generated mutants and none of them were equivalent. Small proportion of generated mutants in CT and CG are valid and no valid mutant is generated by CI.

Two formulas were defined previously for the efficiency of mutation operators showing how many of the mutants reveal the faults [2]. By calculating the number of equivalent mutants we can calculate the mutation efficiency for each mutation operator as follows:

Mutation Efficiency: For each mutation operator, we calculate how many mutants are alive. We calculate the efficiency of each mutation operator in generating alive mutants:

$$ME_i = \frac{A_i}{V_i} \ , \tag{1}$$

where A is the number of alive mutants, V is the number of valid non-equivalent mutants of the mutation operator i.

Mutation Fault Detection: Since after finding a fault in each category, we discard the rest of them, the formula for fault detection will be applied based on the categories. For each category that shows a fault, we score the corresponding operators. For each operator, we measure the mutation fault detection with following formula:

$$MFD_i = \frac{NE_i}{T_i - E_i} \ , \tag{2}$$

where NE is the number of non-equivalent mutants that reveal hidden faults, T is the number of total mutants and E is the number of equivalent mutants.

By using Formula 1, we calculated the efficiency of the operators that have alive mutants. The result shows that all alive mutants generated by CG remained alive, however, none of them shows a fault and thus CG has 0% in fault detection. A quarter of generated mutants by CT and only 6.3% of the mutants generated by CS were efficient. While CS is in the bottom of the list in ME, it has the highest rank in MFD. It means that CS is more able to detect faults in the model and the test adapter. Table 3 illustrates information on how the mutation operators are able to show some faults in the case study.

Table 3. Mutation efficiency and mutation fault detection for the suggested operators.

Operators	ME	MFD
CNI	∼1%	0.7%
CT	25%	1.6%
CS	6.3%	2.3%
CG	100%	0%
NG	6.2%	0%

6 Discussion

Some improvements can reduce the test execution time while increasing the probability of finding faults. The results of mutation testing from this study and the previous one [2] indicate that an intelligent choice of the mutation operators can attain high mutation efficiency scores while reducing the time of testing. For instance, the mutations generated by IR, NG and CG were not effective, thus either they should be changed to stronger operators or simply not considered.

Another improvement could be done in the process of fault detection. Redundant work is done on detecting the same faults. This extra effort can be reduced by categorizing the alive mutants in such a way that all mutations of a certain location or action in the model will be in a single category. As soon as any of the mutants in a category detects a fault, then the rest of the mutants on that group can be eliminated from the fault detection analysis. The idea behind this is that the locations and actions in a model represent actual states of the system under test and if there is a state which contains a fault, then any mutant from that state may be able to reveal that fault. However, more experiments are needed to show the correctness of this test effort reduction technique.

The main downside of model-based mutation testing comes from MBT: the process of designing models from the specification, verifying them and writing the test adapter (to translate model-level test inputs into acceptable test script for the IUT and vice versa) is time consuming. However, once the above artifacts are created, the model-based mutation testing process could be automated to a large extent. Also reusing existing models from development process would help in reducing the effort as it is often the case in regression testing. Moreover, the new method of equivalence detection helped us to reduce the number of alive mutants into half. The mutation testing does not add any overhead into MBT. The mutation generator tool automatically generates correct and valid mutations and thus, it reduces the mutant generation time.

Finally, the process of result analysis is manual and for large-scale systems, it is tedious. But some degree of tool support could be provided via TRON2UPPAAL backtracing tool described in [17]. This tool allows one to load a test execution trace generated from TRON and load it in the UPPAAL simulator for visualization and step-wise debugging.

7 Threats to Validity

There are two main threats related to our study. One is related to the mutation operators. Despite the fact that we have followed the systematically and formally defined mutation operators and implemented them in our study, there might be some more effective mutation operators or combinations of operators that we have missed. We argue that the current number of mutation operators provides a large number of mutants which can provide faulty test inputs which are close to the accepted inputs.

The other threat is that the results are strongly related to the test model and to the case study used. Different test designers can specify the same system in various ways which may provide different results of mutation testing. Perhaps having a systematic modeling approach specifically for web services (if there is any) would resolve such threat.

More extensive studies are needed in order to investigate how the specification mutation can be applied in larger case studies preferably industrial-sized web services. Besides, more experiment on larger scales would be helpful in finding whether there is any correlation between certain mutation operators and the real faults in design and implementation of web services.

8 Related Work

A comprehensive analysis is done on all available mutation testing method presenting the current state of the art in this field and the open challenges [18].

Lee and Offutt [19] introduced an Interaction specification Model which formalize the interactions among Web components. They defined a set of mutation operators for XML data model in order to mutate the inputs of the Web components. Li et al. [20] presented mutation testing methods using XML schema to create invalid inputs. Mutation testing is extended to XML-based specification languages for Web services. Lee et al. presented an ontology based mutation operators on OWL-S, which is an XML-based language for specifying semantics of web services [21]. They mutate semantics of the specifications of their case study such as data mutation, condition mutation, etc. Wang and Huang presented a mutation testing approach based on OWL-S to validate the requirements of web services [22]. Also, Dominguez et al. presented a mutation generator tool for WS-BPEL.

We discuss those that are most similar to our approach. Work has been done on using model checking techniques for validation and verification of web servicesWSC. There are two studies that review the literature on testing Web services [23,24]. Starting from specification languages for modeling Web services, researchers perform simulation, verification and test generation using model checking tools. Most of the works use model checking for specification and verification and only one group use the models for the test generation as well.

Using TA models for mutation testing has been mostly studied on a real-time and embedded system. In [10,11] mutation operators for TA are presented. Aboutrab et al. proposed a set of mutation operators for timed automata to empirically compare priority-based testing with other testing approaches [10]. However, in their approach, the generation of mutations is done manually. Aichernig et al. presented model-based mutation testing real-time system using Uppaal [11]. The mutation operators that are defined in their work are more detailed and some of them are implemented as mutation bounded model-checking and incremental SMT solving. They showed that using mutations for timed automata has potential on debugging and revealing the unexpected behavior of the IUT.

We applied/modified the mutation operators of TA presented by these studies for testing the robustness of web servicesWSC. Similar to [11], we apply mutations on non-deterministic models, however, in their work, they use only the UTA model of the IUT and do not consider the environment. In our approach, however, each mutant is a closed model communicating with its environment. We check deadlockfreeness and reachability in order to reduce the number of invalid mutants. Also, we use different verification and test generation processes.

There are some works that target UTA as the specification language for Web services. In most of the works, the authors transformed the specification that is defined in their selected languages into UTA and then they investigated its properties. For instance, in [25], the specification of a web serviceWSC is defined initially in the form of UML and then transformed into UTA for an online testing

purpose. In [26], Cambronero et al. verify web services by the Uppaal tool for validation and verification of their described system that is transformed from WS-CDL into a network of TA. In [27], Diaz et al. also provide a translation from WS-BPEL to UTA. Time properties are specified in WS-BPEL and translated to UTA. However, requirements are not traced explicitly, while verification and testing are not discussed.

9 Conclusions and Future Work

The popularity of web services has significantly increased in recent years and as a consequence their robustness and reliability have become more important. One way of testing robustness of such dynamic systems is to check their behavior against invalid and stressful environment. In this paper, a model-based mutation testing method is presented using the Uppaal TA for assessing the robustness of web services.

The method includes six steps, starting with designing a test model via Uppaal TA and executing online conformance testing with Uppaal-Tronagainst the implementation of the service, continuing with the generation of mutant test models based on a selection of mutation operators and with eliminating invalid and equivalent mutants. Then, the IUT is tested against each mutant, and the mutants which result in a failure are considered killed and eliminated. Finally, the results of the mutation testing are evaluated manually, by investigating the alive mutants, in order to reveal potential faults in the test model or in the IUT.

In this paper, we evaluated the presented method by experimenting Blog System as a case study. The web service is implemented in REST architectural style and with timing constraints. The results showed that from a total 1019 generated mutants, 470 were found to be valid mutants, that were usable for testing and from 33 alive mutants, three different errors in the test model were uncovered. Combined with the previous results on a different case study, in which several faults were uncovered in the implementation, it shows the our approach has the potential to detect faults not found otherwise via functional testing. One improvement compared to the previous work of the paper is in the automatic detection of equivalent mutants, which was previously done manually and which now allowed us to reduce the number of mutants.

There are some research directions that would certainly improve the current method. From the two different case studies of web services, we have achieved some useful information for reducing the testing effort specially in larger scale use cases. Automation of this process of the approach reduces the errors and increases the scalability of the target applications.

For future work, we plan to run more experiments and evaluate a larger set of mutation operators. We also plan to make the mutation process more efficient by applying more mutation selection and mutation reduction techniques and by running different processes in parallel.

References

1. Budd, T.A., Gopal, A.S.: Program testing by specification mutation. Comput. Lang. **10**, 63–73 (1985)
2. Siavashi, F., Truscan, D., Vain, J.: On mutating UPPAAL timed automata to assess robustness of web services. In: Maciaszek, L., Cardoso, J., Ludwig, A., Sinderen, M.V., Cabello, E. (eds.) Proceedings of the 11th International Joint Conference on Software Technologies, vol. 1, pp. 15–26. SCITEPRESS-Science and Technology Publications (2016)
3. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Softw. Test. Verif. Reliab. **22**, 297–312 (2012)
4. Hessel, A., Larsen, K.G., Mikucionis, M., Nielsen, B., Pettersson, P., Skou, A.: Testing real-time systems using UPPAAL. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) Formal Methods and Testing. LNCS, vol. 4949, pp. 77–117. Springer, Heidelberg (2008). doi:10.1007/978-3-540-78917-8_3
5. Alur, R., et al.: Model-checking for real-time systems. In: Proceedings of Fifth Annual IEEE Symposium on e Logic in Computer Science, LICS 1990, pp. 414–425. IEEE (1990)
6. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004). doi:10.1007/978-3-540-30080-9_7
7. Larsen, K.G., Mikucionis, M., Nielsen, B.: Online testing of real-time systems using Uppaal. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 79–94. Springer, Heidelberg (2005). doi:10.1007/978-3-540-31848-4_6
8. Larsen, K., et al.: Testing real-time embedded software using UPPAAL-TRON: an industrial case study. In: Proceedings of the 5th ACM International Conference on Embedded Software, pp. 299–306. ACM (2005)
9. Belli, F., et al.: Model-based mutation testing approach and case studies. Sci. Comput. Program. **120**, 25–48 (2016)
10. Aboutrab, M., et al.: Specification mutation analysis for validating timed testing approaches based on timed automata. In: 36th Annual IEEE Computer Software and Applications Conference, COMPSAC 2012, Izmir, Turkey, 16–20 July 2012, pp. 660–669 (2012)
11. Aichernig, B.K., Lorber, F., Ničković, D.: Time for mutants — model-based mutation testing with timed automata. In: Veanes, M., Viganò, L. (eds.) TAP 2013. LNCS, vol. 7942, pp. 20–38. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38916-0_2
12. Gargantini, A., Heitmeyer, C.: Using model checking to generate tests from requirements specifications. In: Nierstrasz, O., Lemoine, M. (eds.) ESEC/SIGSOFT FSE -1999. LNCS, vol. 1687, pp. 146–162. Springer, Heidelberg (1999). doi:10.1007/3-540-48166-4_10
13. Bengtsson, J., Yi, W.: Timed automata: semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004). doi:10.1007/978-3-540-27755-2_3
14. Čerāns, K.: Decidability of bisimulation equivalences for parallel timer processes. In: Bochmann, G., Probst, D.K. (eds.) CAV 1992. LNCS, vol. 663, pp. 302–315. Springer, Heidelberg (1993). doi:10.1007/3-540-56496-9_24
15. Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly, Sebastopol (2008)
16. Grinberg, M.: Flask Web Development: Developing Web Applications with Python. O'Reilly Media, Inc., Sebastopol (2014)

17. Iqbal, J., Truscan, D., Vain, J., Porres, I.: TRON2UPPAAL backtracer tool from TRON logs to UPPAAL traces. Technical report 1138, Turku Centre for Computer Science (2015)
18. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**, 649–678 (2011)
19. Lee, S.C., Offutt, J.: Generating test cases for XML-based web component interactions using mutation analysis. In: Proceedings of 12th International Symposium on Software Reliability Engineering, ISSRE 2001, pp. 200–209 (2001)
20. Li, J.H., Dai, G.X., Li, H.H.: Mutation analysis for testing finite state machines. In: Second International Symposium on Electronic Commerce and Security, ISECS 2009, vol. 1, pp. 620–624. IEEE (2009)
21. Lee, S., et al.: Automatic mutation testing and simulation on OWL-S specified web services. In: 41st Annual Simulation Symposium, ANSS 2008, pp. 149–156 (2008)
22. Wang, R., Huang, N.: Requirement model-based mutation testing for web service. In: 4th International Conference on Next Generation Web Services Practices, NWESP 2008, pp. 71–76 (2008)
23. Rusli, H.M., et al.: Testing web services composition: a mapping study. In: Communications of the IBIMA 2011, pp. 34–48 (2007)
24. Bozkurt, M., et al.: Testing web services: a survey. Department of Computer Science, King's College London, Technical report TR-10-01 (2010)
25. Rauf, I., Siavashi, F., Truscan, D., Porres, I.: An integrated approach for designing and validating REST web service compositions. In: Monfort, V., Krempels, K.H. (eds.) 10th International Conference on Web Information Systems and Technologies, vol. 1, pp. 104–115. SCITEPRESS Digital Library (2014)
26. Cambronero, M.E., et al.: Validation and verification of web services choreographies by using timed automata. J. Logic Algebraic Program. **80**, 25–49 (2011)
27. Dıaz, G., et al.: Model checking techniques applied to the design of web services. CLEI Electron. J. **10**, 5–11 (2007)

Publication VI

Vulnerability Assessment of Web Services with Model-based Mutation Testing

VI

Siavashi, Faezeh and Truscan, Dragos and Vain, Juri (2018). In *QRS 2018 – IEEE 18th International Conference on Software Quality, Reliability, and Security*, pages 301–312. IEEE Computer Society Conference Publishing Services

Vulnerability Assessment of Web Services with Model-based Mutation Testing

Faezeh Siavashi
dept. Information Technologies
Åbo Akademi University
Åbo, Finland
Email: faezeh.siavashi@abo.fi

Dragos Truscan
dept. Information Technologies
Åbo Akademi University
Åbo, Finland
Email: dragos.truscan@abo.fi

Jüri Vain
dept. Software Science
Tallinn University of Technology
Tallinn, Estonia
Email: juri.vain@ttu.ee

***Abstract*—We present a model-based mutation testing approach, for evaluating the authentication and authorization of web services in a multi-user context. Model of a web service and its security requirements are designed using UPPAAL Timed Automata. The model is mutated to create invalid behavior which is used for test generation to reveal faults in the system under test. The approach is supported by a model-based mutation testing tool, μUTA, that automatically generates mutants, selects a collection of suitable mutants for testing and generates test cases from them. We modify a previously defined mutation operator and introduce three new operators for additional mutants. We define criteria for the mutation-selection and demonstrate the approach on a blog web service. Results show that the approach can discover authorization faults that were not detected by traditional methods.**

***Index Terms*—Model-Based Mutation Testing, Timed Automata Mutation, User Behavioral model, UPPAAL, Security Testing**

I. INTRODUCTION

Popular social web services such as Facebook, Twitter, concentrate large amounts of sensitive data related to their users and are expected to be responsible for their integrity and security. One of the top security risks that are reported by OWASP is the incorrect configured user and session authentication which enables attackers to exploit passwords, keys, or session tokens, or take control of users accounts to assume their identities [1].

Authentication and authorization are the two main ways of securing a web service and the data it maintains. Authentication is the process of verifying a user's identity, while authorization is the process of proving user's permission to access resources provided by a web service. For example, once a user is signed in to a blog (authentication), she can manage/edit her posted articles but is not allowed to manage/edit other users' articles (authorization). Furthermore, access control and authorization in web services are often defined based on user's roles in a group setting. For instance, a blog's administrator may have permission to manage all posted articles, whereas other users do not. Such role-based requirements make implementation of security systems of a web service more challenging.

As defined in BS 7799-3:2017 standard, vulnerability is a weakness of an asset or group of assets that can be exploited by one or more threats, where an asset is anything that has value to the organization, its business operations and their continuity, including information resources that support the organization's mission [13]. To ensure that the security system of a web service is implemented correctly, all possible scenarios of user activities should be tested in an ideal case. Since manual testing is error-prone and mostly not exhaustive, model-based testing (MBT) has gained more attention by offering automated test generation from the model of a system under test (SUT) and its environment. As models can be designed based on specific test oracles such as robustness and security, they create test cases to validate the systems based on the test oracles. The test cases are executed against the SUT, and the test outputs are compared with the expected outputs.

Leveraging MBT by adding mutation testing leads to more powerful testing technique, known as Model-based Mutation Testing (MBMT) [11]. In MBMT, the original test model is altered systematically by *mutation operators* creating multiple versions of a model (known as *mutant* models). The mutants can be used for automatic generation of invalid test inputs that are executed against the SUT. The goal in MBMT is to find whether any invalid tests can pass the testing, thus they can reveal unexpected behavior (i.e., fault) in the SUT. Hence, MBMT can expose the mistakes that are caused by missing requirements or incorrect implementation.

The concept of MBMT itself is not new, however to our knowledge, the security requirements such as authentication and authorization have not been assessed using such technique yet. Available modeling and testing approaches focus on the integrity of the specification based on individual user activities and not multi-user interactions.

In our previous work on testing web services with a similar MBMT approach, the model of the SUT was defined for single user activities in a web service with the purpose of evaluating the robustness of the SUT [36]. However, the authorization and privileges for multiple users were not designed and tested. In this paper, we extend and improve the MBMT approach with three main contributions, as follows:

- we extend the previous approach to evaluating the vulnerability of web services in multi-user context;
- we improve the mutation-selection process to achieve more efficient mutants (i.e., the mutants that reveal

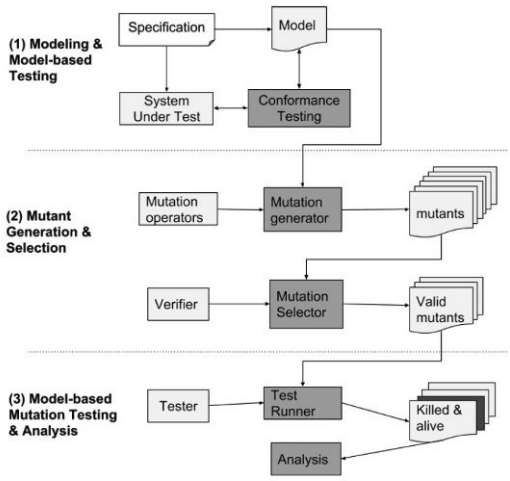


Fig. 1. Overview of our model-based mutation testing approach

faults);

- we introduce and evaluate three new mutation operators of Uppaal Timed Automata models and extend one of the previously defined mutation operators to create additional mutations.

We model a web service, its users and their authentication and authorization requirements. The model will be mutated to create invalid test inputs that target faults in the implementation of the web service. Besides, some mutation-selection criteria such as reachability are employed to provide more suitable mutants for testing.

Model-based mutation testing approach utilizes black-box testing for detecting vulnerabilities in web services, assuming that their source-code is not available.

Figure 1 presents an overview of the process. At first, a model is designed and verified by model checking. In the second step, mutants are generated by applying the mutation operators. In this paper, we select some suitable mutants from a list of mutation operators presented in [2] and [5], extend one of the operators and introduce three new operators to create further mutations. To increase the efficiency of MBMT, we apply a mutation-selection technique to the mutants and eliminates trivial mutants (i.e., mutants that are unreachable or incorrect). The selected mutants are called *valid* mutants and will be used for test generation.

In the third step, a test runner executes the valid mutants, mimicking possible faults in the SUT. If the SUT detects an invalid input, the corresponding mutant will be *killed*; otherwise, it will be *alive*. In the analysis of the results, the alive mutants will be assessed for detecting possible vulnerabilities in the implementation of the system. The efficiency of the mutation operators will be measured as well.

II. Preliminaries

We use Uppaal Timed Automata (Uppaal TA) as the base of modeling and mutating a web service. We first briefly describe TA and Uppaal TA formalisms and then review their formal definitions.

A. Overview of Timed Automata

Alur and Dill introduced the theory of Timed automata (TA) for modeling and verification of real-time systems [6]. TA are expressed as a set of locations and directed edges that can connect the locations to each other and extended with real-valued clocks. A timed automaton can execute individually or in synchrony with other automata. During execution of TA, all clocks increase with the same speed.

Clocks can be updated or reset along with transitions of TA. The transitions can be constrained by *guards* of edges and enable or disable transitions.

If there is more than one enabled transition at a time, then one of them will be chosen randomly. This characteristic provides more freedom to design non-deterministic behavior in the systems with random discrete events [20], such as real-time systems.

Definition 1. Timed Automata (TA)

Let C be a set of non-negative real valued variables of n clocks and $\mathcal{G}(C)$ be a set of guards on clocks that are conjunctions in form of $x \bowtie c$, where $x \in \mathcal{G}$, $c \in \mathbb{N}$, and $\bowtie \in \{\leq, <, =, >, \geq\}$. Let $v : C \rightarrow \mathbb{R}_{\geq}c$ indicates that a real value is assigned to every clock $c \in C$ and Let $\mathcal{U}(C)$ denote the set of *updates* of clocks. A timed automaton is a tuple (L, l_0, I, E), where:

- L is a finite set of *locations* and $l_0 \in L$ is *initial* location;
- $E \subset L \times A \times \mathcal{G}(C) \times 2^c \times L$ is a set of edges including an action, a guard and a set of clocks.
- $I : L \rightarrow \mathbb{G}(C)$ assigns location invariants.

A transition can be denoted by $l \xrightarrow{a,g,u} l'$, iff $(l, g, a, u, l') \in E$. A state in TA is defined in form of $s = (l, \overline{v})$, where l is a location and $\overline{v}$ is a non-negative clock value that satisfies the invariant of l. A TA progresses either by changing from a state to other by executing an edge,i.e., $(l, \overline{v}) \xrightarrow{a} (l', \overline{v}')$, or by staying in a location and passing time, i.e, $(l, \overline{v}) \xrightarrow{d} (l, \overline{v} + d)$, as long as the invariant of location l is true.

Definition 2. Timed Input-Output Automata (TIOA)

Timed Input-Output Automata (TIOA) was introduced as extensions of TA, in a way that the actions set, A is divided into two sets of inputs and outputs actions, A_i and A_o respectively [22]. The input actions model the behavior of the environment and output actions model the external actions of the system. Thus, for each input action in the system, there is an output. A TIOA, A is a tuple $< I_A, O_A, L_A, l_A^0, C_A, T_A >$, where:

- I_A is a finite set of inputs, labeled by "?", O_A is a finite set of outputs, labeled by "!",
- L_A is a set of locations that indicates the state of the system after the transition,
- l_A^0 is the initial location,
- C_A is a set of clocks instantiated to zero at l_A^0, and
- T_A is a set of transitions in the system.

The theory of TIOA is implemented in modeling frameworks such as UPPAAL.

B. Overview on UPPAAL Timed Automata & UPPAAL-TRON

UPPAAL is widely used modeling and verification tool for real-time and reactive systems. The tool and its formalism were introduced as Ph.D. thesis [31]. It extends TA with other data types in addition to clocks. In UPPAAL , due to the distinction between local and global variables, it is possible to model systems and their environment as separate interacting automata. Such functionality enables refining the specification of either of a system or the environment without having a significant change in the other. Moreover, various testing goals can be designed in the environment such as safety, robustness, user scenarios.

Definition 3. Uppaal Timed Automata

A UPPAAL TA model is a network of n timed automata that share variables, clocks and actions. A UPPAAL TA model A_i is a tuple $< L_i, l_i^0, C, A, E_i, l_i >$, $1 \leq i \leq n$.

A UPPAAL TA model of a system can be analyzed if a particular criterion will be satisfied during the execution of the model. This analysis is done by defining reachability properties in UPPAAL TA models as explained below:

In UPPAAL a model of a system and its environment can be synchronized by *channels* over edges. Channels are labeled by "!" as emitting and "?" as receiving. Thus, UPPAAL TA leverage modeling complex systems by supporting parallel transitions from different automata.

Reachability Analysis in UPPAAL TA

Reachability analysis in UPPAAL TA is implemented as a finite *symbolic state space* exploration by executing *symbolic computation steps*. A symbolic state denoted as (l, D), where l is a location of a timed automaton and D, clocks valuations, represents $\{(l', \overline{v}) | l' = l \wedge \overline{v} \in D\}$. The initial state of the automaton is (l_0, D_0), where $D_0 = \{\overline{v} | (l_0, \overline{v_o}) \xrightarrow{d} (l_0, \overline{v})\}$. The reachability for symbolic state indicates that the location l is reachable at some point of the time. A symbolic computation step is defined in the form of $(l, D) \xrightarrow{a} (l', D')$, representing an action followed by some delay. An action is reachable iff $(l, \overline{v}) \xrightarrow{a} (l', \overline{v}')$ and $D' = \{\overline{v}'' | (l, \overline{v}) \xrightarrow{a} (l', \overline{v}') \wedge (l, \overline{v}') \xrightarrow{d} (l'', \overline{v}'') \wedge \overline{v} \in D\}$.

Beside reachability, in UPPAAL TA, safety, deadlock-freeness and liveness properties also can be defined. The UPPAAL model checker contains an engine for verifying such properties [23].

Once a model of a system is verified based on its specification criteria, it can be used for validation of its actual behavior. UPPAAL utilizes an online testing tool, TRON which generates test cases from UPPAAL TA models and executes them against systems.

Conformance Testing With UPPAAL TA

UPPAAL TRON generates symbolic timed traces in UPPAAL TA models. A symbolic timed trace $TTrS$ of a UPPAAL TA model is a sequence of symbolic states, each state being defined as a tuple (l, D, v), where l is a location, D is the clock constraints and v a set of non-negative variables' values. Similar to the TA progress described above, a transition in UPPAAL TA from a symbolic state to another is possible either by an action or by some delay $(l, D, v) \xrightarrow{a/d} (l', D', v')$.

UPPAAL TRON takes environment constraints of a system into account. Thus, the model of a system and its environment are defined as TIOAs, where the model is split into two parts of the SUT and the environment that are synchronized by input/output actions. The interaction between the system and its environment are identified as observable actions in UPPAAL TRON. The actions among the SUT (or environment) with other systems are known as internal actions and are not observable. During the execution, these internal actions are abstracted as delays by UPPAAL TRON. Therefore, conformance testing contains delays and observable actions.

Definition 4. Relativized Timed Input/Output Conformance (rtioco)

For input enabled timed input/output labeled transition systems $i, s \in \mathcal{S}$ and $e \in \mathcal{E}$, relativized timed input/output conformance is defined asl below:

$$i \ \mathbf{rtioco}_e \ s \ , \forall \sigma \in \mathtt{TTr}(e) . \mathbf{Out}(\langle i, e \rangle \ \mathbf{after} \ \sigma) \subseteq \mathbf{Out}(\langle s, e \rangle \ \mathbf{after} \ \sigma),$$

where $\mathcal{S}$ and $\mathcal{E}$ are TIOAs with observable inputs and outputs, i, s and e are initial states of the implementation under test, specification and environment respectively. $\mathtt{TTr}(e)$ is a set of timed i/o traces of the environment e, $\langle i, e \rangle$ and $\langle i, s \rangle$ are observable i/o actions that are synchronized. $\langle i, e \rangle$ **after** σ indicates that observable trace σ is executed on implementation i via environment e and $\langle i, e \rangle$ **after** σ means that an observable trace σ is evaluated on the specification s via environment e and returned a set of possible states. $\mathbf{Out}(states)$ contains a list of possible output actions or delays.

The practical implication of the previous definitions is that the implementation conforms the specification within a shared environment if and only if the observable i/o behavior in the model is always the same as the behavior of the implementation. Thus, the result of conformance testing with UPPAAL TRON will be one of the three cases: *passed*, *failed*, or *inconclusive*. When the specified behavior in the model conforms to the implementation, the test will pass; otherwise, it will fail. If the output is not in the set of inputs for the environment or no input/output is provided within the defined time (test timeouts), then the test result is interpreted as inconclusive.

C. Model-Based Mutation Testing

Mutation testing extends the fault detection capabilities of MBT by exposing more vulnerabilities of systems. It changes a system's program or its specification, to create new versions of the system (mutants). Mutation operators are rules that establish the mutants by altering the syntax of the program (or the specification). The syntax alternations usually result in different behavior in mutants. In model-based mutation testing, the mutants are generated from the test model of the SUT. The tests exhibit altered inputs (incl. faulty inputs) and input sequences against the SUT. Hence the mutants allow testing of the SUT with invalid.

Although mutation testing has shown to be more efficient than other test criteria [30], it suffers from a large number of mutants that are not suitable or have equivalent behavior to their original specification/program. Creating and executing all mutants are usually costly. In this paper, we follow the mutants selection principles, reachability, infection, and propagation (RIP) for killing mutants. The RIP model was primarily defined for code-based mutation testing [7]. We adopt the RIP model for model-based mutation testing. To kill a mutant:

1) It must be **reachable**, which means that the mutated part of the model is executed at some point of test run,
2) It must cause **infection** (i.e., changes the state), on the mutant model after the mutation is visited,
3) It can **propagate** the mutation through the model (i.e., the difference in the behavior of the mutant is observable).

If the reachability and infection are satisfied by a mutant, it is called *weak mutant* and if all conditions are met, it is called *strong mutant*.

In our MBMT approach, during mutant selection, we check whether each mutation is reachable and potentially infects the SUT during test execution. These conditions help to eliminate unfit mutants. In UPPAAL TA, reachability, liveness, and deadlock-free properties can be defined on states and paths. We define reachability criteria based on the model elements where the mutation is applied.

1) RIP condition for mutants in UPPAAL *TA:* In the context of UPPAAL TA, if a mutation applied on an edge, guard, or update, the **reachability** for *symbolic computation step* is defined using a global variable in the model and update its value on the action the mutation occurs. Let $m \in \overline{v}$, be a boolean variable that is initialized to zero and action a is a mutated action. A mutated action is reachable iff in $(l, \overline{v}), m = 0$ and in $(l', \overline{v'}), m = 1$, and $(l, \overline{v}) \xrightarrow{a} (l', \overline{v}')$ and $D' = \{\overline{v}'' | (l, \overline{v}) \xrightarrow{a} (l', \overline{v}') \wedge (l, \overline{v}') \xrightarrow{d} (l'', \overline{v}'') \wedge \overline{v} \in D \wedge m = 1\}$

After a mutation is reached, if input/output actions are different from the original input/output actions, then it means that the mutation changes the original state and causes an **infection**. One way of detecting the infection is to apply *bisimulation* relation, which compares traces of the original model and its mutants and checks whether they are equivalent. We used this technique in [37]. The mutants that pass the reachability and infection conditions will be considered as valid mutants. Aichering and Jöbstl introduced such condition regarding refinement relation in [5].

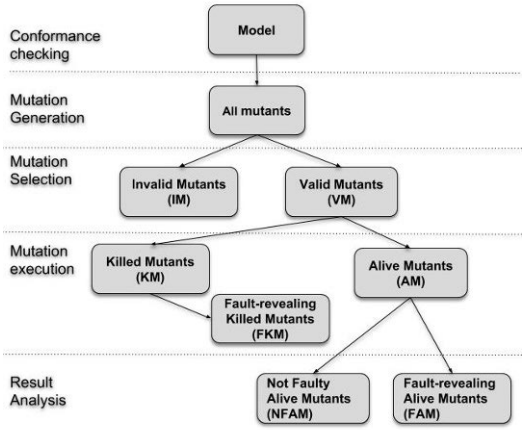


Fig. 2. Mutant classification in our approach

Mutated test cases are generated from the valid mutants the online testing tool UPPAAL TRON. The **propagation** condition can be checked during the test execution by comparing the observable behavior of the mutants with the behavior of the SUT. Since UPPAAL TRON evaluates the test results based on $rtioco$ relation, it is a proper tool for detecting propagation of the mutations at runtime.

If the mutation causes a change in observable input/output or the delays, and it can be detected during the test generation, then it considered as killed. Otherwise, if the SUT provides test outputs to the mutated test stimuli, then the mutated test case will not be detected and the mutant will be alive. The reason is that the SUT might be more permissive than its specification.

2) Mutation Classification in MBMT: In [11], Belli et al., an extended classification of the mutants for MBMT after executing them against the SUT. Such designation elevates the quality of testing and distinguishes whether the faults are raised by the mutants or by poor implementation of the SUT. In this study, however, we classify the mutants during mutation generation. The reason behind this is to identify more suitable mutants from the beginning and reduce the time of test execution and thus attain more efficient testing.

Figure 2 gives the classification in each step of our MBMT approach, starting from the mutation generation until the test analysis. After the model is designed and its conformance with the implementation is approved, we use the mutation operators to generate all possible combinations of mutants. Then, the reachability and infection properties of the mutants will be checked. The outcomes of this step are as follows:

- An **invalid** mutant model is either incorrect syntactically, or does not satisfy the mutation selection criteria (reachability and infection). Syntactically incorrect mutants are known as stillborn mutants.
- A **valid** mutant model must be a syntactically correct model and satisfy the mutation selecting criteria (reachability and infection).

Valid mutants are used for generating mutated test cases. During test execution they will be classified further as follows:

- A **killed** mutant model is a mutant which does not conform to the behavior of the SUT resulting in a failed or inconclusive verdict during testing. The killed mutants can be either trivial (killed by every test) or non-trivial, but in both cases, they are considered not interesting and thus eliminated.
- An **alive** mutant is a valid mutant that generates faulty behavior that cannot be observed or distinguished from the behavior of the SUT.
- If a mutant is killed but also reveals an anomaly in the SUT, then we count it as a **fault-revealing killed** mutant.

Since infection criterion ensures that the mutants are not equivalent to the original model, we do not have any equivalent mutants. Analyzing alive mutants classifies them as follow:

- If the SUT behaves the same as a mutant, then there is a fault in the implementation of the SUT. Such mutant is called **fault-revealing alive** mutant.
- If a mutant's behavior does not harm the SUT and does not cause any change in the state of the SUT, then it is known as a **not faulty alive** mutant. Distinguishing between such mutant and fault-revealing alive mutants should be checked by manual inspection.

III. Modeling Web Services with Uppaal Timed Automata

As mentioned earlier, Uppaal TA can be used for modeling behavior of a system as well as its environment, which includes behavior of other systems and users that interact with the system under test. The interaction between a system and its environment is via observable channel synchronizations in the Uppaal TA model.

First, we demonstrate a simple communication between a user and a web service with an excerpt of the actual model and then we explain how the model is extended for multi-user communication and their security specification.

Figure 3 shows two automata, a user and a web service. The automata are synchronized via channels following the request-response paradigm specific to web services. When a user wants to post a new article in a blog, she sends a request to the web service by clicking corresponding button or link in the browser. For each HTTP request, a channel is defined emitting from `User1` and receiving in `Blog`. The response to the request corresponds to a channel synchronization in the opposite direction, i.e., emitting from `Blog` and receiving in `User`. To edit and article, the user first gets access to the article ($manage\_article$) and then she submits the changes ($edit\_article$).

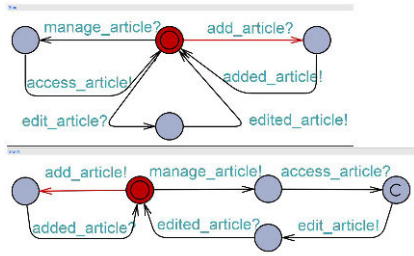


Fig. 3. Excerpt model of user interaction with a Blog

The extended version of model consists of `Blog`, `User1` and `User2` automata which describe multi-user interactions in a blog web service as shown in Figure 4. For demonstration, a limited number of user activities are modeled. From a user's point of view, the activities are limited to posting articles, commenting, reading, editing and deleting. In the web service's side, the events are in the form of HTTP requests that trigger corresponding functions creating or modifying resources or interacting with other systems.

The users automata are modeled with two different *user-id*s. The model contains some shared resources for tracking articles and comments and are accessible by both users. Each user can create new articles (add comments to the articles), or have access to available resources arbitrarily. The shared resources are defined as follows:

- *cu, sa, su* and *sc*, indicate "current user", "selected article", "selected user" and "selected comment" respectively. These variables enable random selection of users, articles and comments.
- *Users*, *max_art* and *max_cmt* variables indicate the number of users, maximum number of articles and maximum number of comments, respectively. Bounding the maximum numbers in the model prevents the possibility of a state space explosion.
- $article[Users][max\_art]$ assign articles to the users, thus, *article[2][3]* means that each user can create up to three articles. For instance, *User1* can create three articles which are mapped to *article[0][0], article[0][1] and article[0][2]*. The initial values are set to zero.
- $Comment[Users][max\_art][max\_cmt]$ contains data about the *user-id* of the person who comments on the articles. Thus, when *User1* (with id=0) adds a comment on *article 1* of *User2*, then we have *Comment[1][0][0]=0+x*. The constant value x is an offset number that prevents confusing the initial values (zero) with the *user-id* (id=0).

The $article$ and $Comment$ variables are updated by functions $addart()$ and $delart()$, $addcmt()$, and $delcmt()$.

As it is shown in user automata, the values of *user-id* and *article* are selected randomly:

$$u : int[0, Users - 1], a : int[0, max\_art - 1]$$

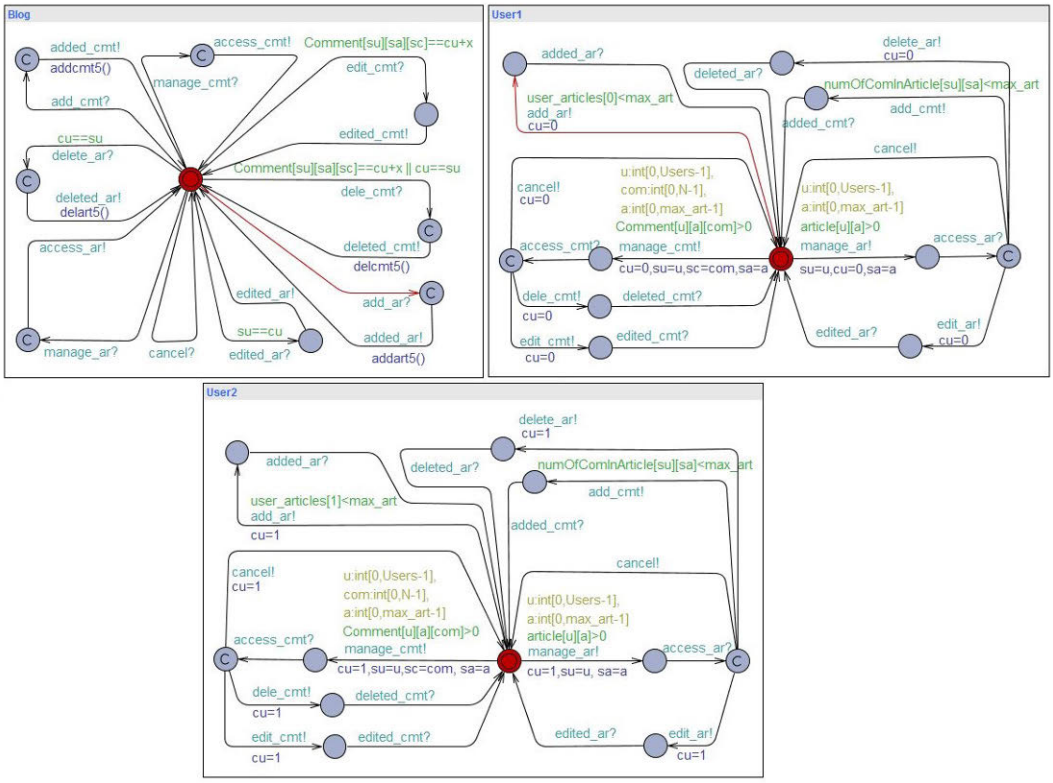


Fig. 4. A UPPAAL TA model of interactions of two bloggers within a Blog

These values will be assigned to *su* and *sa* variables identifying the *selected user* and *selected article*. For instance, if u and a are zero and 1 and if the selected article exists (i.e., guard: $article[0][1] > 0$), then *manage_ar* can be executed. The model is non-deterministic and either of the two users can start sending requests.

General security requirements are extracted from the requirements and are presented in Table I. In each request, the user credentials should be sent to the web service and verified. The authentication is specified in the user automata by updating the *cu* variable, which is used as guard conditions in `Blog`. Besides, only the owner of a resource has the right to delete and edit his comments or articles. Therefore, in `Blog`, the *cu == su* guard is used to check this condition.

For instance, the first requirement in Table I includes an authentication condition, i.e., *the user should be verified*, and an authorization condition, i.e., *the user is the owner of the article*. The authentication condition is defined in such a way that the model updates the shared variable `cu` (i.e., *cu=0* or *cu=1*). When $dele\_cmt$ is requested, in `Blog` the authorization is defined by comparing the owner of the request and the owner of the article (*cu*). The way in which the security requirements of the model have specified allows us to scale the model up for verifying the behavior of more than two users.

IV. MODEL-BASED MUTATION TESTING WITH μUTA

As shown in 1, our approach on MBMT includes three main steps. Once the test model conforms with the SUT (i.e., step (1)), it can be used for generating mutants. In this section, we describe step (2) and step (3) of the approach, namely *Mutation Generation and Selection*, and *Model-based Mutation testing and Analysis*. The μUTA tool automates step (2) and, partially, step (3). The analysis of the test results is done manually.

A. Mutation Generation & Selection

The idea of generating mutants for TA for testing the dynamic behavior of real-time systems was first presented by Nilsson et al. in [29]. Then, various mutation operators on timed automata elements were formally defined by Aboutrab et al. [2] and Aichernig et al. [3]. From the the previously

TABLE I
SECURITY REQUIREMENTS FOR A WEB LOG

Security requirements	Authorization	Authentication
A user can delete any comment under his article.	Guard in Blog $(comments[su][sa][sc] == cu)$	update in User1/User2 (cu=0, or cu=1)
An **article** can only be edited/deleted by its owner.	Guard in Blog (cu==su)	”
A **comment** can only . be edited by its owner	Guard in Blog (cu==su)	”
A **comment** can only be deleted by its owner OR by the owner of the article.	Guard in Blog $(Comments[su][sa][sc] == cu \|\|cu == su)$	”

TABLE II
MUTATION OPERATORS (* NEW OPERATORS)

Element	Mutation Operator	Description
Action	CN	Change names of actions
	CT	Change targets of actions
	CS	Change sources of actions
	RA*	Remove actions
	DA*	Duplicate actions
Guard	RG*	Remove guards
	CGL	Change guards logical operators
	CGV	Change guards variables

proposed operators, we have selected the ones shown in Table II, which apply to *actions* and *guards*. However, the mutation operators for *invariant* and *locations* are not selected because they are not applicable in this particular case study.

- **Change Name of actions (CN)** – replaces the name of an action with the name of other actions in the model. Thus, the expected sequence of the inputs to the implementation will be different.
- **Change Targets of actions (CT)** – changes the target location of an action to another location in the model. This operator breaks the flow of test inputs and violates the state of the model. Both input and output actions can be mutated by this operator.
- **Change Sources of actions (CS)** – changes the source location of an action to another location. Similar to CT, this operator mutates the sequence of input/outputs.

In addition, we introduce three new mutation operators as follows:

- **Remove Actions (RA)** – randomly deletes one action at a time and creates a mutant. Omitting an action will manipulate the sequence of input/output actions.
- **Duplicate Actions (DA)** – randomly copies an action in different parts of a model, thus alternates the sequence of inputs and outputs by repeating actions in unexpected states of the model.
- **Remove Guards (RG)** – randomly selects an action and removes its guard. Actions that are mutated by RG will be always enabled.

Previously, the mutation operators on guards were defined to negate conditions, in [3], or to alter timing constraints [2]. We modify this operator and split it in two new operators:

- **Change Guards Logical operators (CGL)** – changes logical operators (i.e., $==$, $<=$, $>=$, $!=$, $<$ and $>$) in guards.
- **Change Guards Variables (CGV)** – alters values of the variables that are used in guards and creates additional mutants that cannot be defined by other mutation operators.

μUTA implements the mutation operators which generate mutants from the given UPPAAL TA model. Distinguishing between the valid and invalid mutants with the tool is done by defining some model-checking conditions. The tool uses *verifyta* [9] to select valid mutants. Verifyta is an UPPAAL interface that can verify an UPPAAL TA model based on given list of conditions that are formalized as queries. The mutants are killable if they do not satisfy at least one of the following conditions:

Deadlock-freeness and livelock-freeness – If the mutants have deadlock or live-lock, they are killable. Deadlock means that a system enters to a state where no further action is enabled, while live-lock means that a system reaches to a condition that continually switches among some states forever, without any progress. UPPAAL supports verifying deadlock-freeness, however, livelock-freeness for mutants are automatically defined by μUTA.

Mutation reachability – As we described in Section II-C, for the mutation operators that alter actions (i.e., CN, CT, CS, and DA), we apply symbolic computation steps. In UPPAAL a boolean variable is declared and initialized to `False`, and on the mutated action, it will be updated to `True`. Therefore, if the action is fired, then the variable will be set to `True`. Consequently, the reachability of the mutated action can be checked symbolically during simulation.

input/output traces – To ensure that the input/output trace in the mutant is different than input/output trace of the original model, we create a set of traces that contain all edge coverage and all location coverage. If a mutant model does not follow the traces, then it is considered as invalid; otherwise, it is valid. The comparison is automatically done via *bi-simulation* following the approach presented in [37].

B. Mutation Testing & Analysis

In the last step of the approach, we use the valid mutants for test generation against the `Blog` web service and analyze the results. A tester sets up individual test sessions using the UPPAAL TRON testing tool. The test execution from the model-level inputs to actual HTTP requests is established by a test adapter, which converts input/outputs actions into their corresponding requests/responses. A HTTP request will be sent to the `Blog` web service as an URL, and UPPAAL TRON waits for the response from the web service. The test adapter converts the response into a receiving action in the model. The result of each test will be categorized into alive and killed mutants, based on the verdict of the test session.

The analysis of the result is a manual process. Distinguishing whether an alive mutant addresses a genuine bug or it is in fact an equivalent model is yet a manual process. Moreover, not all possible bugs that are found during the mutation analysis will reveal vulnerabilities in the implementation or indicate wrong specifications. A bug proves that the expected and actual behavior do not conform, while vulnerability is a specific bug that manifests the possibility of exploiting the system under test. Therefore, deciding whether an alive mutant is a bug, which reveals vulnerability should be done manually based on the experience of the tester. Defining useful mutation operators has a significant impact on computational time and test effort. Typically, mutation score is a standard for evaluating the mutation operators.

Mutation Fault Detection The ratio of mutants (killed or alive) that reveal faults in the SUT to the number of generated mutants is measured by

$$MFD = \frac{FAM_i + FKM_i}{VM_i},$$

where FAM_i indicates the number of fault-revealing alive mutants, FKM_i is the number of fault-revealing killed mutants, and VM_i is the number of all the valid mutants generated.

Mutation testing has some fundamental problems. One of the problems is caused by having identical mutants generated by two or more mutation operators. Redundant mutants not only increase the test generation and test execution time but also have an impact on the validity of the assessment. However, this problem is tackled in our approach. The mutation operators in the μUTA tool are carefully designed, implemented and tested to prevent generating redundant mutants. Each mutation operator is implemented in such a way that provides unique mutants.

Another main problem is the equivalency. Equivalent mutants are those that are behaviorally equivalent to the original model. In code-based mutation testing, equivalency has been proven to be undecidable. However, in timed automata, it can be prevented. Our technique for solving such problem is to compare input and output traces of each mutant with the original model.

V. EVALUATION

A. Blog Web service

We selected the Blog web service as an example of a web service that provides multiple user interactions and supports some of the general security requirements such as authorization and authentication. The Blog web service is designed in REpresentational State Transfer (REST) [34] architectural style and provides functionality for creating new user accounts, posting new articles, commenting, deleting/editing posts and comments, managing user's profile, similar to common social networks. The web service is implemented in Python using Flask web developing micro framework [19]. We chose Flask as it has a simple and flexible structure which does not restrict developers to specific formats. This feature, however, makes the web applications prone to error and an interesting topic for testing.

B. Tool chain

Our MBMT approach is supported by a set of tools that automate the mutation generation, selection and execution procedures. The UPPAAL model-checker is used for modeling and verification the original test model. For online conformance testing, we use UPPAAL TRON.

μUTA tool contains some components: a *generator*, a *selector* and a *test runner*. The generator systematically creates mutants form a given UPPAAL TA model based on given set o mutation operators. The selector is a component that creates model-checking rules for each mutant and utilizes the *verifyta* tool [9] to validates them. If the rules are satisfied by verifyta, then the mutants will be classified as valid, otherwise discarded (i.e., invalid mutants). The test runner sets up test sessions using UPPAAL TRON, orchestrates the execution of valid mutants against the implementation of the SUT, and synthesizes the test results in a report.

To execute test cases that are generated by the mutants, we developed a test adapter, which translates the model-level test cases into test scripts that create HTTP requests that include test inputs. The responses from the web service under test will also translated by the test adapter into model-level responses.

The μUTA tool is developed using Python language and the test adapter is developed in Java language.

C. Results

The results of the MBMT approach for the Blog case study are presented in detail in Table III. From the `Blog` model, in total 2962 mutants were generated, whereas only 138 were valid for testing (i.e., VM). The generation and validation process took 24 hours and 44 minutes on a PC running Windows 7 Enterprise 64-bit operating system with Intel quad-core CPU, and 16 GB RAM. The test execution time for each valid mutant set to 150 seconds. It took about 200 minutes to run all the valid mutants against the SUT.

TABLE III
MODEL-BASED MUTATION TESTING PROCESS AND THE RESULTS.

	Mutation Generation and Selection			**Mutation Testing**			**Analysis**		
Operators	time	#of IM	#of VM	#of KM	#of FKM	#of AM	# of FAM	#of NFAM	MDF
CN	0:51:24	302	2	2	0	0	0	0	0
CT	01:18:22	160	20	12	0	8	0	8	0
CS	01:08:34	171	9	1	8	0	0	0	88.9%
RA*	00:02:05	9	9	9	0	0	0	0	0
DA*	18:43:57	2099	79	78	0	1	0	0	0
RG*	00:24:30	3	1	1	0	0	0	0	0
CGL	0:36:08	8	12	12	0	0	0	0	0
CGV	1:39:00	210	6	4	0	2	2	0	33.3%
Total	24:44:00	2962	138	119	8	11	2	8	7.2%

In total three different defects are found, one of them is revealed during the test execution and two others are found during the analysis.

From 138 valid mutants, 119 mutants were killed during the test execution (i.e., KM), eight mutants caused a crash in the SUT during the test and by investigating them, we found that all of the crashes belong to a bug in the implementation. Therefore, they are labeled as fault-revealing killed mutants (FKM).

Eleven mutants passed the test execution and remained as alive mutants (AM). All FKMs were generated by the CS operator and revealed a vulnerability in the implementation of the SUT. From eleven AMs, eight of the mutants were generated from the CT operator, two from CGV and one from DA, respectively.

The alive mutants were investigated by comparing their behavior to the original model, and whether they were representing any vulnerabilities. Deciding whether an AM shows faults (FAM), or does not create a fault (NFAM) is done manually. For instance, if a mutation changes the input/outputs in a way that the mutant does not generate additional behavior, then the mutant is NFAM. For example, in the `User1` automaton, a mutant is generated using the CT operator, which changed the target of `access_ cmt?` edge to the initial location, then the actions `dele_ cmt!`, `edit_cmt!` and `cancel!` cannot be executed (they are not accessible via the `access_ cmt?` action), thus the mutant is not able to generate the same traces as the original model. However, the mutant's traces will not be faulty and it will be not faulty alive mutant (NFAM).

To this extent, we investigated the AMs and found that none of the eight alive mutants generated by the CT operator were able to create faulty behavior and thus they were counted as not faulty alive mutants (NFAM). Similarly, the single alive mutant generated by the DA operator did not address any vulnerabilities in the SUT and was counted as a NFAM. The two CGV mutants, however, were able to reveal two distinct vulnerabilities in the SUT and were labeled as faulty alive mutants (FAM). The vulnerabilities were caused by mistakes in two separate parts of the implementation and are explained below.

D. Describing the bugs and identifying vulnerabilities

As described in Table IV, there are three bugs that were detected by the MBMT approach.

The first bug was detected during the test execution. The mutations were applied on the "delete" request for non-existing resources. The expected behavior of the SUT in this situation is to reject the request by a standard HTTP response, such as "`404: Not Found`". It can be done by adding a condition in the source code to check if the resource is available. However, this condition was missing in the code. Thus, invalid delete requests could crash the execution of the SUT. Eight FKMs generated by CS revealed the same bug.

In the analysis of the alive mutants, we detected two other bugs that show incorrect authorizations in two different HTTP requests: deleting articles and editing comments. In both cases the expected behavior of the SUT is first to verify whether the user is the owner of the resource, however, since such condition was missing in the source code, the SUT wrongly allows unauthorized users to modify resources. Two FAMs generated by CGV detected these bugs.

Vulnerabilities are the specific bugs that can exploit the system under test. If the bugs can cause abusing the system or unintended behavior occur in the system, then they are the system's vulnerabilities. Therefore from the bugs that we detected, we concluded that all three bugs are Blog vulnerabilities.

E. Efficiency of MBMT

Mutation Fault Detection (MFD) score is presented in the last column in Table III. CS has the highest score of 88.9%, followed by CGV with 33.3%, whereas other operators did not reveal any fault and got zero scores. The result shows that having a large number of mutations may not necessarily provide better results. For instance, DA generated the highest number of valid mutants. However, all of them were killed during the test execution. In contrast, CS has only nine valid mutants, which eight of them revealed one fault.

F. Testing Effort

To evaluate whether applying the reachability criterion is efficient regarding mutation generation and testing effort, we conducted two experiments on the same case study: (1) MBMT without verifying mutations' reachability properties,

TABLE IV
DESCRIPTION OF DETECTED VULNERABILITIES

Fault description	Operator	on which step
The SUT stops working when it receives a *delete* request to an non-existing article	CS	during testing
The SUT allows the user to delete an article without authorization check	CGV	during analysis of alive mutants
The SUT allows the user to edit Comments without authorization check	CGV	during analysis of alive mutants

and (2) MBMT including verification of mutations' reachability.

In the first experiment, the time of the mutation generation was roughly 120 hours, which is five times more than the second experiment. Moreover, the number of valid mutants for testing in the first experiment was 348, whereas in the second experiment we had 138 valid mutants. The test execution and analysis of the results in both experiments were similar. This comparison confirms that applying the reachability criterion significantly reduces the time of mutation generation and selection and consequently the time of test execution.

G. Evaluation of Mutation Operators

In this work, we extended one of the previously defined mutation operator to create mutants on values of variables in the guards (CGV) and were able to detect two authorization defects. The results suggest that additional modification on the mutation operators may provide more efficient mutants.

The new mutation operators (RA, DA, and RG) offer a large number of valid mutants for testing. However, none of the mutants were able to reveal any fault. The mutation operator for removing actions (RA) simulates a condition in the web service where there is no response to a request or there is a response to a non-existing request.

The mutation operator for duplicating actions (DA) suggests that a request (or a response) will appear unexpectedly. Finally, removing guards (RG) creates a test model in which mutated actions do not have guards, thus they are always enabled.

Despite none of the new mutation operators were fault revealing in our case study, they still provide new faulty behavior that cannot be created by any other mutation operators. Nevertheless, these mutation operators can deliver valuable mutants and should be investigated more.

H. Relation Between Mutations and Real Faults

To understand what kind of faults can be simulated by certain mutation operators, we compare the faults with the corresponding mutations. The first fault in Table IV is caused by the changing source of a transition (CS). It shows that deleting a resource was not properly implemented in the source code. The source code is implemented in such a way that it does not have the necessary condition of checking the availability of a resource, before deleting it. Such condition should always be included in the source code. Thus, CS can show lack of such condition by breaking the test sequences.

The second and third faults are caused by Change Guard Variables (CGV). The correlation between the faults and the mutation operator is straightforward: the operator changes the guards' values which simulate changing user's id in HTTP requests. Changing guards enables some transitions that otherwise are disabled.

VI. RELATED WORK

Due to the importance of testing security of web services, many studies and tools have been proposed in the literature. To draw the position of our research among the available studies, first, we describe related model-based mutation testing approaches, then we present some of the most related work on the model-based design and testing of web services and compare the available studies with our work, and then we focus on security testing approaches of web services.

A. MBMT approaches and improvements

MBMT has been gaining attention in software testing field, especially safety and security applications. Apart from theoretical research and experiments, one of the keys to making mutation testing available tool support. In a recent survey conducted by Papadakis et al. [30], available model-based mutation testing approaches are discussed. We only review some of the tools that are more similar to our work.

One of the well-defined tools is MoMuT that supports automatic mutation of different modeling languages, such as UML statecharts and Timed Automata [4], [3]. MoMuT for UML contains mutation operators for a large number of elements in UML. It only supports non-deterministic models. Thus test cases are linear sequences of inputs and outputs. MoMuT for TA uses UPPAAL TA models and applies timed input-output conformance (tioco) check between the specification and mutants. The conformance checking is done via SMT solver Z3.

Larsen et al. present an efficient technique of MBMT using Ecdar tool, which belongs to UPPAAL family [24]. The tool creates a strategy for refinement check for MBMT. The tool supports only deterministic models.

Belli et al. present MBMT with directed graphs using only two elementary mutation operators (insertion and omission) [11]. They describe the test algorithms and mutation generation technologies that are used to experiment with two different case studies. Their work is supported by a chain of tools including an event sequence graph (ESG)-based mutant and test set generator (MTSG). SIMULTATE is a toolset that uses fault injection technique on Simulink models to perform mutation analysis for certain parts of systems[32].

Several optimization techniques have been studied to reduce the cost of test execution. Devorey et al. presented featured mutants model (FMM) [15]. Their technique significantly reduces the time of testing by integrating multiple executions into a single one.

B. Modeling and Testing of web services

Model-driven development is a standard way of developing web services and can be used both for design and generation tests. Modeling web services has been presented by formal verification methods, model-checkers, specification languages, and theorem proving. Bozkurt et al. investigated a comprehensive survey on testing web services presenting available web service testing strategies [12].

Typically, web services are modeled with specification languages, simulated, verified and tested. Majority of the studies use model-checkers to verify the correctness of specifications, while some of the studies use models for verification as well as test generation. We review the studies that are similar to our approach.

Model-based testing of web services with symbolic transition systems (STS) is introduced by Frantzen et al. [18]. Belli et al. presented an event-oriented approach for MBT of a composite of web services in [10] and described expected behavior as a positive model and generated tests. They also defined some possible fault scenarios as a fault model which specifies undesired situations in communications of the web service composition and used for negative testing.

Modeling behavior of web services and their compositions are studied using UML [8], [28]. In order to automate the test generation, UML model of web services has been transformed into executable models. For instance, UML to UPPAAL TA transformation has been deployed in [16], [14] and [33].

In [33], we specified model of a web service composition using UML state diagrams and verified the requirements. Then, we transformed the model into UPPAAL TA models and generated the tests from the UPPAAL TA. In this study, however, we used the UPPAAL TA model to generate mutants to assess the vulnerabilities of the web services. Besides, in this paper, we focus on evaluating the authorization and authentication of web services.

C. Testing security of web services – different approaches

Security of web services is one of the fast evolving subjects in software testing. The reason behind such challenging issue is that the complexity of web services especially online social networks is growing fast. Mistakes in implementing and defining user credentials in web service are still one of the most common faults that are reported.

Several studies have been done on testing security in web services using fault injection, cross-site scripting (XSS) or modeling attacks scenarios. Salas and Martins presented a new approach to analyzing the robustness of Web Services by fault injection [35]. In their approach, attacks are simulated which can be used for evaluating the penetration on the web services. However, the process is not automated.

Dragoni and Massacci presented a framework which establishes communications between client and server based on the defined privileges as well as behavioral constraints [17]. They showed that the framework works as expected against both cooperative and malicious behavior.

A comprehensive analysis is done on all available mutation testing method presenting the current state of the art in this field and the open challenges [21]. Lee and Offutt [26] introduced an Interaction Specification Model which formalize the interactions among Web components. They defined a set of mutation operators for XML data model to mutate the inputs of the Web components. Li and Miller presented mutation testing methods using XML schema to create invalid inputs [27]. Mutation testing is extended to XML-based specification languages for Web services.

Lee et al. presented ontology-based mutation operators on OWL-S, which is an XML-based language for specifying semantics on web services [25]. They mutate semantics of the specifications of their case study such as data mutation, condition mutation.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we described three new improvements on our MBMT approach. Our approach includes the specification of a web service model in UPPAAL TA, generating mutants from the model, eliminating the trivial mutants and generating mutated tests. We presented the μUTA tool, which automates the mutation generation, mutation-selection, and mutation execution. As one of the improvements, we demonstrated how to select fewer, but suitable mutants by following mutation testing principles.

The second improvement of the approach was the demonstrating its relevance for the testing security of web services concerning multi-user context. As a consequence of intensive user collaborations in such services, user authentication credentials should be protected from malicious parties and adversaries. The dependability of social web services depends on offering privacy and security. We modeled and verified a Blog web service with two users including their privacy.

Lastly, we extended one of the previously defined mutation operators and introduced three new mutation operators to generate additional mutants. The evaluation of the approach showed that the mutation-selection criteria speed up the MBMT approach preserving the same quality of the test.

In future work, we plan to provide further experiments on web services and employ smarter mutations by combining primary mutation operators. Higher order mutations might be more efficient since infection of the first order mutations can be quickly discarded by other guards in the first place; thus they will not be selected for testing. Therefore, creating less restricted models using multiple mutations would help in generating stronger mutants.

Some of the problems that should be addressed in future work are scalability of test models and context-aware test generation. For large-scale web applications, which concurrent operations are prone to vulnerabilities, modeling and testing

the critical parts using the approach would be helpful. For these issues, we can focus on various test modularization principles such as aspects, contracts etc. and validating that part rather than including other parts of the system.

ACKNOWLEDGMENT

This work has been funded by the ECSEL MegaM@Rt^2 project. This project has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 737494. This Joint Undertaking receives support from the European Union's Horizon 2020 research and innovation programme and Sweden, France, Spain, Italy, Finland and the Czech Republic.

We would like to thank Tewodros Deneke and anonymous reviewers for comments that significantly improved the manuscript.

REFERENCES

[1] The Open Web Application Security Project (OWASP). [Online; accessed 11-June-2018].

[2] M. Aboutrab et al. Specification mutation analysis for validating timed testing approaches based on timed automata. In *36th Annual IEEE Computer Software and Applications Conference, COMPSAC 2012, Izmir, Turkey, July 16-20, 2012*, pages 660–669, 2012.

[3] B. Aichering et al. Time for Mutants – Model– Based Mutation Testing with Timed Automata. In *Tests and Proofs*, pages 20–38. Springer, 2013.

[4] B. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick, and S. Tiran. MoMuT::UML model-based mutation testing for UML. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–8, April 2015.

[5] B. K. Aichernig and E. Jöbstl. Towards symbolic model-based mutation testing: Combining reachability and refinement checking. *arXiv preprint arXiv:1202.6123*, 2012.

[6] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[7] P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[8] C. Armstrong. Modeling web services with uml. In *OMG Web Services Workshop*, 2002.

[9] G. Behrmann, A. David, and K. G. Larsen. A tutorial on uppaal. In *Formal methods for the design of real-time systems*, pages 200–236. Springer, 2004.

[10] F. Belli et al. A holistic approach to model–based testing of Web service compositions. *Software: Practice and Experience*, 44(2):201–234, 2014.

[11] F. Belli et al. Model-based mutation testing - approach and case studies. *Sci. Comput. Program.*, 120:25–48, 2016.

[12] M. Bozkurt, M. Harman, and Y. Hassoun. Testing web services: A survey. 2011.

[13] BS 7799-3:2017. Information security management systems – guidelines for information security risk management.

[14] M. E. Cambronero et al. Validation and verification of web services choreographies by using timed automata. *Journal of Logic and Algebraic Programming*, 80(1):25–49, 2011.

[15] X. Devroey, G. Perrouin, M. Papadakis, A. Legay, P.-Y. Schobbens, and P. Heymans. Featured model-based mutation analysis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 655–666. ACM, 2016.

[16] G. Dıaz et al. Model checking techniques applied to the design of web services. *CLEI Electronic Journal*, 10(2), 2007.

[17] N. Dragoni and F. Massacci. Security-by-contract for web services. In *Proceedings of the 2007 ACM Workshop on Secure Web Services*, SWS '07, pages 90–98, New York, NY, USA, 2007. ACM.

[18] L. Frantzen et al. Towards model-based testing of web services. 2006.

[19] M. Grinberg. *Flask Web Development: Developing Web Applications with Python*. " O'Reilly Media, Inc.", 2014.

[20] A. Hessel et al. *Testing Real-Time Systems Using UPPAAL*, pages 77–117. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[21] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, 2011.

[22] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. The theory of timed i/o automata. *Synthesis Lectures on Distributed Computing Theory*, 1(1):1–137, 2010.

[23] K. G. Larsen et al. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct. 1997.

[24] K. G. Larsen, F. Lorber, B. Nielsen, and U. M. Nyman. Mutation-based test-case generation with ecdar. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 319–328, March 2017.

[25] S. Lee et al. Automatic mutation testing and simulation on owl-s specified web services. In *Simulation Symposium, 2008. ANSS 2008. 41st Annual*, pages 149–156. IEEE, 2008.

[26] S. C. Lee and J. Offutt. Generating test cases for XML-based Web component interactions using mutation analysis. In *Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on*, pages 200–209, Nov 2001.

[27] J.-h. Li et al. Mutation analysis for testing finite state machines. In *Electronic Commerce and Security, 2009. ISECS'09. Second International Symposium on*, volume 1, pages 620–624. IEEE, 2009.

[28] E. Marcos et al. Representing web services with uml: A case study. In *International Conference on Service-Oriented Computing*, pages 17–27. Springer, 2003.

[29] R. Nilsson, J. Offutt, and J. Mellin. Test case generation for mutation-based testing of timeliness. *Electronic Notes in Theoretical Computer Science*, 164(4):97 – 114, 2006. Proceedings of the Second Workshop on Model Based Testing (MBT 2006).

[30] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman. Mutation testing advances: an analysis and survey. *Advances in Computers*, 2017.

[31] P. Pettersson. *Modelling and verification of real-time systems using timed automata: theory and practice*. 1999.

[32] I. Pill, I. Rubil, F. Wotawa, and M. Nica. Simultate: A toolset for fault injection and mutation testing of simulink models. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 168–173, April 2016.

[33] I. Rauf et al. An integrated approach for designing and validating rest web service compositions. In *10th International Conference on Web Information Systems and Technologies*, volume 1, pages 104–115. SCITEPRESS Digital Library, 2014.

[34] L. Richardson and S. Ruby. *RESTful web services*. O'Reilly, 2008.

[35] M. Salas and E. Martins. Security testing methodology for vulnerabilities detection of xss in web services and ws-security. *Electronic Notes in Theoretical Computer Science*, 302(Supplement C):133 – 154, 2014. Proceedings of the XXXIX Latin American Computing Conference (CLEI 2013).

[36] F. Siavashi et al. On mutating uppaal timed automata to assess robustness of web services. In *Proceedings of the 11th International Joint Conference on Software Technologies*, volume 1, pages 15–26. SCITEPRESS-Science and Technology Publications, 2016.

[37] F. Siavashi et al. *Testing Web Services with Model-Based Mutation*, volume 743, page 4567. Springer, 2017.

Faezeh Siavashi

Model-based Verification and Testing of Web services

Functionality, Robustness and Vulnerability Analysis

In this thesis, we define a model-based testing approach to evaluate the behavior of web services and their compositions. The goal of using models is to introduce a verifiable specification of the web service that is used later on to generate tests that are executed against the implementation of the web service. Moreover, we extend our Model-Based Testing approach in the context of mutation testing to assess the robustness and security vulnerabilities of web services in the presence of unexpected or invalid conditions and inputs. The approaches defined in this thesis have been applied in two case studies and the results show that our testing methodology can create test cases that explore the behavior of the systems extensively and reveal new faults that remain undetected by traditional model-based testing methods.

ISBN 978-952-12-3936-6