Dissertation

It’s about Time!

Model-Based Mutation Testing for Synchronous and Asynchronous Timed Systems

Florian Lorber[1]

Institute of Software Technology (IST)
Graz University of Technology
Austria

Supervisor/First reviewer:	A.o. Univ.-Prof. DI Dr. Bernhard K. Aichernig
Second reviewer:	Assoc. Prof. Cristina Seceleanu, Docent, Ph.D.

Graz, 10 June 2016

[1] E-mail: florber@ist.tugraz.at

Dissertation

Modellbasiertes Mutationstesten von synchronen und asynchronen zeitkritischen Systemen

Florian Lorber[1]

Institut für Softwaretechnologie (IST)
Technische Universität Graz
Österreich

Betreuer/1. Gutachter:	A.o. Univ.-Prof. DI Dr. Bernhard K. Aichernig
2. Gutachter:	Assoc. Prof. Cristina Seceleanu, Docent, Ph.D.

Graz, 10. Juni 2016

Diese Arbeit ist in englischer Sprache verfasst.

[1] E-Mail: florber@ist.tugraz.at

Abstract

The amount of software in nowadays life is increasing rapidly. This affects various different areas, including the automotive industry. In 2009, the amount of code in a typical premium-class automobile was estimated to be hundred million lines of code. This code regulates everything from basic functionality to the most safety-critical parts. Many of these functionalities must comply to a strict real-time behavior. Such systems, where the timing behavior is as important as correct functional behavior, are called real-time systems. Some examples from the automotive domain are the brakes and the airbag. A delayed reaction of either might end fatally, and must be prevented at any cost.

Testing has proven to be an effective method for detecting bugs and gaining confidence in a system. However, the manual creation of high quality test cases is a tedious and error prone task. Consequently, automated test-case generation is an active and important research area. One of its main fields is model-based testing, where test cases are derived from a formal specification of the system. These tests are usually generated according to specified coverage criteria, as for instance transition coverage or state coverage, in case of graphical models. One special instance of model-based testing is model-based mutation testing: it is a fault-based approach that alters the correct specification according to predefined fault models and generates test cases that successfully detect these alterations.

The main goal of this thesis was to extend this model-based mutation testing technique to real-time systems. This proved to be a rather imprecise goal, given the vast amount of different types of real-time systems. Thus, the goal was refined to applying the technique to two instances of oppositional types of models, asynchronous and synchronous models.

As an instance of the first type, timed automata were chosen. They are among the most established and well-known models for timed systems, and attracted a high volume of research activities in the last decades. We developed a bounded model-checking algorithm for model-based mutation testing of timed automata that was implemented via SMT-solving. The approach is restricted to deterministic models, which is a rather limiting restriction in practice. Thus we also developed a bounded determinization approach for timed automata. While it is well known that timed automata can not be determinized in general, this approach still can be applied to non-determinizable timed automata, due to the bounded setting. Furthermore, we investigated how the mutation-based approach can be used for the localization and repair of bugs in faulty systems-under-test.

We developed requirement interfaces as an instance of synchronous models. They are a contract based formalism we propose as a means for easily building traceability between natural language requirements and corresponding parts of the specification. The contracts are composed via conjunction. In a first step, we developed a test purpose driven test-case generation for untimed requirement interfaces. Then we integrated model-based mutation testing by generating test purposes leading to the introduced faults. Finally, we investigated how to model and test real-time constraints with requirement interfaces.

We implemented the techniques both for the asynchronous and the synchronous systems. The two implementations are called MoMuT::TA and MoMuT::Reqs. We evaluated these tools in several case studies. The most relevant ones were an airbag chip of Infineon and an adjustable speed limiter of Volvo.

Keywords: Test-Case Generation, Model-Based Testing, Mutation Testing, Timed Automata, Real-Time Systems, Determinization, Conformance, Timed Input-Output Conformance (tioco), SMT Solving.

Kurzfassung

Die Menge an Software in modernen Geräten steigt laufend an. Dies zeigt sich unter anderem auch stark in der Automobilindustrie: Schon im Jahr 2009 enthielt ein durchschnittliches Premiumauto bereits hunderttausende Zeilen an Code. Die Verifikation dieses Codes ist ein anhaltendes Forschungsthema, da schon kleine Abweichungen vom geplanten Verhalten zu großen Problemen führen und Menschenleben gefährden können. Viele dieser Teile sind zusätzlich einem sehr strikt definierten Zeitverhalten unterworfen. Eine Verletzung dieser Zeitbedingungen, wie verzögertes Bremsverhalten oder verspätetes Auslösen eines Airbags, kann ebenso zu Katastrophen führen wie generelle Fehlfunktionen.

Testen hat sich als eine der wichtigsten Methoden zur Verifikation der korrekten Funktionsweise solcher Bauteile erwiesen und dient sowohl dem Auffinden von Fehlfunktionen als auch dazu höheres Vertrauen in die Systeme zu gewinnen. Viele Firmen verwenden jedoch noch manuell erstellte Testfälle, welche sowohl einen hohen Aufwand fordern als auch sehr fehleranfällig sind. Daher ist die Entwicklung automatisierter Testfallgenerierungsmethoden ein permanentes und wichtiges Forschungsthema. Modellbasierte Testfallgenerierung ist eine der prominentesten Methoden dafür. Ausgehend von einem formalen Modell, welches das Verhalten des Systems darstellt, werden Testfälle erstellt, bis ein vorbestimmtes Ziel, wie die Abdeckung aller Zustände im Modell, erreicht ist. In dieser Dissertation wird eine fehlerbasierte Methode namens modellbasiertes Mutationstesten zur Testfallgenerierung verwendet. Zuerst werden automatisiert Fehler in das formale Modell eingebaut, welche typische Programmierfehler modellieren. Diese fehlerhaften Modelle werden Mutanten genannt. Danach werden Testfälle erzeugt die gezielt zu den eingebauten Fehlern führen und in der Lage sind äquivalente Fehler in echten Implementierungen zu erkennen. Das Hauptziel dieser Arbeit war die Kombination von modellbasiertem Mutationstesten und zeitkritischen Systemen. Dabei wurden asynchrone und synchrone Systeme behandelt.

Als Formalismus um asynchrone Systeme zu modellieren wurden Timed Automata gewählt. Für diese weit verbreitete Modellierungssprache existiert bereits eine Vielzahl an theoretischen Ergebnissen und praktischen Tools. Im Rahmen dieser Arbeit wurde eine Methode für Mutations-basierte Testfallgenerierung entwickelt, die auf bounded model-checking und SMT-solving beruht. Diese Methode war vorerst auf deterministische Systeme eingeschränkt, was für viele praktische Anwendungen ein Hindernis darstellt. Daher wurde auch eine Methode entwickelt, um Timed Automata zu determinisieren, was durch eine Einschränkung der Länge der Pfade im Automaten möglich wurde. Zusätzlich wurde untersucht, wie sich Mutanten von Timed Automata dazu einsetzen lassen, Fehler in Programmen zu lokalisieren.

Um synchrone Systeme zu testen, wurde der Formalismus Requirement Interfaces entwickelt. Diese textuelle Modellierungssprache erlaubt es, einzelne Bereiche des Systems getrennt zu modellieren und diese Teilmodelle zu einem gesamten Modell zu kombinieren. Außerdem zeichnen sich Requirement Interfaces dadurch aus, dass sie eine starke Verbindung zu den textuellen Anforderungen an das System aufbauen. Requirement Interfaces basieren auf Kontrakten, welche für gewisse Zustände das Systems das Folgeverhalten des Systems garantieren. In einem ersten Schritt wurde eine Testfallgenerierung entwickelt die Testfälle für manuell definierte Testziele erstellt. Danach wurde diese Technik erweitert, um für Mutanten eines Kontrakts automatisiert Testziele zu entwerfen, mit dem Ziel eingebauten Fehler zu erkennen. Abschließend wurde untersucht, wie Requirement Interfaces dazu verwendet werden können, Zeitanforderungen an das System zu modellieren.

Zu beiden Ansätzen wurden Tools entwickelt, die unter den Namen MoMuT::TA und MoMuT::Reqs in die MoMuT Tool-Familie aufgenommen wurden. Die Evaluierung der Tools wurde anhand mehrerer industrieller Fallbeispiele durchgeführt, darunter ein Airbag Chip von Infineon und ein Geschwindigkeitsbegrenzer von Volvo.

Schlagworte: Testfallgenerierung, Modellbasiertes Testen, Mutationstesten, Timed Automata, Echtzeitverhalten, Determinisierung, Konformanz, Timed Input-Output Conformance (tioco), SMT-Solving.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

. . . .
place, date

. . . .
(signature)

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

. . . .
Ort, Datum

. . . .
(Unterschrift)

Acknowledgements

There are many people I want to thank for their support during the years of my doctoral studies, and it is hard to put them in order.

But clearly, first off, I want to thank my supervisor, Bernhard Aichernig and I want to thank him for several reasons: for introducing me to the topics of testing and formal methods, for keeping up my motivation during several years, for weekly meetings to keep me on track, for help and discussions on scientific and non-academic topics, for the fast and helpful reviewing of this thesis and for bearing with me, until it *was about time* to finish my thesis.

Next, as any good son should (and wants to) do in a thesis, I want to thank my parents. For enabling my education up to this point, for having an open ear after paper rejections, for celebrating with me after acceptions and for always providing me a home to visit.

I especially want to thank Dejan Ničković for helping me set a foot in the world of timed automata and his ongoing support through the years since. I would like to thank Elisabeth Jöbstl for being a role model and showing me how to be a PhD student and how to finish a thesis, and for being a good friend and colleague. And I want to thank all of my co-authors, especially Amnon Rosenmann, Martin Tappler and Stefan Tiran.

I want to thank all my colleagues at work for the great time we had together, making my time at the university very rewarding, not only from the scientific point of view.

Last but not least, I want to thank Cristina Seceleanu for reviewing my thesis and attending my Rigorosum as external examiner.

This work has been financially supported by the ARTEMIS Joint Undertaking under grant agreements N° 269335 and N° 332830 and from the Austrian Research Promotion Agency (FFG) under grant agreements N° 829817 and N° 838498 for the implementation of the projects MBAT, Combined Model-based Analysis and Testing of Embedded Systems and CRYSTAL, Critical System Engineering Acceleration.

Florian Lorber
Graz, Austria, June 2016

Danksagung

Ich möchte mich hiermit bei allen bedanken, die mich während meines Doktoratstudiums unterstützt haben.

Zuallererst möchte ich meinem Betreuer Bernhard Aichernig für alles danken, das er für mich getan hat! Er hat nicht nur mein Interesse am Thema der Arbeit geweckt, sondern mich auch in den Jahren danach beständig unterstützt und meine Motivation und Freude an der Arbeit über die gesamte Zeit aufrecht erhalten.

Nun möchte ich meinen Eltern danken, die mir nicht nur meine Ausbildung ermöglicht haben, sondern auch während den letzten Jahren immer ein offenes Ohr geboten haben, mich nach Misserfolgen aufgebaut haben, Erfolge mit mir gefeiert haben und mir immer ein offenes Zuhause bieten.

Außerdem möchte ich bei Dejan Ničković sowohl dafür bedanken, dass er mich in das Thema Timed Automata eingeführt hat, als auch für die gute Zusammenarbeit in den Jahren danach. Bei Elisabeth Jöbstl möchte ich mich dafür bedanken, dass sie mir neben einer Freundin auch ein gutes Vorbild war und mir gezeigt hat, wie man ein Doktorat durchlebt und zu einem Abschluss bringt.

Natürlich möchte ich mich auch bei all meinen Koautoren für die gute Zusammenarbeit bedanken, insbesondere bei Amnon Rosenmann, Martin Tappler und Stefan Tiran.

Ebenso möchte ich mich bei all meinen Kollegen am Institut für die großartige gemeinsame Zeit bedanken, die wir gemeinsam hatten.

Und abschließend möchte ich Cristina Seceleanu für das Begutachten meiner Arbeit und die Teilname als Zweitprüferin bei meinem Rigorosum danken.

Diese Arbeit wurde finanziell unterstützt durch das ARTEMIS Joint Undertaking unter den Projektnummern N° 269335 und N° 332830 und durch die österreichische Forschungsförderungsgesellschaft (FFG) unter den Projektnummern N° 829817 und N° 838498, im Rahmen der Projekte MBAT, Combined Model-based Analysis and Testing of Embedded Systems, und CRYSTAL, Critical System Engineering Acceleration.

Florian Lorber
Graz, Österreich, Juni 2016

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

1 Introduction

1.1 Motivation

Almost all industrial applications incorporate more and more software into their products. Especially in the context of the automotive industry, this is an ever-growing trend. In 2015, Ford announced that the new Ford GT contains 10 millions lines of ”mission critical” code. Quality assurance of this code must be among the highest-prioritized tasks during development, but at the same time achieving high quality for this embedded code is very hard. In cases where the quality assurance fails, this may on one hand lead to hazardous accidents, and on the other hand, to horrendous sums needed to be spent by the companies. In 2014, around 700000 cars had to be recalled due to bugs in embedded software.

Testing has proven to be an effective way of assuring the quality of a system. However, traditionally, all tests were designed manually, which is a labor-intensive, tiresome and error-prone task. Up to now, many companies still mostly rely on manually created test cases.

Model-based testing, which is a well known testing technique, provides an alternative to automate that workflow: it enables modeling the system in one compact formal model, which is then used to automatically generate test suites, according to some coverage criteria on the test model. Model-based mutation testing, which is the technique applied in this thesis, uses fault models as a coverage criterion, and aims at producing test suites able to cover a defined set of possible faults in a system. While testing can never guarantee the total absence of bugs [76], this technique can at least guarantee the absence of certain bugs.

Real-time systems deployed in cars need special attention. Besides the general challenges during quality assurance, real-time systems require their timing constraints to be analyzed especially. Even if such a system works correctly with regards to their functional specification, a slightly delayed reaction of a brake or an airbag can lead to hazardous events. In the context of real-time systems, there exist several formalisms that take special account of timing properties, facilitating to model them both clearly and precisely. In this thesis we will investigate two of these formalisms, to prove their value in the context of model-based mutation testing and to show their capability for detecting timing-related faults.

1.2 Testing

Testing is one of the main approaches for software *verification*. Verification tries to verify that a system works according to the specified requirements. Verification is often compared to *validation*, where we try to determine, whether the system satisfies the users needs and actually can perform the tasks it was designed for. In this thesis, we will investigate verification, focusing on model-based testing.

The goal of testing is to provide a verdict for a *system under test*, stating whether it behaves correct or incorrect. The system under test may either be an implementation, a simulation or a physical system. A system is defined to be *deterministic*, if it always shows the same behavior for the same input, and *non-deterministic* if the same inputs may lead to different outputs.

During the testing process, we trigger the system under test with different sequences of inputs, and observe the outputs it produces. These outputs are then validated via a *test oracle*, which provides the outputs we expect from the system, and is assumed to be correct.

Test cases are sequences of inputs and outputs that contain both the inputs that are applied on the system under test, and the outputs expected by the test oracle. For deterministic systems, the test cases are linear sequences, that cover one trace through the system. For non-deterministic systems, test cases may branch, as the same inputs might lead to different internal states and different outputs.

During *test-case execution*, we apply the inputs, observe the outputs, and finally issue a *verdict*. If all outputs along the test case conformed to the outputs of the test oracle, we return the verdict *pass*. If an output of the system deviated from the path we wanted to test, but was still a correct output, we assign the verdict *inconclusive*. If, however, one of the outputs did not conform to the test oracle, we assign the verdict *fail*, indicating that there was a bug in the system.

We distinguish three terms for bugs [35]: A *fault* is a static defect in the software. If the fault causes the system to reach a wrong internal state, it is called an *error*. If that error propagates to a wrong output, it is considered a *failure*. Testing aims at detecting failures in the system.

In model-based testing approaches, a formal model of the system is used as the test oracle. The system under test is generally considered a *black-box*. Thus, we do not expect to have access to the source code and do not know anything of the state and internal transitions of the system. We thus can only determine its behavior through inputs and outputs. The outputs of the system are used to determine whether it conforms to the formal model. More details on model-based testing will be given in the following section.

1.3 Model-Based Testing

Model-based testing [165, 164] is a well-established technique for automated test-case generation. Instead of manually producing a test suite, in model-based testing we build a formal model and generate a test suite, according to that model. Besides its advantages with regards to reduced effort and higher quality, the main advantage of model-based testing is adaptability. Where previously an update of the specification required to update each of the test cases individually, now it is possible to simply update the model, and rerun the test-case generation.

The test suites are generated in order to satisfy certain coverage criteria. For graphical models, this may be edge coverage or state coverage, where the test suite needs to reach each edge, resp. state, at least once. More refined coverage criteria, like k-transition coverage where each possible combination of k transitions must be tested, obviously produce better test suites. In the next section, we will discuss a fault-oriented coverage criterion, that will be applied in this thesis.

Figure 1.1 gives an overview of the workflow of model-based testing. The formal model is built independently from the system under test, to ensure that if some requirements were misunderstood during development, those errors are not integrated into the test model as well. Then, the test-case generation is performed, according to the selected coverage criteria. This produces a test suite, which can be executed on the *system under test* (SUT) by the test driver. The test driver usually needs to adapt the level of abstraction, so that the usually more abstract test cases fit to the system under test. If the system conforms to the model, each of the test cases should return the verdict *pass*. If only one of the test cases *fails*, there exists a bug. This bug may be either in the system under test, or in the model.

Model-based testing can be categorized in many ways. One main categorizations we want to point out especially is the separation into *online* and *offline* testing. In online testing approaches, the system under test and the model are executed simultaneously. The model is used to determine the next inputs to pass to the system under test. The outputs produced by the system are then validated using the model. The advantages of this approaches are the high flexibility and the fact that no time and effort needs to be spent on creating test suites beforehand. On the other hand, offline testing has the advantages of reusability and reproducibility. The test suite is generated once, and can be run again whenever the system changes. Additionally, for hard real-time systems online testing might not always be possible, as the simulation of the model in parallel to the execution may not be fast enough to satisfy all deadlines. In this thesis, we focus on offline testing, and most of the contributions of this thesis lie in the context of test-case generation.

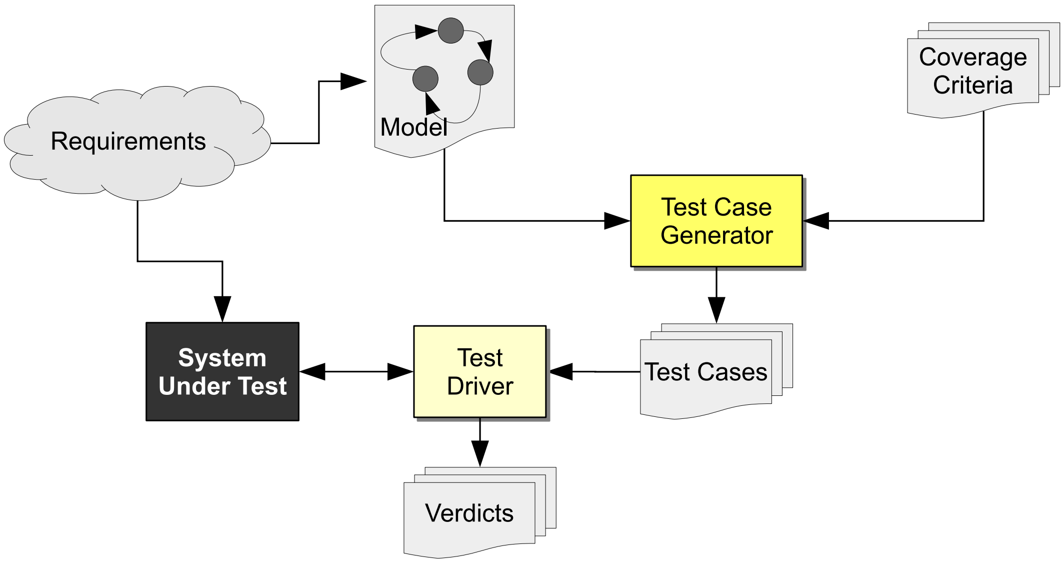


Figure 1.1: Overview of the model-based testing workflow, based on a figure by Elisabeth Jöbstl [105].

Utting et al. [165] provided a taxonomy for model-based testing, defining three major and six minor dimensions. We give an illustration in Figure 1.2, and classify where our approaches belong to, by marking the corresponding classifications in blue.

The dimension labeled by *model specification* is split into three parts, *scope*, *characteristics* and *paradigm*. The *scope* of the specification states whether only the *formal model of the SUT* is given, or whether the *environment* is modeled as well. In our work, we only model the SUT, and do not limit the behavior of the environment. The *characteristics* states whether the system contains timing and non-determinism and whether it is a discrete, a hybrid or a continuous system. As already discussed, the work in this thesis focuses on timed behavior. We will also provide support for non-determinism. Both modeling types we use, timed automata and requirement interfaces, are discrete models. Note that timed automata can be extended to hybrid automata [93]. The *paradigm* defines what kind of notations are used, dividing them into *pre / post conditions*, *transition-based*, *history-based*, *functional*, *operational*, *stochastic* and *data-flow* notations. Our requirement interfaces are data-flow specifications based on pre/ post conditions, and our timed automata are transition-based.

The *test-case generation* dimension is split into *test-selection criteria* and *test-case generation technology*, where *test-selection criteria* are categorized into *structural model-coverage*, *data coverage*, *requirements-based coverage*, *ad-hoc test-case specification*, *random and stochastic* and *fault-based* criteria and the *test-case generation technology* defines the used technology and can be categorized into *manual*, *random*, *graph search*, *model-checking*, *symbolic execution*, *theorem proving* and *constraint solving*. We concentrated on fault-based criteria, but did implement a random testing functionality for timed automata and our requirement interfaces implicitly support ad-hoc test-case specifications via our test purposes and requirements-based coverage due to the tight coupling of each contract in the requirement interfaces to its textual requirements. We use bounded model-checking as our test-case generation technology, and implement it via SMT-solving.

Test-case execution is differentiated into *offline-testing* and *online-testing*, where we focused on offline-testing.

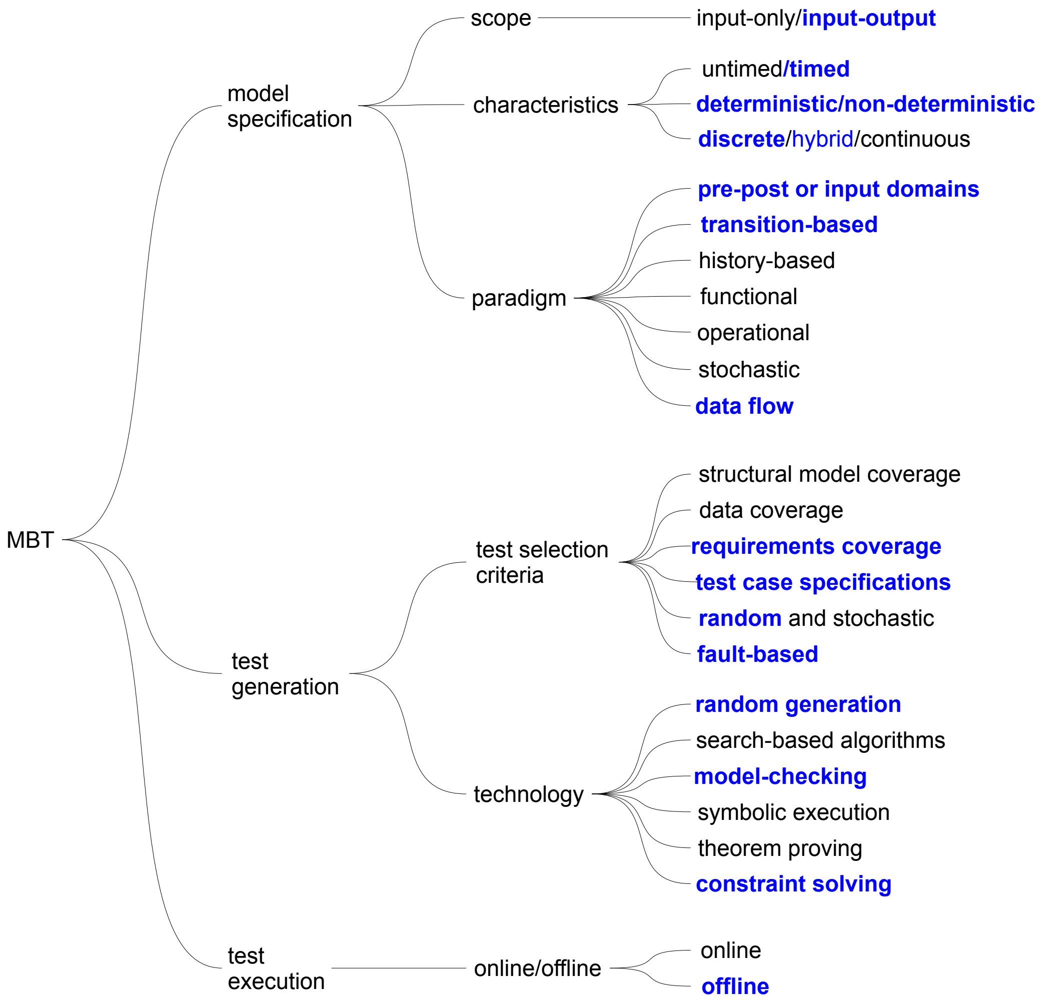


Figure 1.2: The taxonomy of model-based testing, based on a figure from Elisabeth Jöbstl [105], which was based on a figure from Utting et al. [165].

1.4 Model-Based Mutation Testing

Model-based mutation testing (see e.g. the following publications [136, 6, 60]) is a subclass of model-based testing, concentrating on the detection of specified faults and using the detection of these faults as coverage criterion. Starting from a correct specification, we intentionally insert modeling errors according to a set of fault models in the form of *mutation operators*. The mutation operators syntactically alter the specification model. This produces a set of faulty models, called *mutants* or *model mutants*, to be distinguished from faulty implementations. In our case, each of the mutants contains one fault at one specific location. They are thus called *first-order* mutants.

Then, we perform a conformance check between the correct specification and each of the model mutants. In case the mutant shows any output behavior that is not allowed by the specification, we consider the mutant *killed* and generate a test case leading directly to that fault. In the other case we consider the mutant an *equivalent mutant* and discard it.

The test suite that is generated by this procedure guarantees to catch all bugs that correspond to any of the specified mutation operators, as long as the corresponding mutant was not equivalent and the system under test is deterministic.

Model-based mutation testing is closely related to general *mutation testing* [103]. However, in classical mutation testing approaches the source code is mutated instead of the model. These techniques are usually not used to generate new test suites, but to asses the quality of existing test suites. The test suites are executed on the implementation mutants, to check how many of them are killed. The results can then be expressed via the *mutation score*, which is defined by the number of killed mutants divided by the number of total non-equivalent mutants.

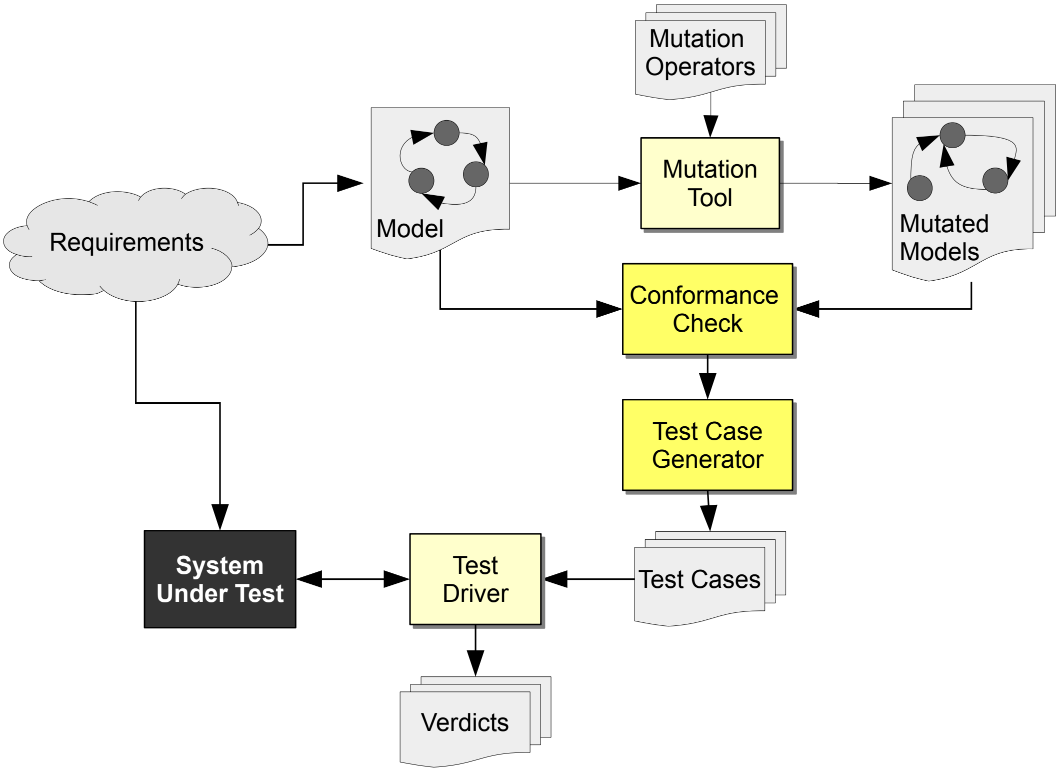


Figure 1.3: Overview of the model-based mutation testing workflow, based on a figure by Elisabeth Jöbstl [105].

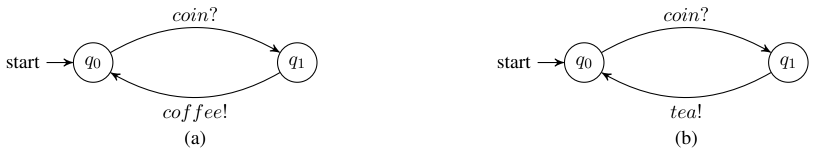


Figure 1.4: A model specification of a coffee machine and a possible mutant.

We illustrate the typical workflow of model-based mutation testing in Figure 1.3. Like in classical model-based testing, the formal model and the system under test are developed independently. The model is then mutated according to the selected mutation operators, producing a set of mutated models. Then, we perform the conformance check between the original model and all of the mutated ones. For each mutated model that does not conform to the original, we generate a test case intended to reveal this non-conformance. The test cases are executed in the same way as in the normal model-based testing workflow.

Example 1.1. Consider the formal model of a simple coffee machine presented in Figure 1.4(a) as a labeled transition system with the input transition *coin?* and the output transition *coffee!*. After inserting a coin, it produces a coffee. A typical mutation operator might for instance change the labels of transitions. Figure 1.4(b) shows such a mutation. In this case, the coffee machine produces tea instead of coffee. To reveal the mutation, one would simply insert a coin and then observe the wrong behavior. A test case corresponding to that trace would contain the input *coin?*, followed by the expected output *coffee!*. When executing the test case on a faulty implementation corresponding to the mutant, one would apply the *coin?* input, observe the output *tea!* from the system under test, compare it to the expected output, and realize that the test case failed. □

1.5 Real-Time Systems

Real-time systems are systems that operate under strict timing constraints, where any delays may lead to severe consequences. Some examples that will appear within this thesis are for instance an airbag chip and a speed limiter for automobiles. The consequences of a delayed airbag or a car still driving at same pace, even though one already hit the brakes, are obvious.

Real-time systems are divided into two classes. Soft real-time systems, where the missing of a deadline only degrades the value of the produced output, and hard real-time systems, where missing a deadline is considered a system failure. We concentrate on hard real-time systems. For testing such systems, it is not enough to test the functional behavior, but one needs to emphasize testing whether all of the deadlines are met as well.

In this thesis, we will discuss two types of systems: first, we will discuss asynchronous systems, where input and output events may appear in any order as long as all interleavings of events conform to the specification. We will discuss these on the example of timed automata [29], which are finite state machines extended by real variables for measuring time delays, called clocks. Then, we will discuss synchronous systems [44]. Synchronous systems are usually reactive systems, which interact with the environment permanently, at fixed rates that are driven by the environment. At each tick of the external clock, all available input signals are processed, and all output signals are assumed to be produced instantaneously. We will define *requirement interfaces* for modeling synchronous systems.

1.6 Problem Statement

The main problem behind real-time systems is their infinite state-space. Assuming dense time, it is not feasible to perform any non-symbolic operations on these models, and even when assuming discrete time, the timing adds to the complexity of most approaches. After an initial investigation period of the problem and the related work, we found several challenges.

The main question intriguing us was whether it is possible to perform model-based mutation testing on real-time systems. We found several topics that are challenges for both, mutation-based testing and real-time systems, e.g. silent transitions and non-determinism. During test-case generation these model-elements increase the complexity, as one can not be sure about the internal state of the specification after a given trace, and the related work on timed automata showed that they also pose a problem from the theoretical aspect. Another challenge for mutation-based testing of real-time systems was the design of mutation operators that reflect timing faults. These challenges led to the definition of our first research question, **Q1**.

- **Q1: Can real-time systems be tested with model-based mutation testing?**
  - **Q1.1**: Can we support non-determinism for model-based mutation testing?
  - **Q1.2**: Can we support internal transitions?
  - **Q1.3**: What kind of mutation operators reflect violated timing properties?

Next, we discussed how to implement the approach. We had several options available, as e.g. symbolic execution, but finally decided to investigate bounded model-checking [55]. Bounded model-checking aims at finding counter-examples to given properties, but only inspect the state space up to a given bound k. This correlates well with test-case generation, as our test cases are bounded anyway. The technique can be implemented using SAT or SMT solvers. Our decision to use bounded model-checking led to our second research question, **Q2**.

- **Q2: Can bounded-model checking be applied to test real-time systems?**
  - **Q2.1**: How can we encode the conformance check via bounded-model checking?
  - **Q2.2**: How should we implement the bounded model-checking?
  - **Q2.3**: Is bounded model-checking an efficient approach?

As already mentioned, we planned to test both synchronous and asynchronous systems. To achieve this, we needed to decide on exemplary formalisms for both types. Once the decision was made, we wanted to analyze their usefulness in the context of real-time systems and find out whether their combination might provide synergies. These goals are formalized in our third research question, **Q3**.

- **Q3: Can the approach be applied to both synchronous and asynchronous systems?**
  - **Q3.1**: What modeling languages should we use as representations for the individual modeling styles?
  - **Q3.2**: How well do timing properties fit in the individual styles?
  - **Q3.3**: How can the two approaches be combined?

Once all other goals were achieved, we needed to evaluate our approaches. Given the challenges we already encountered, especially the expected state-space explosion during the conformance checks due to analyzing all bounded traces of the specifications and mutants, we decided to evaluate the approaches based on their runtimes and achievable search depths. To optimize the results, we also planned to investigated how to reduce the state-space before or during our conformance checks. The concreate goals are summarized in our last research question, **Q4**.

- **Q4: Can the approach be effectively performed on industrial use cases?**
  - **Q4.1**: How big does the state-space of the models become?
  - **Q4.2**: How can we reduce the state-space explosion?
  - **Q4.3**: Are the reachable search depths of our bounded model-checking approach sufficient for industrial case studies?

1.7 Research Context

The model-based mutation testing approach already is a long-term research topic in our research group. The extension to timed systems was partly chosen due to personal interest, and partly due to industrial needs within the ongoing research projects. In the next two subsections we will first discuss the research projects that funded the presented research and then present the MoMuT tool family, to which the prototypes developed in this thesis belong.

1.7.1 Research Projects

For the first two years, the research done within this thesis was solely driven by the European Artemis Project MBAT, which stands for model-based analysis and testing. Within this project, we were strongly collaborating with the Austrian Institute of Technology (AIT), AVL List GmbH and Infineon Technologies, Austria AG. As already indicated by the name of the project, its main focus was to build a stronger connection between the often separated tasks of analyzing a system and testing it. While we obviously focused more on the testing aspect, we also put considerable effort in complying to the standards for data exchange between the tools, to facilitate the analyzing of our input models and test cases by other tools.

Halfway during the runtime of MBAT, a second research project, the European Artemis Project CRYSTAL, Critical System Engineering Acceleration, started. The goal of CRYSTAL was to extend the interoperability between tools even further, and again couple the tools from different verification and validation tasks even closer. In CRYSTAL, our main cooperation partners were AIT, AVL, Chalmers University of Technology and Volvo .

Currently, we are also involved in the local research project TRUCONF, where the insights and expertises gained during this thesis are applied for the testing the real-time aspects of measurement devices for automobiles.

1.7.2 The MoMuT Tool Family

The MoMuT tool family, which focuses on model-based mutation testing, was created during the completed local research project MOGENTES, in a collaboration of our research group and AIT. There, and in the following local research project TRUFAL, they developed the frontend for UML and two backends for action systems, one explicit and one symbolic. This was the first tool of the family, called MoMuT::UML. The tool suite is presented at `www.momut.org`, including documentation and related papers.

The two prototype tools developed by the author of this thesis and discussed in the following chapters, were incorporated into the MoMuT tool family, and named MoMuT::TA and MoMuT::Reqs. MoMuT::TA stands for ”MoMuT for Timed Automata” and MoMuT::Reqs for ”MoMuT for Requirement

Interfaces". MoMuT::TA is available at the homepage for 64bit Windows. The release of MoMuT::Reqs is planned for the near future.

1.8 Use Cases

In this section we will shortly illustrate the three main industrial use cases that were tested in this thesis. More details can be found in the results chapters of Part I (Chapter 5) and Part II (Chapter 11).

1.8.1 Car Alarm System

The car alarm system is an industrial use case that was provided by Ford in the previous research project MOGENTES. While the project is already finished, the case study persisted as a valuable benchmark example, that was already used in several publications of our research group, e.g. the publications by Aichernig et al. [5, 13], the publication by Krenn et al. [114] and the master thesis and PhD thesis by Elisabeth Jöbstl [104, 105].

The car alarm system is responsible for arming the system twenty seconds after all doors are closed and locked and activating the alarm if the doors are opened afterwards without being unlocked first. The sound of the alarm is then activated for 30 seconds, and the flash for 300 seconds. Although violations of any timing constraints are not leading to hazardous events, the timing constraints did pose a problem when the system was previously verified using action systems [5], and it made sense to analyze them more thoroughly once we had the means to do so.

We analyzed the car alarm system in several variants, including both asynchronous and synchronous specifications. The individual results are presented in Chapter 5 and Chapter 11.

1.8.2 Airbag Chip

The airbag chip was a use case in the project MBAT, that was provided by Infineon Austria. The tested component was, however, not the main airbag chip itself, but a part called the safing engine: if the airbag chip decides to fire, the safing engine is responsible to evaluate the data from the CPU once more, and check whether the data really indicated a crash. Only then, the airbag is deployed. This is a very typical example for a safety-critical real-time system, where both functional faults and timing faults can cost lives. We modelled the safing engine via requirement interfaces, and present our test-case generation results in Chapter 11.

1.8.3 Adjustable Speed Limiter

The adjustable speed limiter is a use case in the ongoing project CRYSTAL, provided by Volvo. A speed limiter is the device for automobiles that allows to set a speed limit and then automatically holds that speed until it is turned off. The current speed limiter can be manually increased or decreased, and momentarily overwritten by a kickdown of the gaspedal. Once more, delays in the reaction of the system, which might be driving at high speed, may lead to catastrophic events. We had several different models of the speed limiter, including both synchronous and asynchronous models. The individual results are presented in Chapter 5 and Chapter 11.

1.9 Contributions and Publications

1.9.1 Contributions

The main contributions achieved within this thesis are summarized below:

- We developed a model-based mutation testing approach for deterministic timed automata. We developed the theory behind it, and implemented it in the tool MoMuT::TA.
- We introduced and implemented a method for bounded determinization and silent transition removal of timed automata, to enable the processing of non-deterministic systems with our test-case generation approach.
- To improve the efficiency of the determinization, we developed an on-the-fly algorithm, which supports networks of timed automata and produces a deterministic bounded unfolding of the product of all automata in the network, while only going through the state-space once.
- We showed how the model-mutants we create for model-based mutation testing can also be used for debugging, where we select a subset of mutants that correspond to a faulty system under test.
- We introduced requirement interfaces, which are a synchronous contract-based specification language for synchronous data-flow systems and showed how to compose them via disjunction, how to check their consistency and how to generate tests from them.
- We introduced the consistency checking and test-case generation approach in the tool called MoMuT::Reqs.
- We extended the test-case generation for requirement interfaces, to support model-based mutation testing, by automatically generating test purposes for mutants.
- We analyzed how time is treated in synchronous languages and how discrete delays can be integrated into requirement interfaces.
- We evaluated both the synchronous and the asynchronous test-case generation on industrial use cases and reported the empirical results.

1.9.2 List of Publications

The work presented in this thesis was for the most parts already published in various international journals, conference proceedings and workshops proceedings, that will be listed below. All of these publications were peer reviewed by at least three anonymous reviewers.

1.9.2.1 Main Publications

- *TAP 2013* [17]. The model-based mutation testing for timed automata was published at the *T*est and Proofs conference in 2013, where I presented it in Budapest, Hungary. The paper was written in strong collaboration with Bernhard Aichernig and Dejan Ničković. I implemented the algorithm, performed all experiments and contributed both to the theory and the writing of the paper.
- *SAFECOMP 2014* [10]. We extended the results from the TAP 2013 paper to the area of model-based debugging. The new content was published at the *C*omputer Safety, Reliability, and Security conference in 2014, where I presented it in Florence, Italy. I wrote most of the paper myself, under supervision of Bernhard Aichernig who wrote some parts, and helped me in polishing the rest. I performed all experiments.

- *QSIC 2014* [12]. The first paper published on requirement interfaces illustrated the airbag chip case study, focusing on the interoperability of all tools in our use case. It was published at the *I*nternational Conference on Quality Software in 2014, where I presented it in Dallas, USA. The paper was written in collaboration with Bernhard Aichernig, Klaus Hörmaier, Dejan Ničković, Rupert Schlick, Didier Simoneau, and Stefan Tiran. I implemented the main parts of the tool and OSLC interface, performed the experiments involving MoMuT::Reqs, organized the writing and wrote most parts that involved MoMuT::Reqs.

- *FORMATS 2015* [123]. Our paper on the bounded determinization and silent transition removal of timed automata was published at the *F*ormal Modeling and Analysis of Timed Systems in 2015 and I presented it in Madrid, Spain. The paper was written in strong collaboration with Amnon Rosenmann, who came up with the initial algorithms and the proofs, Dejan Ničković and Bernhard Aichernig. I implemented the tool, performed all experiments, contributed an algorithm for determinization that decreases the state space by using diagonal and conjunctive constraints and collaborated on the silent transition removal. I also contributed strongly to the writing of the paper.

- *A-MOST 2015* [15]. To extend the applicability of our determinization approach for test-case generation, we then wrote a paper on how to produce partial models conforming to the original specification during the determinization. It was published at the *W*orkshop on Advances in Model Based Testing in 2015, where I presented it in Graz, Austria. The paper was written mostly by myself, with Bernhard Aichernig supervising the work and helping with proof-reading and polishing. I came up with most of the presented ideas and performed the experiments.

- *FMICS 2015* [11]. Our paper covering the theory on requirement interfaces and the first formal experiments for MoMuT::Reqs was published at the *F*ormal Methods for Industrial Critical Systems Workshop in 2015, where I presented it in Oslo, Norway. The paper was written in strong collaboration with Bernhard Aichernig, Klaus Hörmaier, Dejan Ničković and Stefan Tiran. I contributed to the theory, did most of the implementation, performed most of the experiments and wrote several sections of the paper.

- *Festschrift FdB 2016* [18]. We submitted a paper comparing symbolic execution and bounded model-checking on timed action systems and timed automata to a Festschrift in honor of the 60th birthday of Frank de Bour, *T*heory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60-th Birthday. Bernhard Aichernig presented it in 2016 in Eindhoven, Netherlands. The paper was written in strong collaboration with Bernhard Aichernig and Martin Tappler. Martin Tappler developed the symbolic execution, and performed all corresponding experiments, while I performed the bounded model-checking experiments with MoMuT::TA. Together we developed the translation from timed automata to timed action systems. The paper was written in joint work.

- *TASE 2016* [16]. Our newest paper focuses on networks of timed automata, and the on-the-fly determinization algorithm. It was accepted at the *Theoretical Aspects of Software Engineering* Symposium, which will take place in July 2016, in Shanghai, China. I wrote this paper mostly by myself, supervised by Bernhard Aichernig, who proofread and polished the paper with me. I developed the theory and implementation, performed the experiments and wrote the paper.

- *FMICS and FORMATS Journals*. For our two papers at FMICS and FORMATS we were invited to submit journal versions. Both of them were already submitted, and will hopefully be published by the end of this year. The FMICS journal version is extended by adding support for model-based mutation testing and an empirical evaluation of this functionality. The FORMATS journal version is extended by proofs and by an enhanced empirical study, that adds test-case generation to the determinization process.

1.9.2.2 Other Related Publications

In the early stages of this thesis, we wrote two more papers on test-case generation. As they do not cover real-time systems or model-based mutation testing, they were not included in this thesis:

- *TASE 2012* [19]. In the first paper, we showed how to combine model-based testing and analysis, by performing analysis on the generated test cases. This enables us to determine, whether the test cases cover all aspects that we demand during analysis. The paper was accepted at the *T*heoretical Aspects of Software Engineering Symposium in 2012, where Bernhard Aichernig presented it in Beijing, China. Bernhard Aichernig wrote most of the paper himself. Stefan Tiran and I mostly focused on the experiments and the results section.
- *MODELSWARD 2014* [20]. The second paper focused on formal test-driven development, where we once more performed analysis on the generated test cases, before we used them for the development. The paper was accepted at the *I*nternational Conference on Model-Driven Engineering and Software Development in 2014, where Stefan Tiran presented it in Lisbon, Portugal. Bernhard Aichernig wrote most of the paper, while Stefan Tiran and I did the experiments.

Additionally, we submitted a paper illustrating the speed limiter use case of the CRYSTAL project to MODELS 2016, which is currently being reviewed. The paper was driven by Grischa Liebel from Chalmers University, and written in cooperation with Grischa Liebel, Anthony Anjorin, Eric Knauss and Matthias Tichy. My contribution to this paper was mainly the execution of the experiments involving the MoMuT tools.

Finally, I got two abstracts to PhD Symposia accepted, at the PhD Symposia of *ICST 2015* [121] and *FM 2015* [122]. I presented them in Graz and Oslo in 2015, receiving a Best Presentation Award for the second one.

1.10 Structure of this Thesis

The remainder of this thesis is structured as follows: there will be three parts, where Part I will cover asynchronous systems, Part II will cover synchronous systems and Part III will discuss the conclusions of the preceding parts.

In Part I we will first give some preliminaries and then discuss our model-based mutation testing approach for deterministic systems. This will include the mutation operators, the encoding of the tioco-conformance check as language inclusion and solving the language inclusion check via bounded model-checking.

Then we will present our approach for bounded determinization and silent transition removal. There we will first discuss the preprocessing needed, afterwards we will focus on the silent transition removal and the determinization. Then we will present our on-the-fly algorithm, and how it works for networks of timed automata, which includes building the product directly into the unfolding.

Then we will discuss the case studies and the empirical results of the asynchronous approach.

Finally, we will discuss some minor contributions, including debugging with timed automata mutations and the pruning of the determinized trees and the translation into timed action systems, to compare symbolic execution on timed action systems with bounded model-checking on timed automata.

In Part II we will first introduce requirement interfaces, focusing on their syntax and semantics, but also defining general properties like consistency, refinement and conjunction.

Then we will present our purpose-driven test-case generation for requirement interfaces and the bounded consistency check. These two operations are the core functionalities of MoMuT::Reqs. We will also show how traceability can be provided during test-case generation and present details on the implementation. Then we present how model-based mutation testing can be integrated into the process, facilitating the automated generation of test purposes.

We will then demonstrate a typical development workflow using MoMuT::Reqs both for analysis and the generation of test cases. We present the OSLC integration of MoMuT::Reqs on this use case.

Next, we will discuss the meaning of time in synchronous systems, and how discrete delays may be added to requirement interfaces, by introducing clocks to measure the delays. We will discuss both symbolic and explicit delays, and summarize their individual advantages.

We conclude this part by presenting the case studies used for the symbolic approach und discussing the results we achieved on them.

Part III begins by first comparing the asynchronous and the synchronous approaches and then showing how to combine them, to gain globally asynchronous, locally synchronous systems.

Finally, we conclude the thesis by presenting a summary, the conclusion of the research questions and an overview of possible future work.

Part I

Asynchronous Systems

Overview

Part I of the thesis shows how to apply model-based mutation testing to asynchronous timed systems, demonstrated on the example of timed automata. It is split into four chapters.

First, in Chapter 2 we will present preliminaries for timed automata. Then, in Chapter 3, we will show how model-based mutation testing can be performed on timed automata, by expressing the conformance check between the specification and the mutant as a bounded model-checking problem and solving it with an SMT-solver. As this approach is restricted to fully-observable deterministic timed automata, we will then discuss our bounded determinization approach for timed automata in Chapter 4. Due to the fact that we bound the traces of the automata, this approach can be applied to all timed automata, even though determinization of timed automata is not possible in general. Then we will present our case studies and our experimental results in Chapter 5. Finally, in Chapter 6 we will present some additional work involving timed automata that has been done in the context of this thesis, including a debugging methodology, some ways of pruning the search space for test-case generation and a translation from timed automata to timed action systems, which can be analyzed symbolically.

2 Timed Automata

Timed Automata were first introduced by Alur and Dill [29] in 1994. Since then, various different versions and extensions have been developed. A recent survey by Waez et al. [171] lists eleven classes of timed automata and almost eighty concrete variants belonging to those classes. Within the different projects of this thesis, we considered different classes of timed automata. We will now define the most basic form of timed automata, and will afterwards go into detail on the different aspects that will vary in the remainder of this thesis.

The time domain we consider is the set $\mathbb{R}_{\geq 0}$ of non-negative reals. We denote by Σ the finite set of actions of an automaton. A *time sequence* is a finite non-decreasing sequence of non-negative reals. A *timed trace* σ is a finite alternating sequence of actions and time delays of the form $t_1 \cdot a_1 \cdots t_k \cdot a_k$, where for all $i \in [1, k]$, $a_i \in \Sigma$ and the accumulation of the delays $(t_1) \cdot (t_1 + t_2) \cdot (t_1 + t_2 + t_3) \cdots$ is a time sequence.

Let $\mathcal{C}$ be a finite set of *clock* variables. Clock *valuation* $v(c)$ is a function $v : \mathcal{C} \to \mathbb{R}_{\geq 0}$ assigning a real value to every clock $c \in \mathcal{C}$. We denote by $\mathcal{H}$ the set of all clock valuations and by **0** the valuation assigning 0 to every clock in $\mathcal{C}$. Let $v \in \mathcal{H}$ be a valuation and $t \in \mathbb{R}_{\geq 0}$, we then have $v + t$ defined by $(v + t)(c) = v(c) + t$ for all $c \in \mathcal{C}$. For a subset ρ of $\mathcal{C}$, we denote by $v[\rho]$ the valuation such that for every $c \in \rho$, $v[\rho](c) = 0$ and for every $c \in \mathcal{C} \backslash \rho$, $v[\rho](c) = v(c)$. A *clock constraint* φ is a conjunction of predicates over clock variables in $\mathcal{C}$ defined by the grammar

$$\varphi ::= c \circ k \mid \varphi_1 \wedge \varphi_2,$$

where $c \in \mathcal{C}$, $k \in \mathbb{N}$ and $\circ \in \{<, \leq, =, \geq, >\}$. Given a clock valuation $v \in \mathcal{H}$, we write $v \models \varphi$ when v satisfies the clock constraint φ. We are now ready to formally define basic timed automata (TA):

Definition 2.1
A TA A is a tuple $(Q, \hat{q}, \Sigma, \mathcal{C}, I, \Delta)$, where Q is a finite set of *locations*, $\hat{q} \in Q$ is the *initial* location, Σ is a finite set of *actions*, $\mathcal{C}$ is a finite set of *clock* variables, I is a finite set of location *invariants*, that are conjunctions of constraints of the form $c < d$ or $c \leq d$, where $c \in \mathcal{C}$ and $d \in \mathbb{N}$ and each invariant is bound to its specific location, and Δ is a finite set of *transitions* of the form (q, a, g, ρ, q'), sometimes denoted by $q \xrightarrow{a,g,\rho} q'$ where

- $q, q' \in Q$ are the *source* and the *target* locations;
- $a \in \Sigma$ is the transition action;
- g is a *guard*, a conjunction of constraints of the form $c \circ d$,
  where $\circ \in \{<, \leq, =, \geq, >\}$ and $d \in \mathbb{N}$;
- $\rho \subseteq \mathcal{C}$ is a set of clocks to be *reset*.

Example 2.1. Figure 2.1 shows the timed automaton $A = (Q, \hat{q}, \Sigma, \mathcal{C}, I, \Delta)$ of simple coffee machine, that upon receiving a coin first heats up and then either refunds the coin if it is empty, or grains, brews and releases the coffe. It consists of five locations, thus $Q = \{q_0, q_1, q_2, q_3, q_4\}$. Location q_0 is the initial location, thus $\hat{q} = q_0$. The set of actions Σ is $\{\mathit{coin?}, \mathit{beep!}, \mathit{refund!}, \mathit{coffee!}, \epsilon\}$. The automaton contains two clocks x and y, thus $\mathcal{C} = \{x, y\}$, and three invariants bound to the locations q_2, q_3 and q_4, thus $I = \{q_2 \leftarrow (x < 2), q_3 \leftarrow (y <= 1), q_4 \leftarrow (x < 4)\}$. The set of transitions Δ contains six transitions, s.t. $\Delta = \{(q_0, \mathit{coin?}, \mathit{true}, \{x\}, q_1), (q_1, \mathit{beep!}, 0 < x < 3, \{\}, q_2), (q_1, \mathit{beep!}, x = 2, \{\}, q_4), (q_2, \epsilon, 1 < x, \{y\}, q_3), (q_3, \mathit{coffee!}, y = 1, \{\}, q_1), (q_4, \mathit{refund!}, \mathit{true}, \{\}, q_1)\}$.

We define $|\mathcal{G}|$ to be the number of basic constraints that appear in all the guards of all the transitions in A, i.e. $|\mathcal{G}| = \Sigma_{\delta \in \Delta} |J_g|$, where $\delta = (q, a, g, \rho, q')$ and g is of the form $\bigwedge_{j \in J_g} c_j \circ d_j$. We define $|\mathcal{I}|$ as the number of basic constraints that appear in all the invariants of all the locations in A.

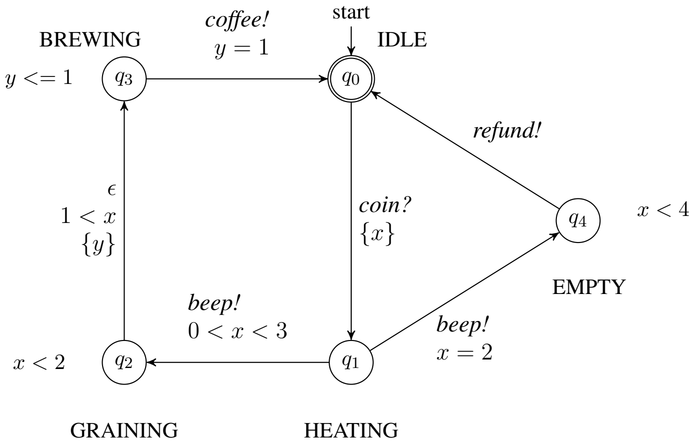


Figure 2.1: A timed automata model of a simple coffee machine.

The *semantics* of a TA $A = (Q, \hat{q}, \Sigma, \mathcal{C}, I, \Delta)$ is given by the *timed transition system* (TTS) $[[A]] = (S, \hat{s}, \mathbb{R}_{\geq 0}, \Sigma, T)$, where $S = \{(q, v) \in Q \times \mathcal{H} \mid v \models I(q)\}$, $\hat{s} = (\hat{q}, \mathbf{0})$, $T \subseteq S \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times S$ is the transition relation consisting of *discrete* and *timed* transitions such that:

- **Discrete transitions:** $((q, v), a, (q', v')) \in T$, where $a \in \Sigma$, if there exists a transition (q, a, g, ρ, q') in Δ, such that: (1) $v \models g$; (2) $v' = v[\rho]$ and (3) $v' \models I(q')$; and
- **Timed transitions:** $((q, v), t, (q, v + t)) \in T$, where $t \in \mathbb{R}_{\geq 0}$, if $v + t \models I(q)$.

A *run* r of a TA A is the sequence of alternating timed and discrete transitions of the form $(q_1, v_1) \xrightarrow{t_1} (q_1, v_1 + t_1) \xrightarrow{\delta_1} (q_2, v_2) \xrightarrow{t_2} \cdots$, where $q_1 = \hat{q}$, $v_1 = \mathbf{0}$ and $\delta_i = (q_i, a_i, g_i, \rho_i, q_{i+1})$, inducing the timed trace $\sigma = (t_1, a_1) \cdot (t_2, a_2) \cdots$. We denote by $\mathcal{L}(A)$ the language of the automaton, that is the set of timed traces induced by all runs of A.

2.1 Aspects of Timed Automata

In this subsection, we will define aspects of timed automata, that were relevant within the different areas of this thesis. The discussed aspects will be *silent transitions*, *input/output transitions*, *(non-) determinism*, *accepting locations*, *networks of timed automata*, *diagonal constraints* and a form of timed automata where we weakened some of the restrictions on guards and invariants. Note that we give the theory and some context on these topics during the remainder of the thesis, and only consider their formal definition here.

2.1.1 Timed Automata with Silent Transitions

Timed Automata with Silent Transitions (TA_ϵ) are a class of TA with an extended set of actions including *silent actions*, denoted by ϵ. These are internal actions that are *non-observable* from the outside, and we distinguish them from the actions that are not silent and called *observable actions*. We call a TA without silent transitions *fully-observable*. The definition and semantics of TA_ϵ are the same as those of TA, only that Σ is replaced by $\Sigma \cup \epsilon$, also denoted Σ_ϵ. The timed automaton from Example 2.1 contains one silent transition (the transition from q_2 to q_3).

A run r in an TA_ϵ may not be fully observable, thus inducing a not fully observable timed trace $\sigma = (t_1, a_1) \cdot (t_2, a_2) \cdots (t_k, \epsilon) \cdots$ defined over Σ_ϵ. We can extract an *observable timed trace* from σ, by removing all the pairs containing silent actions, while taking into account the passage of time.

Note that silent transitions can in general not be removed from a timed automaton, without changing the language of the TA$_\epsilon$. Details will be given in Sections 4.2 and Section 13.2.

2.1.2 Timed Automata with Inputs and Outputs

Timed Automata with Inputs and Outputs (TAIO) partition the set of actions Σ into two disjoints sets Σ^I and Σ^O of input and output actions, respectively. TAIO are similar to UPPAAL TA, which we use to illustrate our examples. One difference is that for simplicity of presentation we do not have *urgent* and *committed* locations. However, these types of locations are just syntactic sugar to make modeling easier, and can be expressed with standard timed automata. We denote by Δ_I the set of transitions labelled by inputs, and by Δ_O the set of transitions labelled by outputs. In case a TAIO also contains silent transitions, we denote it by TAIO$_\epsilon$. The timed automaton presented in Example 2.1 is a timed automaton with inputs and outputs, where $\Delta_I = \{\mathit{coin}?\}$ and $\Delta_O = \{\mathit{beep}!, \mathit{refund}!, \mathit{coffee}!\}$.

The partitioning into inputs and outputs allows modeling the communication with the environment. An input is marked with a question mark, and depicts an event that is received from the environment. An output, marked with an exclamation mark, is an emission from the system to the environment.

2.1.3 Timed Automata with Accepting Locations

Timed Automata with accepting locations partition the set of locations into accepting and non-accepting locations. While our test-case generation approach does not take accepting locations into account, we can use them during the bounded determinization, to prune branches that end in non-accepting locations. Since we want our test-cases to end with an observation, we could for instance mark locations that are reached via an input as non-accepting.

2.1.4 (Non-) Deterministic Timed Automata

We say that a TA A is *deterministic* if for all transitions (q, a, g_1, ρ_1, q_1) and (q, a, g_2, ρ_2, q_2) in Δ, $q_1 \neq q_2$ implies that $g_1 \wedge g_2 = false$. We denote by $\mathcal{A}$ the set of all TA and by Det$(\mathcal{A}) \subset \mathcal{A}$ its deterministic subset. The timed automaton in Example 2.1 contains non-determinism, as the two transitions labeled by $\mathit{beep}!$ leaving q_1 can be enabled at the same time, contain the same label and lead to different locations.

Note that we differ between non-determinism, where two transitions with the same label leave the same state, and *underspecification*, where different outputs may leave the same state. While our test-case generation does not support non-determinism without preprocessing it first to remove it, it is capable to process underspecification.

We will denote deterministic TA by DET(TA) and non-deterministic ones by NON-DET(TA). Note that, as with silent transitions, non-deterministic timed automata are strictly more expressive than deterministic ones, and can thus not be determinized in general.

2.1.5 Networks of Timed Automata

We define a *Network of Timed Automata with Inputs, Outputs and Silent Transitions* $N = (\mathcal{A}, \Sigma_e^I, \Sigma_e^O, \Sigma_i)$, where $\mathcal{A} = \{A_1, \ldots, A_n\}$ is a set of TAIO$_\epsilon$, $n = |\mathcal{A}|$ is the number of automata in the set, and the observable actions of the automata are split into three disjoint sets where Σ_e^I is the set of external input actions, Σ_e^O is the set of external output actions and Σ_i is the set of internal actions. Internal actions are exclusively used for synchronization between the TAIO$_\epsilon$, and the external output and input actions are exclusively used for communication with the environment. Formally, the three sets are defined as below:

$$\begin{array}{lcl} \Sigma_e^I & = & \{a \mid \exists\, A_m \in \mathcal{A} : a \in \Sigma_m^I \quad \wedge \quad \forall\, A_m \in \mathcal{A} : a \notin \Sigma_m^O\} \\ \Sigma_e^O & = & \{a \mid \exists\, A_m \in \mathcal{A} : a \in \Sigma_m^O \quad \wedge \quad \forall\, A_m \in \mathcal{A} : a \notin \Sigma_m^I\} \\ \Sigma_i & = & \{a \mid \exists\, A_m \in \mathcal{A} : a \in \Sigma_m^I \quad \wedge \quad \exists\, A_m \in \mathcal{A} : a \in \Sigma_m^O\} \end{array}$$

where Σ_m^I and Σ_m^O depict the inputs and outpus of the m-th automaton.

Example 2.2. Figure 2.2 shows a network of timed automata specifying three components of a coffee machine. The first machine handles the payment. It waits for the external input *coin?* and than triggers the internal action *paid!*, which is received by the second automaton. Upon receiving the internal action *paid?*, the second automaton waits for the external input *button?*, where the user may select a drink. Upon receiving, it triggers one of the internal actions *c!* and *t!*, which are received by the third automaton. The third automaton then brews either the tea or the coffe, and produces the corresponding external output signal. Then, it triggers the internal signal *ready!* for the first automaton. Formally, the different actions of the coffee machine are separated the following way: $\Sigma_e^I = \{coin, button\}$, $\Sigma_e^O = \{\mathit{coffee}, tea\}$, $\Sigma_i = \{t, c, paid, ready\}$ and ϵ. □

2.1.6 Diagonal Constraints

Diagonal constraints denote constraints in the guards or invariants of timed automata, that contain subtraction between clocks. Thus, were normally a guard is defined as a conjunction of constraints of the form $c \circ d$, where $\circ \in \{<, \leq, =, \geq, >\}$ and $d \in \mathbb{N}$, it may now be a conjunction of constraints of the form $f \circ f$ where $f = d \mid c \mid c_1 - c_2$, s.t. $d \in \mathbb{N}$ and $c, c_1, c_2 \in \mathcal{C}$.

It is known that diagonal constraints do not add to the expressiveness of timed automata and thus can be removed [29, 50]. However, the standard forward analysis of many tools does not work correctly for timed automata with diagonal constraints. Bouyer et al. [58] show how the standard analysis produces erroneous traces and how to patch it.

In the context of this thesis, we produce diagonal constraints during the silent transition removal and determinization. They do not pose a problem for us, as we work with SMT solving, but the resulting automata can not be processed by most of the other tools working with timed automata.

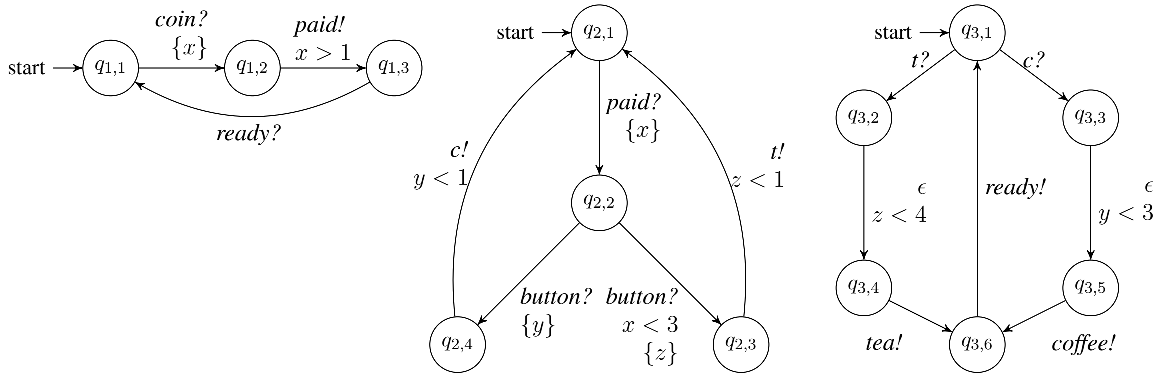


Figure 2.2: An NTA depicting a coffee machine with three components. One for payment, one for product selection and one for providing the drinks.

2.1.7 Timed Automata with Disjunction

Classic timed automata only allow the conjunction of constraints, as disjunction-free timed automata can be encoded in Difference Bound Matrices (DBM), which are supported by many tools. However, during the determinization process we propose, we gain timed automata with disjunction between the constraints.

Disjunction can easily be removed from timed automata. Looking at a single transition whose guard contains disjunction, the disjunction is resolved by generating one new transition for each of its clauses [50]. However, in our case that would remove the determinism property. Additionally, for model-checking via SMT-solving the disjunction does not pose a problem. Hence we keep the disjunctions.

2.2 Timed Input-Output Conformance

One of the main conformance relations for testing whether a system conforms to a formal model is the *input-output conformance* relation ioco introduced by Tretmans [160]. It considers an additional output, $quiescence$, denoting the absence of all other outputs. For real-time systems, this notion is not always sufficient, given that time delays of the implementations also have to conform to the specification.

Different real-time extensions of the input-output conformance relation ioco were studied and compared by Tretmans [150]. We consider the timed input-output conformance relation introduced by Krichen and Tripakis [117] and inspired by ioco. Intuitively, $\mathcal{A}_I$ conforms to $\mathcal{A}_S$ if for each observable behavior specified in $\mathcal{A}_S$, the possible outputs of $\mathcal{A}_I$ after this behavior is a subset of the possible outputs of $\mathcal{A}_S$. In contrast to ioco, tioco does not use the notion of quiescence, but requires explicit specification of timeouts. For the conformance check we consider TAIO without silent transitions, thus all actions are observable. Hence, we present a simplified version of the tioco definition from Krichen and Tripakis [117], first introducing auxiliary operators illustrated in Equation 2.1.

$$\begin{array}{lcl} A\,\text{after}\,\sigma & = & \{q \in Q \mid \hat{q} \xrightarrow{\sigma} q\} \\ \text{elapse(q)} & = & \{t > 0 \mid q \xrightarrow{t}\} \\ \text{out}(q) & = & \{a \in \Sigma^O \mid q \xrightarrow{a}\} \cup \text{elapse(q)} \\ \text{out}(Q) & = & \bigcup_{q \in Q} \text{out}(q) \end{array} \tag{2.1}$$

Given a TAIO A and $\sigma \in \mathcal{L}(\Sigma)$, $A\,\text{after}\ \sigma$ is the set of all locations of A that can be reached by the sequence σ. Given a location $q \in Q$, elapse(q) is the set of all delays that can elapse from q without A making any action, and out(q) is the set of all output actions or time delays that can occur when the system is at location q, a definition which naturally extends to set of locations Q.

Definition 2.2
The *timed input-output conformance relation*, denoted by tioco, is defined as

$$A_I\,\text{tioco}\,A_S \text{ iff } \forall \sigma \in \mathcal{L}(A_S) \,:\, \text{out}(A_I \text{ after } \sigma) \subseteq \text{out}(A_S \text{ after } \sigma)$$

Hence, an implementation is tioco-conform to a specification, if for all traces in the specification, the possible outputs of the implementation (including the passage of time) is a subset of the possible outputs in the specification. An implementation may add additional inputs, leaving the specified area. After that, every behavior is allowed.

Example 2.3. Figure 2.3 shows an example specification and three implementations. The first implementation I_1 conforms to the specification, even though it produces only a subset of the outputs of the specification, since it does not introduce new output behavior. The second implementation I_2 allows new input behavior that was not defined in the specification. This is also a conforming specification, because

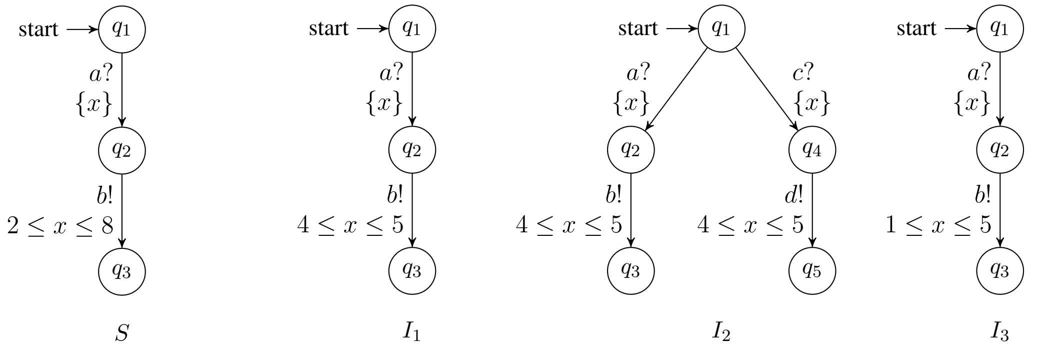


Figure 2.3: A specification and three implementations to illustrate the tioco conformance.

$tioco$ only considers the output behavior after all traces of the specification, allowing the implementations to add new inputs. Implementation I_3, however, does not conform to the specification, as it may produce the output $b!$ already when $x = 1$, and thus adds new output behavior. □

Krichen and Tripakis [117] gave a comparison of tioco to other timed conformance relations. Some of them, like timed bisimulation [154] and timed trace inclusion [108] completely differ from $tioco$, by being too restrictive for many applications. E.g., unspecified inputs may not occur in the implementation and outputs with stricter guards in the implementation than in the specification are not allowed either. This would reduce our support for partial models, and less abstract implementations. The *relativized timed conformance relation (rtioco)* [119] is defined with respect to an environment, which would restrict the test-case generation, only allowing inputs defined by that environment. This may well be useful, as some inputs may really be restricted in real tests. E.g. a button may only be pushed at most every second. However, if the defined environment changes, this might enforce a rerun of the test-case generation.

Another conformance relation that is very close to *tioco* is $\sqsubseteq_{tioco}$ [61]. The relation is very similar, but they use a notion of quiescence, adding quiescence self-loops to all silent states in the timed transition system. However, in practice it is not always possible to determine, whether a state is really quiescent, or whether one just did not wait long enough. They tackle this, by adding an upper bound M to all outputs, and calling states M-quiescent, if they do not trigger an output within M time units. However, we preferred $tioco$ as it keeps the timed transition systems used for the conformance check smaller.

3 Test-Case Generation

Parts of this chapter are based on our publication at TAP 2013 [17].

As discussed in Section 1.4, the goal of model-based mutation testing is the generation of a test suite according to a predefined set of fault models. The test cases are generated in a way that attempts to "steer" the SUT towards failure, if one exists. Hence, the rationale behind this approach is that if the mutated model does not conform to its original version, the mutation introduces traces which were not in the original model, and the non-conformance witness trace serves as the basis to generate a test case. In case that the mutated model conforms to its original version, the mutation does not introduce new behavior with respect to the original specification, hence no useful test case is generated. It follows that test cases are generated only if the mutated model does not conform to its original version. We propose a TCG algorithm, summarized as follows:

1. Given a deterministic TAIO A, a mutation operator m and a mutation function μ_m, generate the mutant $M \in \mu_m(A)$;
2. Generate $d(A)$ by demonic completion of A and $a(M)$ by angelic completion of M;
3. Check M tioco A, by effectively checking $\mathcal{L}(a(M)) \subseteq \mathcal{L}(d(A))$;
4. If $\mathcal{L}(a(M)) \not\subseteq \mathcal{L}(d(A))$, generate a test case based on the trace which witnesses non-conformance of M to A.

In the first step, discussed in Section 3.1, we mutate the original specification model. This creates a set of faulty models, called mutants, where each mutant contains exactly one fault, introduced at a specific location. Then, both the original model and the mutants are transformed to become input-enabled. This will be explained in detail in Section 3.2. The mutants need to be input-enabled during the *tioco*-conformance check, and the original model is required to be input-enabled in order to express the conformance check via language inclusion (as explained in Section 3.3). In the next step, we perform the k-bounded *tioco* conformance check: we will first explain, why the *tioco*-check can be expressed via language inclusion in Section 3.3.1. Then, in Section 3.3.2 we will show how to perform the language inclusion check via bounded model-checking using an SMT-sover. If the language of a mutant is included in the language of the original model, it is considered an equivalent mutant and discarded. Otherwise, we generate a test case out of the trace we get from the SMT-solver, as discussed in Section 3.4. Once the methodology is explained, we will give details on the implementation and its restrictions in Sections 3.5 and 3.6.

In the current chapter we consider TAIO, as the separation into inputs and outputs plays an important role in the context of testing. While some of the mutation operators could also be applied to TA, the *tioco*-conformance check strictly relies on the inputs and outputs. Additionally, the approach is restricted to deterministic TAIO.

3.1 Model Mutation

Mutation of a specification consists in altering the model in a small way, mimicking common implementation errors. In our setting, a mutation is a function $\mu_m : \text{Det}(\mathcal{A}) \to 2^{\mathcal{A}}$ parameterized by a mutation operator m which maps a deterministic TAIO A into a finite set $\mu_m(A)$ of possibly non-deterministic TAIOs, where each $M \in \mu_m(A)$ is called an m-mutant of A. For our experiments we only created first-order mutants, i.e., each mutated TAIO covers only one particular mutation.

We now introduce and define specific mutation operators which are relevant to the TAIO model.

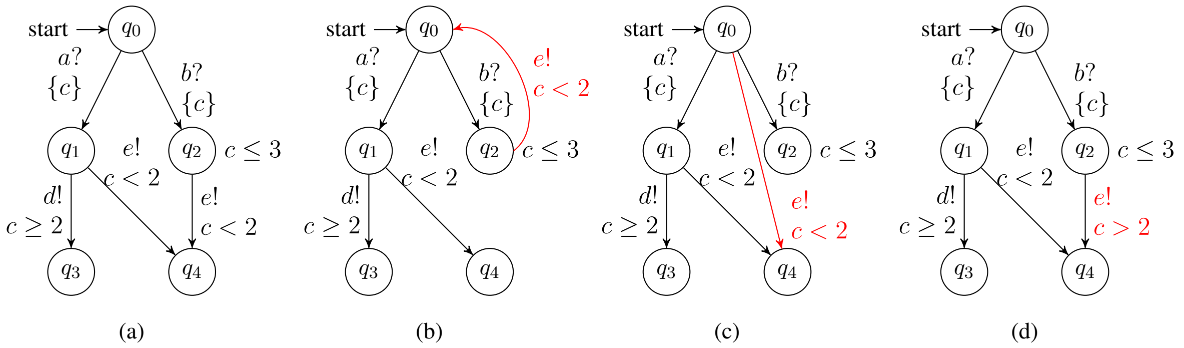


Figure 3.1: Model A (a) and mutants M resulting from: (b) $\mu_{ct}(A)$; (c) $\mu_{cs}(A)$; (d) $\mu_{cg}(A)$.

Definition 3.1
Given a TAIO $A = (Q, \hat{q}, \Sigma^I, \Sigma^O, \mathcal{C}, I, \Delta)$, its mutants are defined by the following *mutation operators*:

1. **Change action** (μ_{ca}) generates from A a set of $|\Delta_I|(|\Sigma^O|) + |\Delta_O|(|\Sigma^O| - 1)$ mutants, where every mutant changes a single transition in A by replacing the action labeling the transition by a different output label. This mimics an implementation fault producing wrong output signals. A TAIO $M \in \mu_{ca}(A)$, if M is of the form $(Q, \hat{q}, \Sigma^I, \Sigma^O, \mathcal{C}, I, (\Delta \backslash \{\delta\}) \cup \{\delta_m\})$, such that $\delta = (q, a, g, \rho, q') \in \Delta$, $\delta_m = (q, a_m, g, \rho, q')$, $a_m \in \Sigma^O$ and $a_m \neq a$;
2. **Change target** (μ_{ct}) generates from A a set of $|\Delta|(|Q| - 1)$ mutants, where every mutant replaces the target location of a transition in A, by another location in A. This reflects the behavior of an implementation fault where a signal leads to a wrong internal state. Note that the target location may also be replaced be the source location, creating a self-loop. A TAIO $M \in \mu_{ct}(A)$, if M is of the form $(Q, \hat{q}, \Sigma^I, \Sigma^O, \mathcal{C}, I, (\Delta \backslash \{\delta\}) \cup \{\delta_m\})$, such that $\delta = (q, a, g, \rho, q') \in \Delta$, $\delta_m = (q, a, g, \rho, q'_m)$, $q'_m \in Q$ and $q'_m \neq q'$;
3. **Change source** (μ_{cs}) generates from A a set of $|\Delta|(|Q| - 1)$ mutants, where every mutant replaces the source location of a transition in A, by another location in A. This expresses an implementation fault where a signal can be triggered from a state where it should be disabled. A TAIO $M \in \mu_{cs}(A)$, if M is of the form $(Q, \hat{q}, \Sigma^I, \Sigma^O, \mathcal{C}, I, (\Delta \backslash \{\delta\}) \cup \{\delta_m\})$, such that $\delta = (q, a, g, \rho, q') \in \Delta$, $\delta_m = (q_m, a, g, \rho, q')$, $q_m \in Q$ and $q_m \neq q$;
4. **Change guard** (μ_{cg}) generates from A a set of $4|\mathcal{G}|$ mutants, where every mutant replaces a transition in A with another one which changes the original guard by altering every equality/inequality sign appearing in the guard by another one. This covers implementation faults with faulty enabling conditions. A TAIO $M \in \mu_{cg}(A)$, if M is of the form $(Q, \hat{q}, \Sigma^I, \Sigma^O, \mathcal{C}, I, (\Delta \backslash \{\delta\}) \cup \{\delta_m\}))$, such that $\delta = (q, a, g, \rho, q') \in \Delta$, $\delta_m = (q, a, g_m, \rho, q')$, $g = \bigwedge_{i \in I} c_i \circ_i d_i$, $g_m = \bigwedge_{i \in I} c_i \circ_i^m d_i$, $\circ, \circ_i^m \in \{<, \leq, =, \geq, >\}$, $\circ_i \neq \circ_i^m$ for some $i \in I$ and $\circ_j = \circ_j^m$ for all $j \neq i$;
5. **Negate guard** (μ_{ng}) generates from A a set of $|\Delta|$ mutants, where every mutant replaces the guard in a transition in A, by its negation. This covers implementation faults where the programmer forgot negating a condition. A TAIO $M \in \mu_{ng}(A)$, if M is of the form $(Q, \hat{q}, \Sigma^I, \Sigma^O, \mathcal{C}, I, (\Delta \backslash \{\delta\}) \cup \{\delta_m\})$, such that $\delta = (q, a, g, \rho, q') \in \Delta$ and $\delta_m = (q, a, \neg g, \rho, q')$. Note that for the sake of simplicity, we represent δ_m as a single transition even though $\neg g$ may also have disjunctions. The guard $\neg g$ can be represented in DNF and every disjunction of the guard can be used as a guard of a separate transition.
6. **Change invariant** (μ_{ci}) generates from A a set of $|\mathcal{I}|$ mutants, where every mutant replaces the invariant of a location with another invariant with 1 added to the right-hand side of the basic

constraint of the invariant. This mimics an ”off by one”-fault allowing to stay longer in a state than intended. A TAIO $M \in \mu_{ci}(A)$, if M is of the form $(Q, \hat{q}, \Sigma^I, \Sigma^O, \mathcal{C}, I_m, \Delta)$, and there exists $q \in Q$ such that $I(q) = \bigwedge_{i \in I} c_i \circ d_i$, $\circ \in \{<, \leq\}$, $I_m(q) = \bigwedge_{i \in I} c_i \circ d_i^m$, $d_i^m = d_i + 1$ for some $i \in I$, $d_j^m = d_j$ for all $j \neq i$ and $I(q') = I_m(q')$ for all $q' \neq q$;

7. **Sink location** (μ_{sl}) generates from A a set of $|\Delta|$ mutants, where every mutant replaces the target location of a transition in A, by a newly created sink location which models a don’t care location which accepts all inputs. This expresses a program fault leading to a quiescent state where every input is accepted, but ignored. A TAIO $M \in \mu_{sl}(A)$, if M is of the form $(Q \cup \{\text{sink}\}, \hat{q}, \Sigma^I, \Sigma^O, \mathcal{C}, I, (\Delta \backslash \{\delta\}) \cup \{\delta_m\} \cup \Delta_{\text{sink}})$, such that $\Delta_{\text{sink}} = \{(\text{sink}, a, \text{true}, \{\}, \text{sink}) \mid a \in \Sigma^I\}$, $\delta = (q, a, g, \rho, q') \in \Delta$ and $\delta_m = (q, a, g, \rho, \text{sink})$;
8. **Invert reset** (μ_{ir}) generates from A a set of $|\Delta||\mathcal{C}|$ mutants, where every mutant replaces a transition in A, by another transition with the occurrence of one clock flipped compared to the original set of clocks. This reflects different timing errors, e.g. the incorrect reseting of a timer. A TAIO $M \in \mu_{cs}(A)$, if M is of the form $(Q, \hat{q}, \Sigma^I, \Sigma^O, \mathcal{C}, I, (\Delta \backslash \{\delta\}) \cup \{\delta_m\})$, such that $\delta = (q, a, g, \rho, q') \in \Delta$, $\delta_m = (q, a, g, \rho_m, q')$, and for some $c \in \mathcal{C}$ either $\rho_m = \rho \cup \{c_m\}$ if $c_m \notin \rho$, or $\rho_m = \rho \backslash \{c_m\}$ if $c_m \in \rho$.

Figure 3.1 illustrates mutants resulting from applying some of the above mutation operators to an example model A. The effectiveness of the mutation operators is analyzed and evaluated in more detail in Chapter 5.

3.2 Model Completion

Prior to performing the conformance check, both the specification and the model need to be made input-enabled. The input-enabledness of the mutants is required by the definition of the *tioco* conformance, as discussed in Section 2.2. The reason why we need the specification to be input-enabled as well, is given in the next section. However, input-enabledness can be achieved by two different operations, angelic and demonic completion and we need to apply a different kind of completion to the specification than to the mutants.

The specification is supposed to contain all behavior that is supposed to be tested. Even if it is only a partial model of a more complex systems, we only want to test the specified part. Thus, if an input event that was not specified by our model occurs during testing, we leave the specified area and cannot predict the future behavior of the system anymore, meaning that from then on, all inputs and outputs are valid. Thus, input-enabling of our specification model is done by *demonic completion* [161, 117]: we create a new location, called *sink state* in the automaton, which contains self-loop transitions for every possible input or output. Thus, in that location any behavior is possible and allowed. Then, for each location in the automaton and for each input that is not allowed in that location, we create a transition labeled by that input, leading to the sink state.

The formal definition of demonic completion is the following: given a deterministic TAIO $A = (Q, \hat{q}, \Sigma^I, \Sigma^O, \mathcal{C}, I, \Delta)$, its demonic completion $d(A)$ is the input-enabled TAIO $d(A) = (Q \cup \{\text{sink}\}, \hat{q}$ $, \Sigma^I, \Sigma^O, \mathcal{C}, I_d, \Delta_d)$, where $I_d(q) = I(q)$ and $I_d(\text{sink}) = \text{true}$ and $\Delta_d = \Delta \cup \{(\text{sink}, a, \text{true}, \{\}, \text{sink}) \mid a \in \Sigma\} \cup \{(q, a, \neg g, \{\}, \text{sink}) \mid q \in Q \wedge a \in \Sigma^I\}$, such that for each $q \in Q$ and $a \in \Sigma^I$, $g = (g_1 \vee \ldots \vee g_k) \wedge I(q)$, where $\{g_i\}_i$ are guards of the outgoing transitions of q labeled by a. Strictly speaking, g contains disjunctions of constraints, and thus cannot be directly used as a guard on a transition in a TAIO. In fact, we would need to transform g into a disjunctive normal form, and have a separate copy of the transition for each disjunction labeled by the appropriate guard. We omit the details of this transformation. It is not hard to see that

$$\mathcal{L}(d(A)) = \mathcal{L}(A) \cup \{\sigma \cdot t \cdot a \cdot (\mathbb{R}_{\geq 0} \cdot \Sigma)^* \mid a \in \Sigma^I, \sigma \in \mathcal{L}(A) \wedge \sigma \cdot a \notin \mathcal{L}(A)\}$$

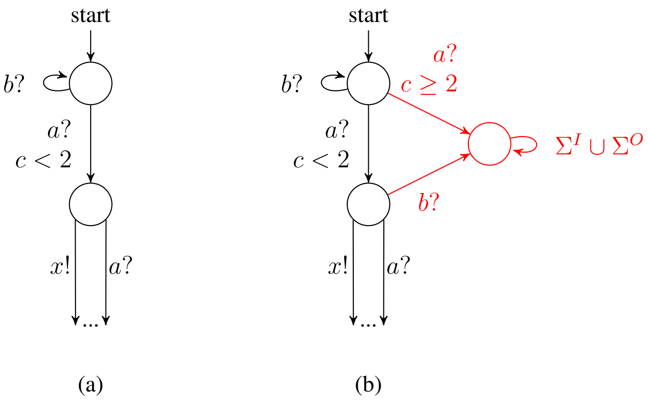


Figure 3.2: Input completion of TAIO: (a) A; (b) $d(A)$; and (c) $a(A)$.

An example is illustrated in Figure 3.2, where (a) is a specification, (b) shows its demonic completion and (c) shows its angelic completion. We marked the sink state and the new transitions in red.

We apply angelic completion [161] to the mutants: input output conformance checks are usually applied between a specification, and its implementation. The implementations are considered input-enabled, which in praxis means that they do not block any inputs, but simply skip all unexpected inputs. In our case, the mutants are considered as implementations. Thus, to model undefined inputs, it suffices to create self-loop transitions for all unspecified inputs in each location.

3.3 Conformance Check

We already introduced the tioco conformance relation in Section 2.2, and discussed several alternatives and our reasons for choosing tioco. In this section we will first show how *tioco* can be expressed via language inclusion, and then how we solve it via SMT-solving.

3.3.1 Conformance Expressed via Language Inclusion

As a short reminder on the definition of tioco, we give an informal definition here: an implementation A_I tioco conforms to its specification A_S, if after all observable traces in A_S, the set of possible outputs of A_I (including the passage of time) is a subset of the possible outputs of A_S.

Krichen and Tripakis. [117] develop a number of theoretical results about the tioco relation. In particular, they establish that given two TAIO A_I and A_S, if the set of observable traces of A_I is included in the set of observable traces of A_S, then A_I tioco A_S, while the converse is not true in general. However, if A_S is input-enabled, then the set inclusion between observable traces of A_I and A_S also implies the tioco conformance of A_I to A_S.

Given an arbitrary TAIO A_I and a deterministic specification TAIO A_S, considering the demonic completion $d(A_S)$ instead of A_S does not affect the conformance relation. Formally, we have the following proposition, proved by Krichen and Tripakis [117].

Proposition 3.1
Given a deterministic TAIO A_S *and its demonic completion* $d(A_S)$*, for any* TAIO A_I*,* A_I *tioco* A_S *if and only if* A_I *tioco* $d(A_S)$.

It turns out that given two TAIO A_S and A_I, by applying demonic completion $d(A_S)$ to A_S, checking tioco of A_I to A_S is equivalent to checking the language inclusion $\mathcal{L}(A_I) \subseteq \mathcal{L}(d(A_S))$, a result stated

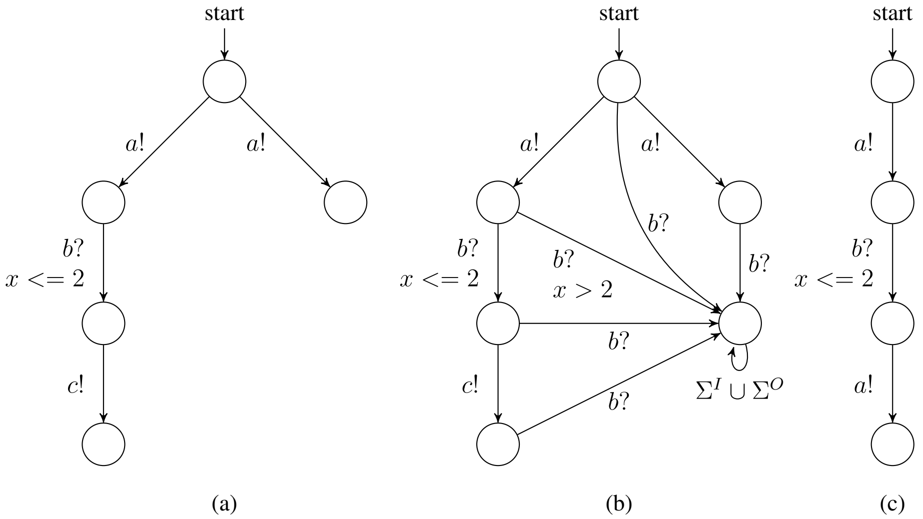


Figure 3.3: A non-deterministic specification (a), its demonic completion (b) and an implementation (c).

in the next Proposition, and which follows from the work by Krichen and Tripakis [117] (Lemma 3 and Proposition 3).

Proposition 3.2
Given a TAIO A_I *and a deterministic* TAIO A_S, A_I *tioco* A_S *if and only if* $\mathcal{L}(A_I) \subseteq \mathcal{L}(d(A_S))$.

By Proposition 3.2, it follows that one can check $\mathcal{L}(A_I) \subseteq \mathcal{L}(d(A_S))$ instead of checking A_I tioco A_S when A_S is deterministic. In addition, the problem of checking $\mathcal{L}(A_I) \subseteq \mathcal{L}(d(A_S))$ is decidable when A_S is deterministic [29].

Example 3.1. We will now present a short example to show why this does not hold for non-deterministic specifications. Figure 3.3 shows a non-deterministic specification A_S, its demonic completion $d(A_S)$ and an implementation A_I. In deterministic systems, for every $\sigma \in A_S$ it holds that out(A_S after σ) = out($d(A_S)$ after σ). In our non-deterministic example, the trace $a! \cdot 0 \cdot b?$ is specified in A_S, in its left branch, and it only allows the output $c!$. However, in the demonic completion of A_S, the trace $a! \cdot 0 \cdot b?$ is also a trace in the right branch, leading to the sink state and thus allowing the output $a!$. Thus, out(A_S after $a! \cdot 0 \cdot b?$) = $\{c!\}$ and out($d(A_S)$ after $a! \cdot 0 \cdot b?$) = $\{a!, c!\}$ and thus the demonic completion of A_S is not tioco conform to A_S. Now, the language of implementation A_I is a subset of the language of $d(A_S)$, but is not tioco conform to A_S, which violates Proposition 3.2.

3.3.2 Language Inclusion as a Bounded Model-Checking Problem

We have seen that mutation-based testing is fault-oriented, i.e. test cases are generated only if the mutated model does not conform to its original version. Consequently, symbolic techniques based on bounded model-checking (BMC) are well-suited to solve this type of problems. In addition, the language inclusion problem between two timed automata A_I and A_S, where A_S is deterministic, is PSPACE-complete (This result was already established by Dill and Alur [29]), hence computationally expensive. In our setting, we are interested in finding finite counter-example traces witnessing violation of language inclusion. Missing such a witness, due to an insufficient bound, results in generating less test cases and is a trade-off between generating a complete test suite and computing it efficiently.

BMC was already used for the reachability analysis of TA [36, 133], and for checking the language inclusion between two timed automata [38]. We encode the language inclusion problem as a k-bounded language inclusion SMT problem. Intuitively, given two TAIO A_I and A_S such that A_S is deterministic and an integer bound k, we have $\mathcal{L}(A_I) \not\subseteq^k \mathcal{L}(A_S)$ if there exists a timed trace $\sigma = t_1 \cdot a_1 \cdots t_i \cdot a_i$ such that $i \leq k$, $\sigma \in \mathcal{L}(A_I)$ and $\sigma \notin \mathcal{L}(A_S)$. We construct a formula $\varphi^k_{A_I,A_S}$ that is satisfiable if and only if $\mathcal{L}(A_I) \not\subseteq^k \mathcal{L}(A_S)$.

Let $A = (Q, \hat{q}, \Sigma^I, \Sigma^O, \mathcal{C}, I, \Delta)$ be a TAIO. We denote by $\text{loc}_A : Q \to \{1, \dots, |Q|\}$ and $\text{act}_A : \Sigma \to \{1, \dots, |\Sigma|\}$ functions assigning unique integers to locations and actions in A, respectively. Given A and a constant k, we denote by X the set of variables $\{x^1, \dots, x^{k+1}\}$ that range over the domain $\{1, \dots, |Q|\}$, where x^i encodes the location of A after the i^{th} step. Similarly, let $\mathcal{A} = \{\alpha^1, \dots, \alpha^k\}$ be the set of variables ranging over $\{1, \dots, |\Sigma|\}$, where α^i encodes the action in A applied in the i^{th} discrete step. We denote by $D = \{d^1, \dots, d^k\}$ the set of real-valued variables, where d^i encodes the delay applied in the i^{th} time step. We will call this delay a *delay action*. Let C^i denote the set of real variables obtained by renaming every clock $c \in \mathcal{C}$ by c^i. We denote by $C = \bigcup_{i=1}^{k+1} C^i \cup \bigcup_{i=1}^{k+1} C^{*,i}$ the set of real (clock valuation) variables, where $c^{*,i} \in C^{*,i}$ and $c^i \in C^i$ encode the valuation of the clock $c \in \mathcal{C}$ after the i^{th} timed and discrete step, respectively.

We express the effect of applying Reset_ρ in the i^{th} step of a run to the set $\mathcal{C}$ of clocks in A as follows:

$$\text{doReset}^i_{A,\rho}(C) \equiv \bigwedge_{c \in \rho} c^{i+1} = 0 \wedge \bigwedge_{c \notin \rho} c^{i+1} = c^{*,i}$$

We express the i^{th} passage of time in A as follows:

$$\text{tDelay}^i_A(D, C) \equiv \bigwedge_{c \in \mathcal{C}} (c^{*,i} - c^i) = d^i$$

The i^{th} time step in a location $q \in Q$ is expressed with:

$$\text{tStep}^i_{A,q}(D, X, C) \equiv x^i = \text{loc}_A(q) \wedge \text{tDelay}^i_A(D, C) \wedge I(q)[\mathcal{C} \backslash C^{*,i}],$$

where $I(q)[\mathcal{C} \backslash C^{*,i}]$ is the invariant of q, with every clock $c \in \mathcal{C}$ substituted by $c^{*,i}$. The formula for the i^{th} discrete step is:

$$\begin{aligned} \text{dStep}^i_{A,\delta}(\mathcal{A}, X, C) \quad \equiv \quad & x^i = \text{loc}_A(q) \wedge \alpha^i = \text{act}_A(a) \wedge g[\mathcal{C} \backslash C^{*,i}] \quad \wedge \\ & \text{doReset}^i_{A,\rho}(C) \wedge x^{i+1} = \text{loc}_A(q') \end{aligned}$$

where $g[\mathcal{C} \backslash C^{*,i}]$ denotes the guard of δ, where every clock $c \in \mathcal{C}$ is substituted by $c^{*,i}$. We express the segment of a path in TAIO A from j to k with the following formula:

$$\text{path}^{j,k}_A(\mathcal{A}, D, X, C) \equiv \bigwedge_{i=j}^{k} (\bigvee_{q \in Q} \text{tStep}^i_{A,q}(D, X, C) \wedge \bigvee_{\delta \in \Delta} \text{dStep}^i_{A,\delta}(\mathcal{A}, X, C))$$

The initial state of TAIO A is expressed as follows:

$$\text{init}_A(X, C) \equiv x^1 = \text{loc}_A(\hat{q}) \wedge \bigwedge_{c \in \mathcal{C}} (c^1 = 0)$$

Let $A_I = (Q_I, \hat{q}_I, \Sigma^I, \Sigma^O, \mathcal{C}, I_I, \Delta_I)$ and $A_S = (Q_S, \hat{q}_S, \Sigma^I, \Sigma^O, \mathcal{C}, I_S, \Delta_S)$ be two TAIOs such that A_S is deterministic. The general formula $\varphi^k_{A_I,A_S}(i, \mathcal{A}, D, X_I, X_S, C_I, C_S)$ specifies the negation of k-language inclusion:

$$\begin{aligned} \varphi^k_{A_I,A_S} \quad \equiv \quad & \textstyle\bigwedge_{i=1}^{k} (d^i \geq 0 \wedge \alpha^i \geq 1 \wedge \alpha^i \leq |\Sigma|) \wedge i \geq 1 \wedge i \leq k \quad \wedge \\ & \text{init}_{A_I}(X_I, C_I) \wedge \text{init}_{A_S}(X_S, C_S) \wedge \text{path}^{1,i}_{A_I}(\mathcal{A}, D, X_I, C_I) \quad \wedge \\ & \text{path}^{1,i-1}_{A_S}(\mathcal{A}, D, X_S, C_S) \wedge \neg\text{path}^{i,i}_{A_S}(\mathcal{A}, D, X_S, C_S) \end{aligned}$$

Thus, we can find a counter example, if there exists a variable valuation for all d^i and α^i so that (1) $1 \leq i \leq k$; (2) the initial invariant holds and all clocks are initially reset to 0, (3) there exists a path up to step $i-1$ both in the specification and the implementation and finally (4) there exists an i-th step in the implementation, but not in the specification. Note that α^i can only be an output, as the demonic completed specification can perform all inputs in all states.

3.4 Generating the Test Case

Given a specification model A and its mutant M, our test-case generation algorithm creates a *test* only if M does not conform to A. The generated test follows a *test purpose*, which is in our case the timed trace σ which witnesses the non-conformance of M to A and exposes the error caused by the mutation in M. We denote a test by A_T and give it in a form of a deterministic TAIO. The test A_T specifies the execution of real-time traces and provides a *verdict* after observing at most k combined (timed/discrete) steps of a trace. The verdict can be:

- *Pass* (**pass**) - if the test purpose was successfully reached and the error introduced by the mutant was not exposed by the SUT during the test execution;
- *Inconclusive* (**inc**) - if the test purpose covering the fault introduced by the mutant could not be reached by the SUT during the test execution. This can happen, as test procedure supports under-specified models in the sense that several outputs may leave the same state and the system under test may produce a different output than the one expected by our test purpose;
- *Fail* (**fail**) - if the fault introduced by the mutant as part of the test purpose was exposed by the SUT during the test execution or in the last step.

The skeleton of A_T consists of the sequence $q_1 \cdot \delta_1 \cdots q_k \cdot \delta_k$ of locations and transitions in A which are executed while observing the witness trace $\sigma = t_1 \cdot a_i \cdots t_k \cdot a_k$. This skeleton corresponds effectively to the test purpose described above. In addition, A_T is completed according to Algorithm 1 satisfying a number of properties described next. After observing a prefix $\sigma' = t_1 \cdot a_1 \cdots t_i \cdot a_i$ of σ, A_T is in location q_i, where $i < k$, and can do one of the following:

- Wait if the invariant of q_i allows a positive time delay;
- Emit action a if a is an input action equal to a_i and the transition δ_i is enabled, and move to location q_{i+1} (this step is implicit in the algorithm, by not entering any of the if-conditions and jumping to the next step of the for-loop);
- Accept action a if a is an output action equal to a_i and the transition δ_i is enabled, and move to location q_{i+1} (this step is implicit in the algorithm, by not entering any of the if-conditions and jumping to the next step of the for-loop);
- Accept action a if a is an output action different from a_i and there exists an enabled transition δ in A with source location q_i and labeled with a, and move to the **inc** verdict location (Line 7);
- Refuse action a if a is an output action and there are no transitions in A with the source location q_i which is both labeled by a and enabled, and move to the **fail** verdict location (Lines 12-18).

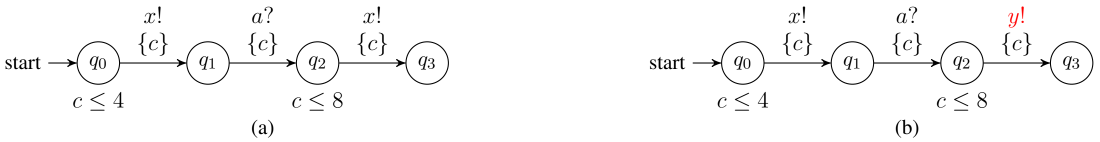
Finally, when A_T is in location q_k, it accepts all outputs a such that there exists an enabled transition δ in A with source location q_k and labeled by a, moving to the **pass** location (Line 9), and it rejects all other outputs, moving to the **fail** location (Lines 12-18).

Note that our test A_T follows a fixed qualitative sequence of actions, defined by the witness σ. In particular, it stops following a valid output in the specification A if it differs from the one in the witness

Algorithm 1 Test-case generation algorithm.

```
Input: $A = (Q, \hat{q}, \Sigma^I, \Sigma^O, \mathcal{C}, I, \Delta)$ and $\delta_1 \cdots \delta_k$
Output: Test automaton $A_T$
$\Delta_T \leftarrow \bigcup_{i=1}^{k-1}\{\delta_i\}$
$Q_T \leftarrow \{q_i | (q_i, a, g, p, q') \in \delta_1 \cdots \delta_k\}$
$Q_T \leftarrow Q_T \cup \{$pass, fail, inc$\}$
for $i = 1$ to $k$ do
    for all $(q_i, a, g, \rho, q') \in \Delta \backslash \{\delta_i\}$ st. $a \in \Sigma_O$ do        ▷ For all outputs not in $\delta_i$
        if $i < k$ then                                                         ▷ During test case
            $\Delta_T \leftarrow \Delta_T \cup \{(q_i, a, g, \{\},$ inc$)\}$
        else                                                                    ▷ At the end of the test
            $\Delta_T \leftarrow \Delta_T \cup \{(q_i, a, g, \{\},$ pass$)\}$
        end if
    end for
    for all $a \in \Sigma^O$ st. $\exists(q_i, a, g_j, \rho, q') \in \Delta$ do                ▷ All outputs that are not enabled
        $g_T \leftarrow (g_1 \vee \ldots \vee g_n) \wedge I(q)$ st. $\{g_j\}$ are guards of outgoing transitions from $q_T$ labeled by $a$
        $\Delta_T \leftarrow \Delta_T \cup \{(q_i, a, \neg g_T, \{\},$ fail$)\}$
    end for
    for all $a \in \Sigma^O$ st. $\nexists(q_i, a, g, \rho, q') \in \Delta$ do                 ▷ All unspecified outputs
        $\Delta_T \leftarrow \Delta_T \cup \{(q_i, a,$ true$, \{\},$ fail$)\}$
    end for
end for
return $A_T \leftarrow (Q_T, \hat{q}, \Sigma^O, \Sigma^I, \mathcal{C}, I, \Delta_T)$
```

σ, and returning **inc** as verdict. It means that the test is not pursued when the SUT deviates from the test purpose. On the other hand, A_T is time adaptive, and the witness σ defines a class of timing constraints which are allowed by the test. In fact, it is unlikely that an expected output action is preceded by the exact time delay as defined by the witness trace. Hence, we need the test to be flexible and accept the expected output in a larger time range defined by the specification model. In addition, if we allow time flexibility for output actions, we cannot use the strict time delay from the witness trace σ, to precede an input action either, since it may violate input assumptions of the specification during some test execution. We illustrate this observation in Figure 3.4, which depicts model A and its mutant M. The trace $\sigma = 4 \cdot x! \cdot 2 \cdot a? \cdot 2 \cdot y!$ witnesses non-conformance of M to A and is used as the skeleton for the test A_T. During test execution, the test may observe the prefix $\sigma' = 2 \cdot x!$, which is allowed by the specification. In that case, if A_T requires exactly 2 time units to elapse between observing $x!$ and emitting $a?$, the assumptions expressed by A are violated. Hence we keep the time constraints symbolic, with an elapse of time between $x!$ and $a?$ dependent on previous observations.



(a)

(b)

Figure 3.4: Necessity of symbolic constraints on inputs in a test.

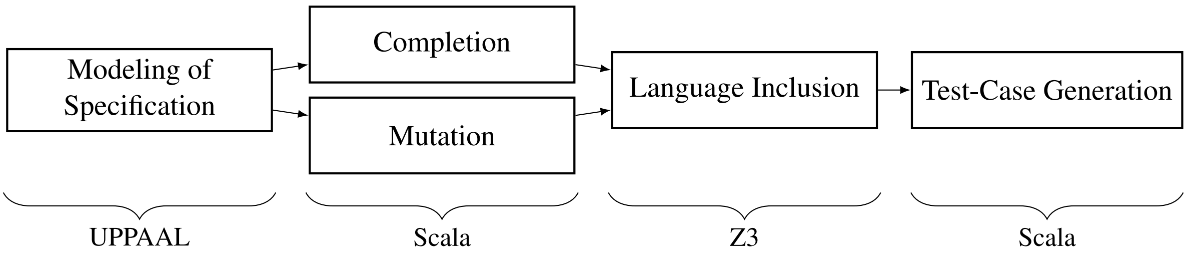


Figure 3.5: Test-case generation framework.

3.5 Implementation

In this section, we present the tool that implements the test-case generation framework. Actually, in the course of the thesis, the tool was implemented twice. The first prototype implementation was done in Scala version 2.9.1, the second, more mature, reimplementation was done in Scala version 2.10.3. Both use standard UPPAAL TA XML format to model TAIO specifications. The (bounded) language inclusion between two TAIOs is computed using the Z3 (v4.0) SMT solver [73]. The communication between our implementation in Scala and the Z3 solver relies on the Scala^Z3 API [110].

The author of this thesis developed the prototype tool, while the reimplementation was done by Willibald Krenn [113] and became a tool in the MoMuT family of tools, called MoMuT::TA[1]. The determinization process introduced in the next chapter was added to the reimplementation be the author of this thesis. After adding the determinization, the tool consisted of 55 classes and about $20,000$ lines of code, though that also includes features not discussed within this thesis.

The test-case generation framework, depicted in Figure 3.5, consists of four main steps: (1) parsing and demonic completion of the TAIO model; (2) mutation of the TAIO model; (3) language inclusion between the original model and its mutant; and (4) test-case generation. In what follows, we present more details about these steps.

Specification Parsing and Demonic Completion: the TAIO model specified in the UPPAAL XML format is parsed with Scala's parser combinator. We require that the UPPAAL automata do not contain urgent nor committed locations. It should be noted that modeling style can have important impact on the number and effectiveness of consecutive generation of mutants and test cases, as shown by Tiran [159] for UML models. We implemented demonic completion of the model by direct application of the procedure from Section 3.2.

Mutation of Models: our tool supports all mutation operators introduced in Section 3.1. We store each mutant as a separate UPPAAL XML model.

Language Inclusion: language inclusion check between a model and its mutant is at core of the TCG framework. We translate an UPPAAL model and its mutant to a bounded language inclusion problem expressed as an SMT-LIB2 formula, following the procedure described in Section 3.3. The formula is fed to the Z3 solver, which looks for the existence of a satisfying assignment to the variables representing a witness trace violating the language inclusion property.

In addition, we implemented the same TCG algorithm using Z3's incremental solving feature, with the aim to improve the computation time of the bounded language inclusion check. Given an SMT formula expressing the k-bounded language inclusion problem, we first feed the Z3 solver with the sub-formula for the i-bounded language inclusion problem, starting with $i = 1$. Z3 checks the satisfiability of the sub-formula, and if a satisfying assignment is found, the procedure stops. Otherwise, we pop

[1] `https://momut.org/?page_id=355`

the sub-formula from the Z3 stack and push the sub-formula expressing the step from i to $i + 1$. The procedure is iterated until a witness is found or the k bound is reached.

Test-case generation: if Z3 generates a counter-example which witnesses violation of language inclusion between the specification and its mutant, we use this counter-example together with the specification model in order to generate a test case. The test-case generation implementation closely follows Algorithm 1.

3.6 Restrictions

The main restriction of the algorithm and tool described in this chapter is the limitation to fully-observable deterministic automata. By composing different models and hiding their communication, one often produces non-deterministic specification models. Consequently, non-determinism is a desired model element, which can not simply be neglected. Thus, to overcome this restriction, we developed a bounded determinization procedure, which also removes silent transitions. Details can be found in Chapter 4. The approach presented in the current chapter does, however, support underspecification in the sense that we allow states with multiple different outputs. This allows the definition of an underspecified model, which grants implementation freedom to the designer who actually builds the system. He may then decide which of the choices will actually be implemented.

Other restrictions are:

- **State-Space Explosion.** As can be seen in Chapter 5, increasing the bound of the bounded model-checking significantly increases the runtime of the test-case generation procedure, due to the increasing state-space that needs to be examined. As already mentioned, BMC tackles parts of that problem, as the search for a concrete counter example needs not necessarily traverse the complete state-space, if it is found early. Another way to tackle this problem is the usage of partial models, as tioco supports them and they can be analysed independently. If the system still is too complex, the search depth needs to be reduced. As already mentioned, this provides a fair tradeoff between generating a complete test suite and computing it efficiently.
- **High Number of Mutants.** The high number of mutants may pose a problem for efficiency as well. By applying all proposed mutations to all transitions/locations in a system one may easily produce over thousand mutants for small to medium examples. For more complex models there might rise the need to reduce the number of mutants. Here we refer to the survey by Jia and Harmann [103], that describes multiple ways of reducing mutants for mutation testing, which can in general also be applied to model-based mutation testing.
- **Limited Modeling Elements.** As a last, more technical restriction, we would like to mention that not all of UPPAALs features are supported by our tool. While the first prototype could process data variables and parameters, the reimplementation (that was published under the name MoMuT::TA) does not take them into account. It only works for classic timed automata with inputs and outputs. While data variables could be added to the test-case generation with comparatively little effort, the bounded determinization approach presented in the following chapter would still be restricted to classical timed automata, as data variables might introduce non-deterministic updates. It is noteworthy, however, that the bounded model-checking approach well with data variables and we could not observe any performance loss when using data variables. A further restriction of the test-case generation approach is that none of the tools can handle urgent or committed locations, though the same behavior can be expressed without using these elements. Finally, we also do not support urgent transitions, probabilities and the c-like functions that can be used in the declaration part of UPPAAL models.

4 Bounded Determinization

Parts of this chapter are based on our publications at FORMATS 2015 [123] and TASE 2016 [16].

In the last chapter we showed a mutation-based test-case generation methodology, which is restricted to deterministic and fully-observable specifications. To overcome this restriction, the current chapter will investigate the removal of silent transitions and the determinization of timed automata.

The design of modern embedded systems often involves the integration of interacting components I_1 and I_2 that realize some requested behavior. Figure 4.1 illustrates two components I_1 and I_2 that realize the integrated system I. In early stages of the design, I_1 and I_2 are high-level and partial models that facilitate considerable implementation freedom to the designer. In practice, this freedom is reflected in the non-deterministic choices that are intended to be resolved during subsequent design refinement steps. In addition, the composition of two components involves their synchronization on some shared actions. Typically, the actions over which the two components interact are *hidden* and become unobservable to the user. It follows that the overall specification $I = I_1 \mid\mid I_2$ can be a *non-deterministic partially observable* model.

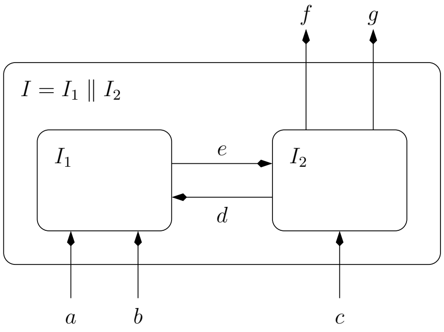


Figure 4.1: Embedded components I_1 and I_2, and their composition I.

The passage from a high-level model towards an implementation consists of an iteration of *refinement* steps. In every refinement step, the designer must ensure that the more concrete model I' restricts the output behavior of I (e.g. by resolving some of the non-deterministic choices in I) and does not add new outputs which I does not admit. It follows that the designer has to check, using for instance model checking or model-based testing techniques, whether I' *refines* I. When considering non-deterministic partially observable models, the notion of refinement is often based on trace or alternating trace inclusion. In practice, checking whether I' refines I often requires the *determinization* of I. Additionally, for many problems such as model-based testing, observability, implementability and language inclusion checking, it is desirable and in certain cases necessary to work with the deterministic model.

In contrast to the classical automata theory, determinism and observability play a crucial role in the theory of timed automata. In particular, deterministic TA (DET(TA)) are strictly less expressive than the fully observable non-deterministic NON-DET(TA) [29, 162, 86], whereas the latter are strictly less expressive than timed automata with silent transitions (TA$_\epsilon$) [50]. This strict hierarchy of TA with respect to determinism and observability has an important direct consequence - NON-DET(TA) are not determinizable in general and silent transitions cannot be removed in general without changing the automaton's language.

In this chapter, we propose a procedure for *bounded determinization* of NON-DET(TA$_\epsilon$). Given an arbitrary *strongly responsive*[2] NON-DET(TA$_\epsilon$) A and a bound k, our algorithm computes a deterministic

[2] In model-based testing, strong responsiveness is the requirement that there are no loops consisting of only silent transitions, otherwise the tester cannot distinguish between deadlocks and livelocks.

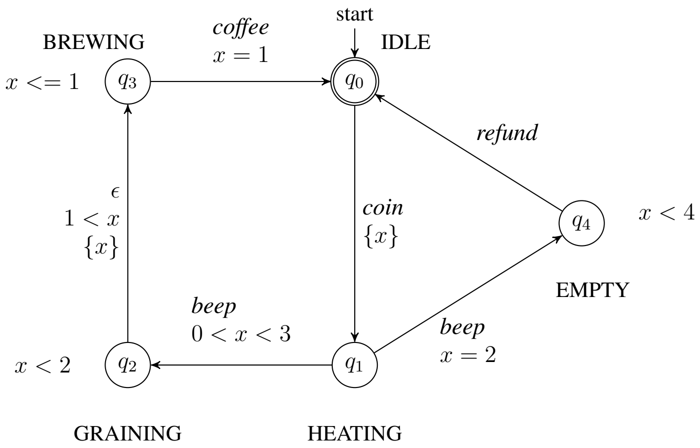


Figure 4.2: Running example for the silent transition removal and the determinization approach.

automaton DET(A) in the form of a timed tree, such that every timed trace consisting of at most k observable actions is a trace in A if and only if it is a trace in DET(A). It provides the basis for effectively applying the model-based mutation testing approach presented in Chapter 3 to non-deterministic specifications, but also aids other applications, like bounded refinement checking and other test-case generation procedures.

The proposed algorithms are performed in three steps:

1. we unfold the original automaton into a finite tree and rename the clocks in a way that only needs one clock reset per transition,
2. we remove the silent transitions from the tree,
3. we determinize it.

Our determinization procedure results in a TA description which includes diagonal [58] and disjunctive constraints. Although non-standard, this representation is practical and optimized for the bounded setting – it avoids costly transformation of the TA into its standard form and exploits efficient heuristics in SMT solvers that can directly deal with this type of constraints. In addition, our focus on bounded determinization enables us to consider models, such as TA with loops containing both observable and silent transitions with reset, that could not be determinized otherwise. We implemented the functionality as an extension to the tool MoMuT::TA.

Running example. The different steps of the algorithms will be illustrated on a running example of a coffe-machine shown in Figure 4.2. After inserting a *coin*, the system heats up for zero to three seconds, followed by a *beep*-tone indicating its readiness. Alternatively, if there is no coffee or water left, the *beep* might occur after exactly two seconds, indicating that the *refunding* process has started and the coin will be returned within four seconds. Heating up and graining the coffee together may only take up to two seconds, indicated by the invariant of the graining location. Then the brewing process starts and finally the machine releases the *coffee* after one second of brewing. There is no observable signal indicating the transition from graining to brewing, thus this transition is silent.

The remainder of the chapter is structured as follows: first, we illustrate the first step of our procedure, the bounded-unfolding of the automaton and the renaming of clocks (Section 4.1). This is followed by the second step, the removal of silent transitions (Section 4.2) and the final step, our determinization approach (Section 4.3). Then, in Section 4.4, we will discuss determinization of networks of timed automata and in Section 4.5 we will discuss how the individual steps of the silent transition removal and determinization can be performed simultaneously in an on-the-fly algorithm. In Section 4.6 we discuss the limitations of the approach and then in Section 4.7 we discuss our implementation and evaluate the prototype on some examples.

4.1 Preprocessing

4.1.1 k-Bounded Unfolding of Timed Automata

Algorithm 2 Unfolding of a TA$_\epsilon$.

```
Input: strongly responsive NON-DET(TA$_\epsilon$) $A$, bound $k$
Output: $U_k(A)$, a tree of depth $K$ and observable depth $k$
 1: CREATE $\hat{q}^t$ in $U_k(A)$                                ▷ root of the tree
 2: $P \leftarrow \{(\hat{q}, \hat{q}^t, 0)\}$                        ▷ set of locations to process
 3: while $P \neq \emptyset$ do
 4:     PICK $(q, q^t, i) \in P$
 5:     $P \leftarrow P \backslash (q, q^t, i)$
 6:     if $i < k$ then
 7:         for each $\tau = (q, \alpha, g, \rho, q') \in trans(q)$ do
 8:             CREATE $q'^t$ in $U_k(A)$                         ▷ add target location to tree
 9:             $I(q'^t) \leftarrow I(q')$                          ▷ copy invariant to new location
10:             CREATE $\tau_t = (q^t, \alpha, g, \rho, q'^t)$ in $U_k(A)$    ▷ add transition to tree
11:             if $\alpha = \epsilon$ then           ▷ update locations that need to be processed
12:                 $P \leftarrow P \cup \{(q', q'^t, i)\}$
13:             else
14:                 $P \leftarrow P \cup \{(q', q'^t, i + 1)\}$
15:             end if
16:         end for
17:     end if
18: end while
```

Given a NON-DET(TA$_\epsilon$) A which is strongly responsive, its k-prefix language $\mathcal{L}_k(A) \subseteq \mathcal{L}(A)$ is the set of observable timed traces induced by all accepting runs of A which are of observable length bounded by k. That is,

$$\mathcal{L}_k(A) = \{w \in \mathcal{L}(A) \mid |w| \leq k\}, \tag{4.1}$$

where $|w|$ depicts the observable length of the trace w. By unfolding A and cutting it at observable level k, the resulting TA, $U_k(A)$, satisfies

$$\mathcal{L}(U_k(A)) = \mathcal{L}_k(A). \tag{4.2}$$

Algorithm 2 illustrates how to create $U_k(A)$. First, we create an initial location in the tree and create a state tuple consisting of the initial location of A, the initial location of $U_k(A)$ and a 0 for the current depth (Line 2). Then, for each state tuple in P, we search for all currently enabled transitons. For each of these transitions, we create a new location in $U_k(A)$, and copy the transition to $U_k(A)$ with the new location as target location. Then, we create a new state tuple, were in case of an observable transition, i is increased, otherwise it stays unchanged. $U_k(A)$ is in the form of a finite tree, where each path that starts at the root ends after at most k observable transitions, and we may also further cut A by requiring that all leaves are accepting locations. Note, that if we reach in $U_k(A)$ a copy of an accepting location q

of A by a silent transition then it will not be marked as an accepting location (but another copy might be marked as an accepting location if reached by an observable transition).

Figure 4.3(a) shows the unfolding of the coffee-machine up to observable depth three. The left branch is longer than the right, as it contains a silent transition.

4.1.2 Renaming the Clocks

Every unfolded timed automaton can be expressed by an equivalent timed automaton that resets at most one clock per transition. This known normal form [39] crucially simplifies the next stages of our algorithm, where we do not need to bother with multiple clock resets in one transition. Additionally, we benefit from the fact, that each transition of same depth resets the same clock, which simplifies the determinization. The basic idea is to substitute the clocks from the original automaton by new clocks, where multiple old clocks reset at the same transition are replaced by one single new clock, as they measure the same time until they are reset again. The substitution of the clocks works straight forward: at each path from the root, at the i-th observable transition, a new clock x_i is introduced and reset, and if this transition is followed by $l > 0$ silent transitions then new clocks $x_{i,0}, \dots, x_{i,l-1}$ are introduced and reset. A clock x that occurs in a guard is substituted by the new clock that was introduced in the transition where the last reset of x happened, or by x_0 if it was never reset. Let τ_i and τ_j be two transitions on the same path in the original automaton at observable depth i, j, s.t. $i < j$. Furthermore, a clock x appearing in the guard of τ_j, is reset before in τ_i, but is not reset on any transition in between τ_i and τ_j. Then, x_i is introduced and reset at τ_i and the original clock variable x is substituted by x_i in the guard of τ_j. Clocks in invariants are updated the same way as guards. Figure 4.3(b) illustrates the clock renaming applied to the coffee machine. In the guards of the two *beep*-transitions starting at q_1, x is replaced by x_1, since the last reset of x in the original automaton was at depth one, while in the *coffee*-transition from q_3 it is replaced by $x_{2,0}$, as x was reset in the first silent transition after depth two.

The concrete algorithm used for renaming of the clocks is presented in pseudo-code in Algorithm 3. The original clocks are $x_0, \dots, x_{n-1}$. Each new clock has either one index (l_1) in case the transition in

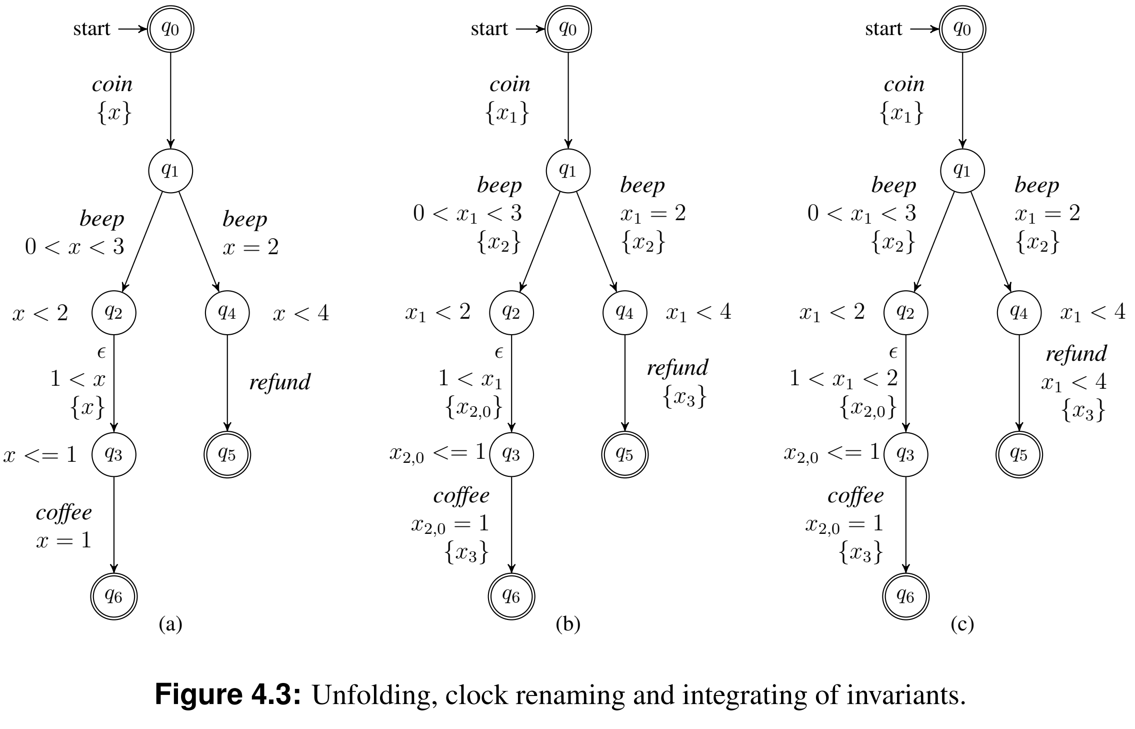


Figure 4.3: Unfolding, clock renaming and integrating of invariants.

Algorithm 3 Renaming the Clocks.

Input: $A \in$ NON-DET($\text{TA}_\epsilon)_K$, a tree of depth K and observable depth k, clocks $\mathcal{C}$, $|\mathcal{C}| = n$

Output: $A \in$ NON-DET($\text{TA}_\epsilon)_K$, clocks $\mathcal{C}'$, single clock reset per transition, same clock reset at same (observable, silent) level

```
$l_1 \leftarrow 0$                                          ▷ observable (primary) level
$l_2 \leftarrow -1$                                         ▷ silent (secondary) level
for $i \leftarrow 0, .., n-1$ do
    $X[i] \leftarrow x_0$                                   ▷ $x_0$ is reset at the initial location
end for
RenameClocks($q_0, X, l_1, l_2$)

procedure RenameClocks($q, X, l_1, l_2$)
    $I(q) \leftarrow I(q)[x_i \leftarrow X[i]]$             ▷ renaming the clocks in the invariant $I(q)$
    for each $\tau = (q, \alpha, g, \rho, q') \in trans(q)$ do
        for $i \leftarrow 0, .., n-1$ do
            $g \leftarrow g[x_i \backslash X[i]]$           ▷ renaming the clocks in the guard $g$
        end for
        if $\alpha = \epsilon$ then                         ▷ silent transition
            $l_2 \leftarrow l_2 + 1$
            $x \leftarrow x_{l_1,l_2}$                      ▷ the new reset clock in case of a silent trans.
            $\mathcal{C}' \leftarrow \mathcal{C}' \cup \{x_{l_1,l_2}\}$
        else
            $l_1 \leftarrow l_1 + 1$
            $l_2 \leftarrow -1$
            $x \leftarrow x_{l_1}$                          ▷ the new reset clock in case of an observable trans.
            $\mathcal{C}' \leftarrow \mathcal{C}' \cup \{x_{l_1}\}$
        end if
        for $i \leftarrow 0, .., n-1$ do
            if $x_i \in \rho$ then
                $X[i] \leftarrow x$                         ▷ updating the clock substitution list
            end if
        end for
        $\rho \leftarrow \{x\}$                             ▷ updating the reset clocks of $\tau$
        if $l_1 < k$ then
            RenameClocks($q', X, l_1, l_2$)                 ▷ recursive call with the target location
        end if
    end for
end procedure
```

which it is reset is observable, or two indices (l_1, l_2) in case of a silent transition. After the removal of the silent transitions stage we will be left with clocks with a single index and the same clock reset for the same level of the tree. The vector $X[0..n-1]$ holds the clock substitution list: $X[i]$ refers to the new clock that substitutes the original clock x_i. The set of transitions with source location q is denoted by $trans(q)$.

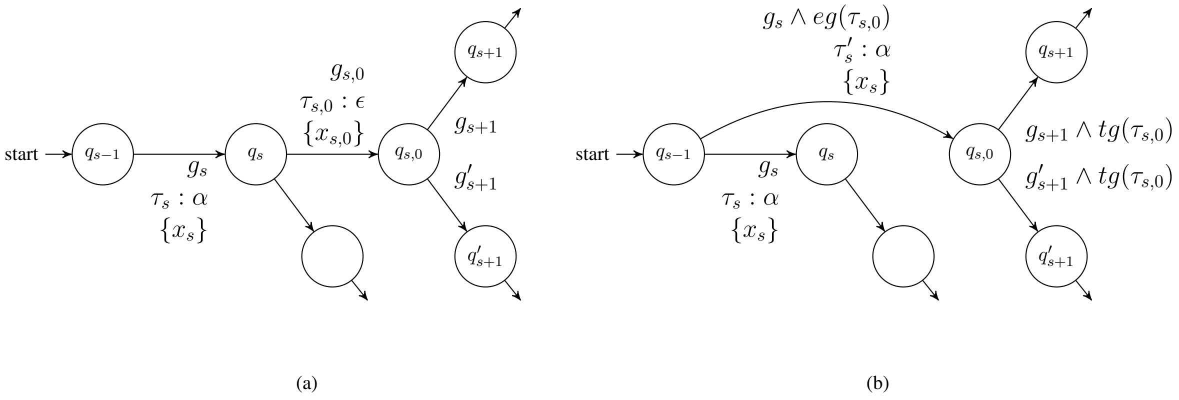


Figure 4.4: An example automaton with a silent transition (a) and the corresponding fully observable automaton with a bypass transition (b).

4.1.3 Integrating Invariants into Guards

In this last preprocessing step, we integrate all location invariants into the guards of all outgoing transitions. When the guards are updated and synchronized during the silent transition removal, this ensures that the constraints of the invariants are also taken into account in that step. They are added to the guards via conjunction. The invariants are still kept unchanged in the automaton. Figure 4.3(c) shows the integrating of the invariants in the running example, where e.g., the guard of the silent transition was augmented by $x_1 < 2$. Note that the guard of the *coffee* transition was not modified, as $x_{2,0} = 1$ is already more strict than $x_{2,0} \leq 1$. As no transition can be traversed if the invariant of its source location is not enabled, adding the invariants to the guards does not change the language of the automaton.

4.2 Silent Transition Removal

In this section we give an algorithm that removes the silent transitions from the NON-DET(TA_ϵ) A, which is in the form of a finite tree with renamed clocks. Thus, at each level i there will be a single clock x_i reset on all transitions of that level. That is, one clock x_0 is reset at the initial location. Then, at each path from the root, at the i-th observable transition the new clock x_i is reset, and if this transition is followed by $l > 0$ silent transitions then new clocks $x_{i,0}, \ldots, x_{i,l-1}$ are reset. In the next observable transition the clock x_{i+1} is reset, and so on. All these clocks are not reset again. After removing the silent transitions, the new TA, $O(A)$, will have the same one clock reset at all transitions of same level. Algorithm 4 shows the workflow of the silent transition removal and Figure 4.4 illustrates the general idea.

We remove the silent transitions one at a time, where at each iteration we remove the first occurrence of a silent transition on some path from the root, until no silent transitions are left (e.g. we can pick a path and remove all its silent transitions one-by-one, then move to another path, and so on). So, let $\tau_{s,0}$ be such a first silent transition found by Line 2 of the algorithm, leading from location q_s to location $q_{s,0}$ with guard $g_{s,0}$ and reset of clock $x_{s,0}$. Let q_s be reached from location q_{s-1} with an observable transition τ_s and with guard g_s. The case where q_s is the initial location is simpler, as it does not require building a bypass transition. In order to remove the silent transition $\tau_{s,0}$ after forming a transition that bypasses it, several steps are carried out, that will be explained in detail in the following subsections. First, we set an auxiliary lower bound on the clock that is reset on the silent transition by updating the guard (Line 3). Then, we create the bypass transition using an *enabling guard* $eg(\tau_{s,0})$ which represents the upper bound until when the silent transition $\tau_{s,0}$ is enabled (Line 4). In Line 5 we construct a *taken guard* $tg(\tau_{s,0})$

Algorithm 4 Removing the Silent Transitions.

Input: $A \in \text{NON-DET}(\text{TA}_\epsilon)_k$ in the form of a tree of observable depth k with renamed clocks

Output: $O(A) \in \text{NON-DET}(\text{TA})_k$, such that $\mathcal{L}(O(A)) = \mathcal{L}(A)$

```
1: while there are silent transitions do
2:     FIND first (from root) silent transition $\tau_{s,0}$ from $q_s$ to $q_{s,0}$
3:     SET lower bound to the silent transition
4:     CREATE bypass transition with enabling guard
5:     AUGMENT transitions from $q_{s,0}$ with taken guard
6:     UPDATE guards on paths from $q_{s,0}$
7:     UPDATE location invariants
8:     REMOVE $\tau_{s,0}$
9: end while
```

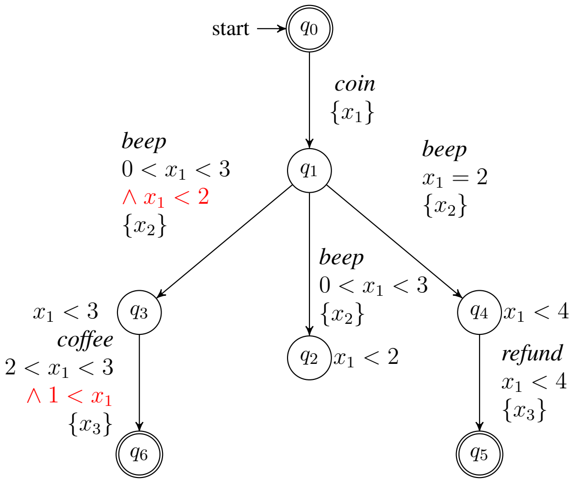


Figure 4.5: Fully observable non-deterministic TA.

that ensures that the transitions from $q_{s,0}$ come after the necessary delay that is enforced by the silent transition. The taken guard is added to all transitions leaving $q_{s,0}$. Next, in Line 6 we update all future guards referring to the deleted clock $x_{s,0}$. Finally, we update all invariants referring to the deleted clock (Line 7) and we remove the silent transition $\tau_{s,0}$.

4.2.1 Setting a Lower Bound to the Silent Transition.

We set a lower bound to the silent transition by augmenting the guard $g_{s,0}$ of $\tau_{s,0}$ to be $g'_{s,0} = g_{s,0} \wedge (0 \leq x_s)$, where x_s is the clock that is reset on the transition τ_s that precedes the silent transition. This additional constraint per definition always evaluates to true, but it is used in the next step to compute the unary constraints of the enabling guard. The guard of the silent transition in Figure 4.3 (c) after setting the lower bound is $1 < x_1 < 2 \wedge 0 \leq x_2$.

4.2.2 Creating a Bypass with the Enabling Guard.

The enabling guard $eg(\tau_{s,0})$ guarantees that each clock's constraint that was part of the silent transition is satisfied at some non-negative delay and that these constraints are satisfied simultaneously, thus at some point during the bypass transition the silent transition would have been enabled as well. We describe here how the enabling guards are defined for strict inequalities, as shown in the upper part of Table 4.1. The other cases are dealt similarly, as seen in the table, and the constraint $x_i = n_i$ is treated as $n_i \leq x_i \leq n_i$. For every pair of a lower bound constraint $m_i < x_i$ and an upper bound constraint $x_j < n_j$, where n and m are constants, $i \neq j$ and $x_i, x_j \neq x_s$ (x_s is the clock that is reset at τ_s), that appear in $g'_{s,0}$ we form the enabling guard binary constraint $x_j - x_i < n_j - m_i$ as shown in the first line of Table 4.1.

The next two lines consider constraints that involve the clock x_s, where x_s will be removed as it is the clock that will be reset on the bypass and is considered of value 0. Note, that for each upper bound constraint $x_j < n_j$ we use the lower bound constraint $0 \leq x_s$ that was added in the previous step of the algorithm to compute the enabling guard unary constraint $x_j < n_j$, which guarantees that at the time of the bypass x_j does not pass its upper bound constraint of the silent transition. An example of such a unary constraint is marked in red in the transition from q_1 to q_3 in Figure 4.5. The silent transition in the original automaton could not have been enabled if x_1 had already been greater than two after the *beep*-transition, thus the bypass can also only be enabled while x_1 is smaller than two. The running example does not contain any binary constraints.

Note that the upper bound constraints of the invariant of q_s were integrated into the guard of the silent transition. Thus, they are also integrated into the enabling guard. Consequently, the bypass will satisfy the constraints of the location invariant. The invariant itself remains unchanged, as it remains active for all traces that pass through q_s, but do not take the silent transition.

To create the bypass, we split the paths through q_s in the original automaton A into two. Those that do not take the silent transition $\tau_{s,0}$ continue as before from q_{s-1} to q_s and then to some location different from $q_{s,0}$. The paths that went through $\tau_{s,0}$ are directed from q_{s-1} to $q_{s,0}$ and then continue as before. The bypass τ'_s from q_{s-1} to $q_{s,0}$ has the same observable actions as those of τ_s, the same new clock reset x_s, and the guard g'_s which is the guard g_s of τ_s augmented with the *enabling guard* $eg(\tau_{s,0})$ (see Figure 4.4). Figure 4.5 shows the removal of the silent transition illustrated on the coffee-machine. The transition from q_1 to q_3 is the bypass and the transition from q_1 to q_2 is the original transition. Since the silent transition was the only transition leaving q_2, q_2 does not contain any outgoing transitions anymore, once the bypass is generated.

4.2.3 Augmenting the Taken Guard.

For each transition from $q_{s,0}$ to q_{s+1} we augment its guard g_{s+1} by forming $g'_{s+1} = g_{s+1} \wedge tg(\tau_{s,0})$ (see Figure 4.4), where $tg(\tau_{s,0})$ is the *taken guard.* $tg(\tau_{s,0})$ is composed of a single constraint: $0 \leq x_{s,0}$, where $x_{s,0}$ is the clock that is reset at the silent transition $\tau_{s,0}$. In the next stage of the algorithm of updating the future guards it will be transformed into the conjunction of the lower bound constraints $m_i < x_i$ or $m_i \leq x_i$ that appear in $g'_{s,0}$. These constraints make sure that we spend enough time at $q_{s,0}$ before moving to the next locations, as if we had taken the silent transition. The constraints are also used for synchronization of the future guards in the next step. In Figure 4.5, the red-marked part of the guard from transition q_3 to q_6 shows the taken guard that has already been updated from $0 \leq x_{2,0}$ to $1 < x_1$.

Silent Trans. Constraints	Clock Reset	Enabling Guard Constraint
$(m_i < x_i) \wedge (x_j < n_j)$	x_s	$x_j - x_i < n_j - m_i$
$(m_s < x_s) \wedge (x_j < n_j)$	x_s	$x_j < n_j - m_s$
$(m_i < x_i) \wedge (x_s < n_s)$	x_s	$m_i - n_s < x_i$
$(m_i \leq x_i) \wedge (x_j < n_j)$	x_s	$x_j - x_i < n_j - m_i$
$(m_i < x_i) \wedge (x_j \leq n_j)$	x_s	$x_j - x_i < n_j - m_i$
$(m_i \leq x_i) \wedge (x_j \leq n_j)$	x_s	$x_j - x_i \leq n_j - m_i$
$(m_i = x_i) \wedge (x_j = n_j)$	x_s	$x_j - x_i = n_j - m_i$

Table 4.1: Enabling guard constraints.

Silent Trans. Constr.	Future Constr.	Replaced Constr.
$m_i < x_i$, $\{x_{s,0}\}$	$m_{s+j} < x_{s,0}$ or $m_{s+j} \leq x_{s,0}$	$m_i + m_{s+j} < x_i$
$m_i \leq x_i$, $\{x_{s,0}\}$	$m_{s+j} < x_{s,0}$	$m_i + m_{s+j} < x_i$
$m_i \leq x_i$, $\{x_{s,0}\}$	$m_{s+j} \leq x_{s,0}$	$m_i + m_{s+j} \leq x_i$
$x_i < n_i$, $\{x_{s,0}\}$	$x_{s,0} < n_{s+j}$ or $x_{s,0} \leq n_{s+j}$	$x_i < n_i + n_{s+j}$
$x_i \leq n_i$, $\{x_{s,0}\}$	$x_{s,0} < n_{s+j}$	$x_i < n_i + n_{s+j}$
$x_i \leq n_i$, $\{x_{s,0}\}$	$x_{s,0} \leq n_{s+j}$	$x_i \leq n_i + n_{s+j}$
$x_i = n_i$, $\{x_{s,0}\}$	$x_{s,0} \sim n_{s+j}$	$x_i \sim n_i + n_{s+j}$

Table 4.2: Update rules for future guards after removing the silent transitions.

4.2.4 Updating the Future Guards.

The removal of the silent transition $\tau_{s,0}$ enforces updating of the guards in the paths that start at $q_{s,0}$ and that refer to the clock $x_{s,0}$ that is reset on the silent transition. The constraints that refer to the other clocks can remain as they are.

The simplest case is when the silent transition guard $g'_{s,0}$ contains an exact constraint $x_i = n_i$, because then any future constraint of the form $x_{s,0} \sim l$ where $\sim$ is one of the signs $=$, $<$, $>$, $\leq$ or $\geq$, can be replaced by $x_i \sim n_i + l$. In that case we know the exact time of the silent transition, and all other constraints may be ignored. So, let us assume that the silent transition does not contain an exact constraint. The rules for updating the future guards are summarized in Table 4.2. Note, that an equality constraint $x_{s,0} = n_{s+j}$ in a future guard may be treated as $n_{s+j} \leq x_{s,0} \leq n_{s+j}$.

Let $g_{s+1}, \ldots, g_{s+p}$ be the ordered list of guards of consecutive transitions $\tau_{s+1}, \ldots, \tau_{s+p}$ along a path that starts at $q_{s,0}$. Suppose, for simplicity, that each g_{s+j} contains constraints that refer to $x_{s,0}$ (if not - we can ignore g_{s+j}). Then, if g_{s+j} contains the constraint $m_{s+j} < x_{s,0}$, it is replaced by the conjunction of constraints of the form $m_i + m_{s+j} < x_i$, for each constraint $m_i < x_i$ that appears in $g'_{s,0}$. Similarly, for upper bound constraints. In Figure 4.5, one future guard was updated in the transition from q_3 to q_6: The original guard of this transition was $x_{2,0} = 1$ (where $x_{2,0}$ was reset on the silent transition) and the guard of the silent transition was $1 < x_1 < 2$. Thus, according to the update rules, the updated future guard is $2 < x_1 < 3$ (written in black), conjuncted with the taken guard (marked in red).

These rules ensure that each future constraint on the clock $x_{s,0}$ separately conforms to and does not deviate from the possible time range of the silent transition. Yet, we need to satisfy a second condition: that along each path that starts at $q_{s,0}$ these future occurrences of $x_{s,0}$ are synchronized. This is achieved by augmenting the future guards with constraints of the form that appear in Table 4.3. No transition in our running example needs synchronization, hence we use a different example: the upper automaton in Figure 4.6 shows one silent transition followed by two observable transitions. Using only the previous update rules when removing the silent transition, the first observable transition might occur between

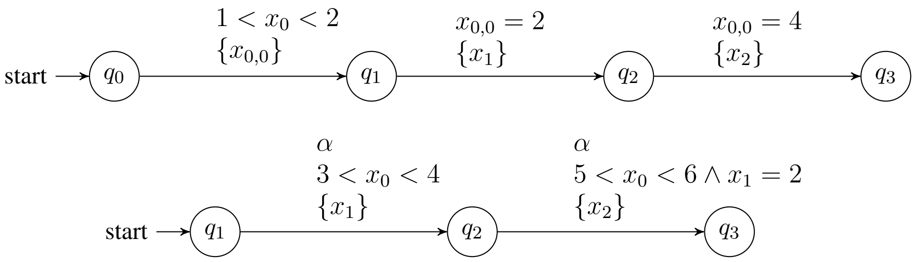


Figure 4.6: Guard synchronization.

Constr. of g_{s+j}	**Constr. of** g_{s+i}**,** $\{x_{s+i}\}$**,** $i < j$	**Sync. Constr. of** g_{s+j}
$m_{s+j} < x_{s,0}$	$x_{s,0} < n_{s+i}$ or $x_{s,0} \leq n_{s+i}$	$m_{s+j} - n_{s+i} < x_{s+i}$
$m_{s+j} \leq x_{s,0}$	$x_{s,0} < n_{s+i}$	$m_{s+j} - n_{s+i} < x_{s+i}$
$m_{s+j} \leq x_{s,0}$	$x_{s,0} \leq n_{s+i}$	$m_{s+j} - n_{s+i} \leq x_{s+i}$
$x_{s,0} < n_{s+j}$	$m_{s+i} < x_{s,0}$ or $k_i \leq x_{s,0}$	$x_{s+i} < n_{s+j} - m_{s+i}$
$x_{s,0} \leq n_{s+j}$	$m_{s+i} < x_{s,0}$	$x_{s+i} < n_{s+j} - m_{s+i}$
$x_{s,0} \leq n_{s+j}$	$m_{s+i} \leq x_{s,0}$	$x_{s+i} \leq n_{s+j} - m_{s+i}$
$x_{s,0} = n_{s+j}$	$x_{s,0} = n_{s+i}$	$x_{s+i} = n_{s+j} - n_{s+i}$

Table 4.3: Synchronization constraints for future guards after removing silent transitions.

three and four seconds, and the second one between five and six seconds. If the first transition occurs after three seconds and the second one after six, this would not conform to the original automaton which required exactly two seconds between them. Thus, applying the last synchronization rule of Table 4.3, the constraint $x_1 = 4 - 2$ is conjuncted to the second guard. The lower automaton in Figure 4.6 illustrates the synchronization. Note, we do not need a bypass transition here, since the silent transition starts in the initial state.

4.2.5 Updating of Location Invariants.

In this last step, we need to update all location invariants that refer to $x_{s,0}$, the clock that was reset by the silent transition. Consider an invariant li in any location behind the silent transition (that might be $q_{s,0}$, the target location of the silent transition, or any following location). All constraints in li that do not involve $x_{s,0}$ can remain unchanged, and we can assume that there is only one constraint for each clock, as a stronger upper bound subsumes a weaker one. Thus we only consider the constraint $x_{s,0} < m$. This constraint is updated the same way as a future guard that refers to $x_{s,0}$. The upper bounds of the silent transition ($x_i < n_i$) and the upper bound of the invariant ($x_{s,0} < m$) are combined to the new upper bounds $x_i < n_i + m$. Note that it is not necessary to synchronize updated invariants among each other, as they only contain lower bounds and thus do not interfere with each other. The synchronization of the future guards with the invariants already happened in the last step, as the constraints of the invariants were added to the guards of the following transitions. Figure 4.5 shows that the invariant of location q_3 was updated from $x_{2,0} \leq 1$ to $x_1 \leq 3$, according to the upper bound $x_1 < 2$ of the silent transition.

4.2.6 Removing the Silent Transition.

Finally, we can safely remove the silent transition $\tau_{s,0}$ from q_s to $q_{s,0}$ after forming the bypass from q_{s-1} to $q_{s,0}$ with the necessary modifications to the transition guards.

Theorem 4.1 (Silent Transitions Removal)
$\mathcal{L}(O(A)) = \mathcal{L}(A)$.

A proof of the theorem can be found in Section 4.8.

4.3 Determinization

Existing determinization algorithms, as e.g. applied by Wang et al. [173], create the powerset of all transitions to be determinized, and build one transition for each subset in the powerset. We propose an alternative approach, that reduces the amount of locations and transitions in the deterministic automata,

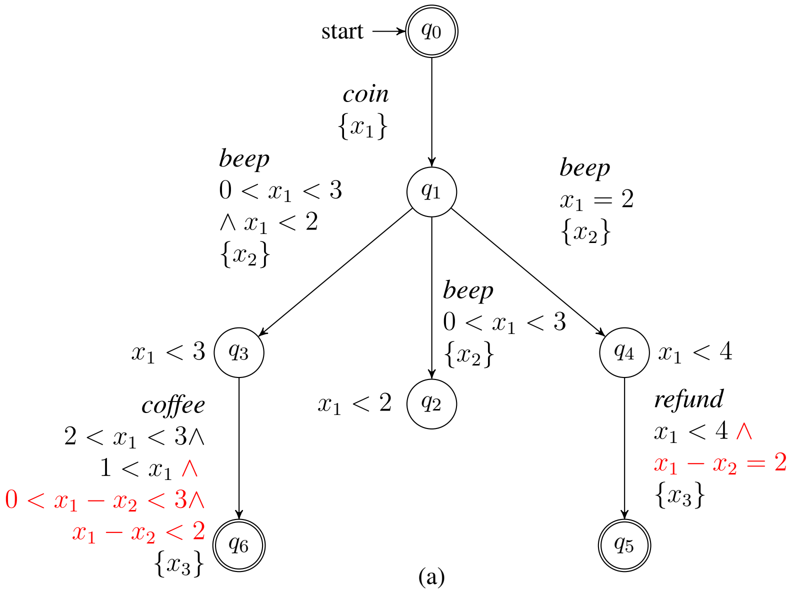


Figure 4.7: Modified guards added to future transition.

.

by shifting some complexity towards the guards. Our motivation is the use of SMT solvers for model-based test-case generation from timed automata. The larger guards can be directly converted into SMT-LIB formulas, and thus should not pose a problem. The produced automata contain disjunctions, both in the guards and the invariants. While this does not conform to the standard definition of timed automata, in the context of SMT-solving the disjcuntions can efficiently be processed and do not hinder test-case generation.

The approach works under the following prerequisites: after the removal of the silent transitions the timed automaton A is in the form of a tree of depth k. At each level i the same new clock x_i is reset on

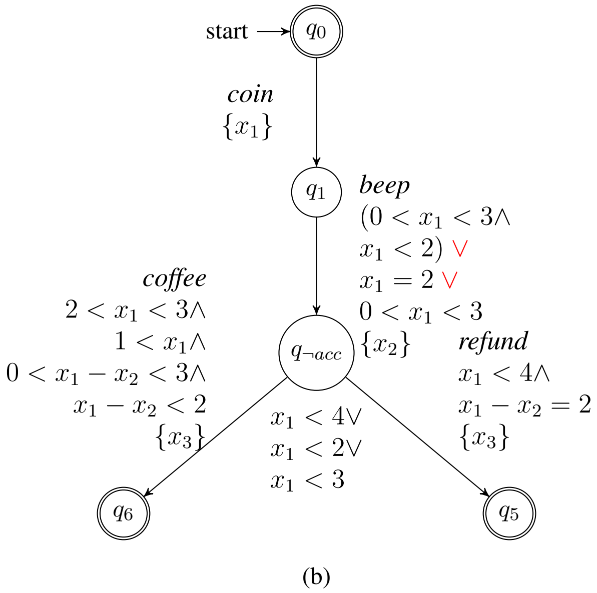


Figure 4.8: Determinization via disjunction.

.

Algorithm 5 Guard-Oriented Determinization.

```
Input: $A \in \textsc{non-det}(\mathrm{TA})_k$ in the form of a tree of depth $k$ with renamed clocks
Output: $D(A) \in \mathrm{TA}_k$, such that $\mathcal{L}(D(A)) = \mathcal{L}(A)$
 1: $P \leftarrow \{(\hat{q}, 0)\}$
 2: while $P \neq \emptyset$ do
 3:     Pick $(q_i, i) \in P$; $P \leftarrow P \setminus \{(q_i, i)\}$                      ▷ For each tuple in $P$
 4:     for each $\alpha \in \Sigma$ do
 5:         if $\exists\ \tau_1(q_i, \alpha, g_1, \{x_{i+1}\}, q_1) \neq \tau_2(q_i, \alpha, g_2, \{x_{i+1}\}, q_2)$ then   ▷ Non-deterministic trans.
 6:             $g_{acc} \leftarrow$ false; $g_{\neg acc} \leftarrow$ false
 7:             Add new locations $q_{acc}$, $q_{\neg acc}$
 8:             Merge transitions$(A, q_i, g_{acc}, g_{\neg acc}, q_{acc}, q_{\neg acc})$
 9:             for each transition $\tau_i(q_i, \alpha, g_{i+1}, \{x_{i+1}\}, q_{i+1})$ do
10:                 $P \leftarrow P \cup \{(q_{i+1}, i+1)\}$                              ▷ Update $P$
11:             end for
12:         end if
13:     end for
14: end while
```

each of the transitions of that level. This is the only clock reset on this level, and no clock is ever reset again.

The basic idea behind the determinization algorithm is to merge all transitions of the same source location and the same action via disjunction, and to push the decision which of them was actually taken to the following transitions. The postponed decision which transition was actually taken can be solved later on by forming diagonal constraints that are invariants of the time progress, and are added via conjunction to immediately following transitions. Note that the distinction between accepting and non-accepting locations increases complexity slightly: the determinization of transitions leading to accepting locations and transitions leading to non-accepting locations cannot be done exclusively by disjunction of their guards. We therefore need to add an accepting and a non-accepting location to the deterministic tree, and merge all transitions leading to non-accepting locations and all transitions leading to accepting locations separately. To ensure determinism for these transitions, we conjunct the negated guard of the accepting transition to the guard of the non-accepting transition. Additionally, the location invariants of merged target locations are combined via disjunction.

A pseudo-code description is given in Algorithm 5 and Algorithm 6. Algorithm 5 contains the outline of the algorithm: the determinization is done in several steps applied to every location q with multiple outgoing transitions with the same action (Line 5), starting at the initial location (Line 1). First, we add an accepting and a non-accepting location q_{acc}, $q_{\neg acc}$ replacing the target locations of the multiple α transitions (Line 7). Then we perform the merging of these transitions according to Algorithm 6: let q_i be such a location with multiple α transitions (Line 1). For each τ_i in the α transitions with guard g from q_i to q_{i+1}, let g' be the result of subtracting the clock x_{i+1} that is reset on τ_i from all clocks that appear in g (Lines 2-5). Next, g' is conjuncted to the guards of each transition τ_{i+1} that follows τ_i and the source location of τ_{i+1} is set to either q_{acc} or $q_{\neg acc}$, depending on whether q_{i+1} is accepting or not. Transitions leaving $q_{\neg acc}$ are additionally copied to q_{acc}, in case the guards of α transitions overlap. (Lines 7,8). Note that g' evaluates to true in every branch below τ_i if τ_i was enabled, thus the conjunction does not change the language of the automaton. Figure 4.7 illustrates the conjunction of the modified guards on our running example, marked in red. Note that the determinization did not involve any accepting locations, thus there was no splitting into q_{acc} and $q_{\neg acc}$. Next, all the α-transitions from q leading to accepting locations are merged into a transition leading to q_{acc} (Line 19) and all others into a transition leading to $q_{\neg acc}$ (Line 20), by disjuncting their guards (Lines 12,15). The guard of the transition leading

Algorithm 6 Merging of Transitions.

```
Input: $A, q_i, g_{acc}, g_{\neg acc}, q_{acc}, q_{\neg acc}$
Output: $A'$, with merged transitions
 1: for each transition $\tau_i(q_i, \alpha, g_{i+1}, \{x_{i+1}\}, q_{i+1})$ do      ▷ For each $\alpha$ transition
 2:     $g' \leftarrow g_{i+1}$
 3:     for each clock $x_j$ in $g_{i+1}$ do
 4:         $g' \leftarrow g'[x_j \backslash x_j - x_{i+1}]$      ▷ Build diagonal constraints
 5:     end for
 6:     for each transition $\tau_{i+1}(q_{i+1}, \beta, g_{i+2}, \{x_{i+2}\}, q_{i+2})$ do      ▷ Move transitions leaving $q_{i+1}$
 7:         ADD $\tau_{acc}(q_{acc}, \beta, (g_{i+2} \wedge g'), \{x_{i+2}\}, q_{i+2})$      ▷ to $q_{acc}$
 8:         ADD $\tau_{\neg acc}(q_{\neg acc}, \beta, (g_{i+2} \wedge g'), \{x_{i+2}\}, q_{i+2})$      ▷ and $q_{\neg acc}$
 9:         REMOVE $\tau_{i+1}$
10:     end for
11:     if $accepting(q_{i+1})$ then
12:         $g_{acc} \leftarrow g_{acc} \vee g_{i+1}; li(q_{acc}) = li(q_{acc}) \vee li(q_{i+1})$      ▷ Merge guards and invariants
13:     end if
14:     if $\neg accepting(q_{i+1})$ then
15:         $g_{\neg acc} \leftarrow g_{\neg acc} \vee g_{i+1}; li(q_{\neg acc}) = li(q_{\neg acc}) \vee li(q_{i+1})$      ▷ Merge guards and invariants
16:     end if
17:     REMOVE $\tau_i$ and $q_{i+1}$
18: end for
19: ADD transition $\tau_{acc}(q_i, \alpha, g_{acc}, \{x_{i+1}\}, q_{acc})$
20: ADD transition $\tau_{\neg acc}(q_i, \alpha, (g_{\neg acc} \wedge \neg g_{acc}), \{x_{i+1}\}, q_{\neg acc})$      ▷ Add merged transitions
```

to $q_{\neg acc}$ is conjuncted to the negation of the other guard, to ensure determinism (Line 20). Additionally, in Lines 12 and 15, the location invariants of the different target locations of the merged transitions are combined via disjunction. Finally, all merged τ_i and their target locations can be removed (Line 17). Figure 4.8 shows the determinized coffee-machine. The location $q_{\neg acc}$ contains a location invariant that is a disjunction of the invariants from locations q_2, q_3 and q_4 of the non-deterministic tree.

Theorem 4.2 (Determinization)
The determinization algorithm constructs a deterministic timed automaton $D(A)$ such that $\mathcal{L}(D(A)) = \mathcal{L}(A)$.

The proof of the theorem can be found in Section 4.8.

4.4 Networks of Timed Automata

Up to this point, we only considered single timed automata, both for the test-case generation and the determinization and silent transition removal. In this section, we will first discuss how to build the product of timed automata with inputs and outputs in a network and then show how to include the product building directly into the bounded unfolding. The integration of the two steps enables us to avoid explicitly building the product, which may become very big and complex.

4.4.1 Product of Timed Automata with Inputs and Outputs

To define *composition* between TAIO$_\epsilon$ we extend the definition of parallel products between I/O Automata introduced by David et al. [71], by including silent transitions and hiding communication transitions after the product. The hiding of the communication transitions provides a black-box view on the system. Our definition differs from the one by Krichen and Tripakis [117], as our TAIO$_\epsilon$ do not contain deadlines. Furthermore, our definition is less restrictive, as we do not require disjoint sets of external input and output actions, respectively. Thus our product may produce non-deterministic automata, which will be determinized later on. Note that, in contrast to our definition below, these previous notions of composition are defined for exactly two automata: in the first definition [71] two synchronizing transitions form a new output transition, that can be used for composition with another automaton. Thus two different automata can synchronize on the same output transition of a third automaton simultaneously. In the second definition [117] communication transitions are hidden by the product. Thus, a third automaton could not synchronize with the product anymore, making the product definition not associative. Our product definition is associative for the closed set of TAIO$_\epsilon$ it is applied to, but looses this property if an additional TAIO$_\epsilon$ is composed afterwards, due to the hiding.

Definition 4.1
The *parallel product* of an NTA N composed of n TAIO$_\epsilon$ A_i is the non-deterministic TAIO$_\epsilon$ $A = (Q, (\hat{q}_1, \dots, \hat{q}_n), \Sigma_e^I \cup \Sigma_e^O \cup \{\epsilon\}, \mathcal{C}_1 \cup \dots \cup \mathcal{C}_n, I, \Delta)$ where $Q \subseteq (Q_1 \times \dots \times Q_n)$, $I : (Q_1 \times \dots \times Q_n) \rightarrow$ location invariants s.t. $I(q_1, \dots, q_n) = I_1(q_1) \wedge \dots \wedge I_n(q_n)$ and Q and the set of transitions Δ are defined by the following inductive rules, where $(\dots, q_i, \dots) \xrightarrow{a,g,\rho} (\dots, q_i', \dots)$ means that only the state of the i-th automaton changes:

Start

$$\frac{}{(\hat{q}_1, \dots, \hat{q}_n) \in Q}$$

External

$$\frac{q_i \xrightarrow{a,g,\rho} q_i' \qquad a \in \Sigma_e^O \cup \Sigma_e^I \qquad (\dots, q_i, \dots) \in Q}{(\dots, q_i, \dots) \xrightarrow{a,g,\rho} (\dots, q_i', \dots) \qquad (\dots, q_i', \dots) \in Q}$$

Internal

$$\frac{q_i \xrightarrow{a,g_1,\rho_1} q_i' \qquad q_j \xrightarrow{a,g_2,\rho_2} q_j' \qquad a \in \Sigma_i^O \qquad a \in \Sigma_j^I \qquad (\dots, q_i, \dots, q_j, \dots) \in Q}{(\dots, q_i, \dots, q_j, \dots) \xrightarrow{\epsilon,\, g_1 \wedge g_2,\, \rho_1 \cup \rho_2} (\dots, q_i', \dots, q_j', \dots) \qquad (\dots, q_i', \dots, q_j', \dots) \in Q}$$

Silent

$$\frac{q_i \xrightarrow{\epsilon,\, g,\, \rho} q_i' \qquad (\dots, q_i, \dots) \in Q}{(\dots, q_i, \dots) \xrightarrow{\epsilon,\, g,\, \rho} (\dots, q_i', \dots) \qquad (\dots, q_i', \dots) \in Q}$$

4.4.2 k-Bounded Unfolding of Networks of Timed Automata

We will now define how to integrate building the product of a network of timed automata directly into the bounded unfolding, as described in Section 4.1.1. Given an NTA N composed of n TAIO$_\epsilon$ A_i, that does not contain any loops consisting purely of silent and internal transitions, Algorithm 7 shows how to unfold it into a tree-like single automaton of observable depth k.

The main structure of the algorithm is very similar to that of the unfolding algorithm for single timed automata (Algorithm 2). However, there are a few adaptions: The set P, which stores the tuples that need yet to be processed, now contains the current locations of all automata in the network, and is initialized with all initial locations (Line 2).

When a tuple is processed, we iterate through all automata in the network, and investigate all transitions that are enabled in that automata. For each transition, we distinguish three cases: the first two cases, external transitions (Lines 9-13) and silent transitions (Lines 14-18), are processed quite similarly to transitions in the unfolding of single automata. For each of them we create a new location and transition in the tree and store a new state tuple in P. The current locations of all automata but the one currently processed stay the same. In the case of external transitions we increase i, otherwise it stays unchanged.

The third case, internal transitions (Lines 19-25), includes building a new silent transition from two communicating transitions. For each enabled output transition of one automaton, we build a new silent transition for each enabled input transition of another automaton that has the same label. The combined transition resets the union of the clock resets of the individual transitions, and combines their guards via conjunction. We then update the current location of both automata, when building the new tuple that is stored in P.

In all three cases, the invariant of the new location in the tree is the conjunction of the invariants of all current locations in the network.

Algorithm 7 Unfolding of an NTA.

Input: strongly responsive NTA N composed of n TAIO$_\epsilon$ A_i, bound k

Output: $U_k(A)$, a tree of depth K and observable depth k

```
CREATE $\hat{q}_t$ in $U_k(A)$                                        ▷ root of the tree
$P \leftarrow \{((\hat{q}_0 \cdots \hat{q}_n), \hat{q}_t, 0)\}$                          ▷ set of locations to process
while $P \neq \emptyset$ do
    PICK $((q_0 \cdots q_n), q^t, i) \in P$
    $P \leftarrow P \backslash \{((q_0 \cdots q_n), q^t, i)\}$
    if $i < k$ then
        for each $A_j \in N$ do                         ▷ for each automaton $A_j$ in the network
            for each $\tau = (q_j, \alpha, g, \rho, q'_j) \in trans(q_j)$ do      ▷ for each enabled transition in $A_j$
                if $\alpha \in \Sigma_e^O \cup \Sigma_e^I$ then                        ▷ process observable transitions
                    CREATE $q'^t$ in $U_k(A)$                            ▷ add target location to tree
                    $I(q'^t) \leftarrow I(q'_j) \wedge \bigwedge_{l \neq j}(I(q_l))$              ▷ copy invariants to new location
                    CREATE $\tau_t = (q^t, \alpha, g, \rho, q'^t)$ in $U_k(A)$            ▷ add transition to tree
                    $P \leftarrow P \cup \{((q_0 \cdots q'_j \cdots q_n), q'^t, i + 1)\}$
                else if $\alpha = \epsilon$ then                                  ▷ process silent transitions
                    CREATE $q'^t$ in $U_k(A)$                            ▷ add target location to tree
                    $I(q'^t) \leftarrow I(q'_j) \wedge \bigwedge_{l \neq j}(I(q_l))$              ▷ copy invariants to new location
                    CREATE $\tau_t = (q^t, \alpha, g, \rho, q'^t)$ in $U_k(A)$        ▷ add silent transition to tree
                    $P \leftarrow P \cup \{((q_0 \cdots q'_j \cdots q_n), q'^t, i)\}$
                else if $\alpha \in \Sigma_j^O$ then                            ▷ process internal transitions
                    for each $A_h \in N$ s.t. $h \neq j \wedge \alpha \in \Sigma_h^I$ do     ▷ for each $A_h$ where $\alpha$ is an input
                        for each $(q_h, \alpha, g_h, \rho_h, q'_h) \in trans(q_h)$ do   ▷ for each enabled $\alpha$ transition
                            CREATE $q'^t$ in $U_k(A)$                    ▷ add target location to tree
                            $I(q'^t) \leftarrow I(q'_j) \wedge I(q'_h) \wedge \bigwedge_{l \neq j, l \neq h}(I(q_l))$        ▷ copy invariants
                            CREATE $\tau_t = (q^t, \epsilon, g \wedge g_h, \rho \cup \rho_h, q'^t)$ in $U_k(A)$   ▷ add silent transition
                            $P \leftarrow P \cup \{((q_0 \cdots q'_j \cdots q'_h \cdots q_n), q'^t, i)\}$
                        end for
                    end for
                end if
            end for
        end for
    end if
end while
```

Algorithm 8 Deterministic bounded unfolding of NTAN.

```
Input: NTA$N$, max. depth $k$
Output: TAIO$A$ such that $\mathcal{L}_k(N) = \mathcal{L}_k(A)$
$\hat{q}_s \leftarrow (\hat{q}_1 \ldots \hat{q}_n)$; $\hat{q}_t \leftarrow$ new location
$locations_t \leftarrow \{\hat{q}_t\}$; $transition_t \leftarrow \emptyset$
$\mathcal{CR} : Clock \to Clock = (x \mapsto x_0) \forall x \in \bigcup \mathcal{C}_i$        ▷ map for clock renaming
$\mathcal{GU} : Clock \to Constraint = \emptyset$        ▷ constraints for future guard update
$\mathcal{GS} : Clock \to (Clock, Constraint) = \emptyset$        ▷ constraints for future guard synch.
$\mathcal{NG} : \{Constraints\} = \emptyset$        ▷ constraints to resolve non-determinism
$P_1 \leftarrow \{(\hat{q}_t, \hat{q}_m, 0, 0, \mathcal{CR}, \mathcal{GU}, \mathcal{GS}, \mathcal{NG})\}$;
for $i \in 1..k$ do
    $P_{i+1} \leftarrow \emptyset$
    while $P_i \neq \emptyset$ do
        PICK $(q_t, q_i, i, e, \mathcal{CR}, \mathcal{GU}, \mathcal{GS}, \mathcal{NG}) \in P_i$
        $P_i \leftarrow P_i \cup$ $\epsilon$-CLOSURE$(q_t, q_i, i, e, \mathcal{CR}, \mathcal{GU}, \mathcal{GS}, \mathcal{NG})$
        $P_i \leftarrow P_i \cup$ INT$(q_t, q_i, i, e, \mathcal{CR}, \mathcal{GU}, \mathcal{GS}, \mathcal{NG})$
        $P_{i+1} \leftarrow P_{i+1} \cup$ EXT $(q_t, q_i, i, e, \mathcal{CR}, \mathcal{GU}, \mathcal{GS}, \mathcal{NG})$
    end while
end for
```

4.5 On-The-Fly Algorithm

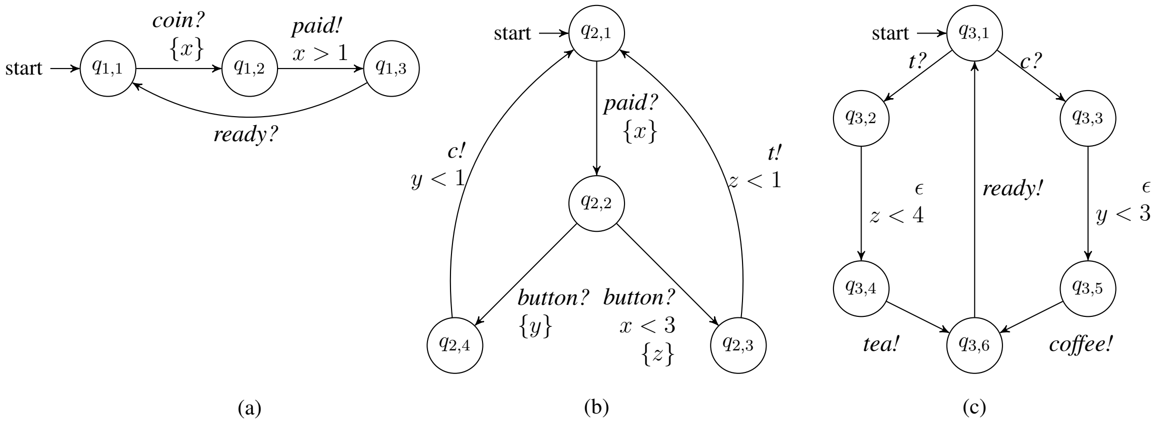
Applying the different steps of the determinization and silent transition removal sequentially requires traversing the complete state-space of the unfolding several times. Once for the unfolding, once for the clock-renaming, multiple times for the silent transition removal (where we need to traverse the subtree beneath each removed silent transition to update the future guards) and once for the determinization.

As the unfolding naturally leads to a huge state-space, this sequential approach is rather ineffective. Thus, in this section we propose an on-the-fly algorithm that performs at each level of unfolding the following tasks: building the product[3], hiding the communication[3], renaming the clocks, removing silent transitions, and determinizing.

The main feature of the algorithm is the fact that it stores all constraints needed for updating future transitions, and only needs to explore the state-space exactly once. Additionally, as this algorithm has access to the original model during the determinization (where the sequential approach only had the unfolded version), it can merge locations that were the same in the original model and store the same constraints for the clock updating. Thus, if that optimization is enabled, the resulting automata are directed acyclic graphs instead of trees, and contain a far lower amount of locations.

The unfolding will be performed on the NTA$N = (\mathcal{A}, \Sigma_e^I, \Sigma_e^O, \Sigma_i)$, where $\mathcal{A} = \{A_1..A_n\}$ and each $A_i = (Q_i, \hat{q}_i, \Sigma_{\epsilon,i}, \mathcal{C}_i, I_i, \Delta_i)$. The approach is restricted to NON-DET(TA) that do not contain loops of purely silent and communication actions. The NTA may as well only contain one automaton, so that using the same algorithm for determinization of single automata is possible. The core part of the algorithm processes *state tuples* of the form $(q_t, q_i, i, e, \mathcal{CR}, \mathcal{GU}, \mathcal{GS}, \mathcal{NG})$, where q_t is the location in the unfolded tree, $q_i = (q_{(i,1)}, \ldots q_{(i,n)})$ are the current locations in the NTA, i is the current depth and e is the number of silent transitions already processed at the end of the current trace. $\mathcal{CR}$ (*clock renaming*) is a partial function mapping clocks to clocks, used for renaming of the clocks. The renaming ensures the property that in each transition only one clock is reset and that all transitions of same depth reset the same clock. $\mathcal{GU}$ (*guard update*) is a partial function mapping clocks to constraints, storing the constraints

[3] These steps are only applied in case of NTA, the algorithm also works for single automata.



(a) (b) (c)

Figure 4.9: An NTA depicting a coffee machine with three components. One for payment, one for product selection and one for providing the drinks.

of removed silent transitions. Keeping these constraints enables the updating of future guards that refer to a clock that was reset on a deleted silent transition, to keep the timing constraints valid. $\mathcal{GS}$ (*guard synchronization*) is a partial function mapping clocks to tuples of clocks and constraints, storing the clock resets and guards of all transitions that were updated using $\mathcal{GU}$, so these transitions can additionally be synchronized among each other, to keep the delays between them valid. $\mathcal{NG}$ (*next guard update*) stores diagonal constraints that are built during the determinization and need to be attached to the guards of all transitions directly beneath the determinized transitions. These state tuples are stored in k sets P_i.

The main algorithm is shown in Algorithm 8, and the three sub-algorithms it uses are shown in detail in Algorithm 10–12. P_1 is initialized with a single state tuple, containing the initial location of the tree, q_t = $\hat{q}_t$, the initial states of $\mathcal{A}$, $q_s = (\hat{q}_1, \ldots \hat{q}_n)$, 0 as current depth, 0 as number of prior silent transitions, $\mathcal{CR}$ mapping every clock to x_0 and $\mathcal{GU}, \mathcal{GS}$ and $\mathcal{NG}$ being empty (Lines 1–7).

The outermost loop in the algorithm increases the index i from 1 to k. At each step, P_i contains the state tuples of the current depth, that need yet to be processed. P_{i+1} contains the state tuple of the next observable depth. Then, for every element in P_i three stages are executed. In Line 12, we process all silent transitions that can be taken in any of the automata A_i in their current location. This does not produce any new locations or transitions in the tree, but creates new state tuples to be stored in P_i. Details of this stage can be found in Subsection 4.5.1 and Algorithm 10. The second stage is performed in Line 13, where communicating transitions between the automata are synchronized. The synchronization via an action $a \in \Sigma_i$ is only performed if one of the current locations of an automaton A_i in the NTA provides a as an output and another automaton A_j accepts it as an input in its current location. As the synchronized transitions are hidden, they are afterwards processed in the same way as the silent transitions. Details can be found in Subsection 4.5.2 and Algorithm 11. The third and last step handles external transitions. This might introduce new locations and transitions in the tree, if one of the old automata can perform a transition with a label not yet available in the current location of the tree. If the tree already contains a transition with the same label and source location, the transition is modified, by adding its guard and the guard of the new transition via disjunction. The state tuples produced in this step are stored in P_{i+1}, to be processed in the next depth iteration. This is explained in detail in Subsection 4.5.3 and Algorithm 12.

Example 4.1. Figure 4.10 shows the first steps of applying the algorithm to the coffee machine example illustrated in Figure 4.9, that was already introduced in the preliminaries (Section 2.1.5). P_1 is initialized with the state tuple $(q_t = q_1, q_i = (q_{1,1}, q_{2,1}, q_{3,1}), i = 0, e = 0, \mathcal{CR} = \{x \mapsto x_0, y \mapsto x_0, z \mapsto x_0\}, \mathcal{GU} = \emptyset, \mathcal{GS} = \emptyset, \mathcal{NG} = \emptyset)$. Starting at the initial state of all three automata, there are no silent or internal transitions enabled. Thus, we start with processing external transitions, where *coin?* is the

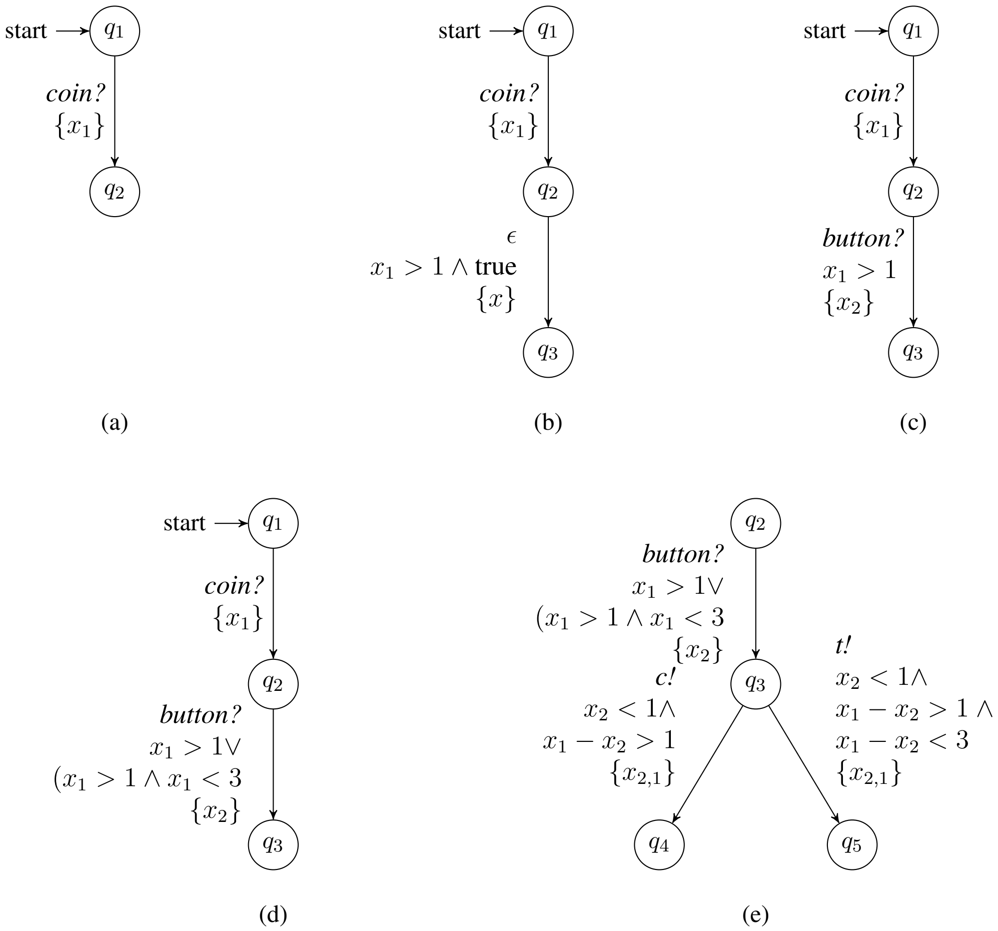


Figure 4.10: Different steps of building the determinized unfolding.

only enabled transition. As there is no transition leaving q_1 with the label *coin?* yet, we build a new location q_2 and a transition leading there, labeled by *coin?* (see Figure 4.10(a)). This step adds the state tuple $(q'_t = q_2, q'_i = (\mathbf{q_{1,2}}, q_{2,1}, q_{3,1}), i' = \mathbf{1}, e' = 0, \mathcal{CR}' = \{\mathbf{x \mapsto x_1}, y \mapsto x_0, z \mapsto x_0\}, \mathcal{GU}' = \emptyset, \mathcal{GS}' = \emptyset, \mathcal{NG}' = \emptyset)$ to P_2. $\mathcal{CR}'$ is updated, because the processed transition resets the clock x, that is now mapped to x_1. $\mathcal{GU}'$,$\mathcal{GS}'$ and $\mathcal{NG}'$ are still empty, as no silent transition or non-determinism was processed.

P_2 only contains the tuple that was just produced. In the current NTA locations of that tuple, only one internal transition, *paid*, is enabled, that is an output of the first automaton, with guard $x > 1$ and an input of the second, where the clock x is reset. First, we build a combined transition, where both guards are combined via conjunction (conjunction with true will be omitted), the clock resets are combined by union and the label is turned into ϵ. This combined transition is shown in Figure 4.10 (b). This is just an illustration, actually the transition will be removed again when it is processed. Then, we substitute the clocks according to $\mathcal{CR}$, changing the guard from $x > 1$ to $x_1 > 1$ and the clock reset to $\{x_{1,0}\}$ (This transition is the first silent transition removed in the current trace on Depth 1, thus we use $x_{1,0}$. A second silent transition in the same trace would then reset $x_{1,1}$). $\mathcal{GU}$, $\mathcal{GS}$ and $\mathcal{NG}$ are empty and thus not applied. We update $\mathcal{CR}$ to $\mathcal{CR}'$ by adding $x \mapsto x_{1,0}$. Thus, in future guards that refer to x, x will be substituted to $x_{1,0}$. We update $\mathcal{GU}$ to $\mathcal{GU}'$ by adding $x_{1,0} \mapsto x_1 > 1$. Thus, if a future guard g refers to $x_{1,0}$ we build a constraint that refers to x_1 instead of $x_{1,0}$, by combining g and the constraint $x_1 > 1$ from $\mathcal{GU}'$. Finally, we add all lower bounds occurring in the guard ($x_1 > 1$) to $\mathcal{NG}'$. They will be attached to the following transitions and ensure they do not occur to soon. Together that gives the state tuple $(q_t' = q_2, q_i' = (\mathbf{q_{1,3}}, \mathbf{q_{2,2}}, q_{3,1}), i' = 1, e' = \mathbf{1}, \mathcal{CR}' = \{\mathbf{x \mapsto x_{1,0}}, y \mapsto x_0, z \mapsto x_0\}, \mathcal{GU}' = \{\mathbf{x_{1,0} \mapsto x_1 > 1}\}, \mathcal{GS}' = \emptyset, \mathcal{NG}' = \{\mathbf{x_1 > 1}\})$, added to P_2.

For the new tuple, there are two observable transitions with the same label (*button?*) enabled. First we process the one without guard, leading to $q_{2,4}$. There exists no transition with label *button?* leaving q_2 in the tree, thus we add the new location q_3 and a transition leading there. Then we add $\mathcal{NG}$ $(x_1 > 1)$ to the guard (see Figure 4.10 (c)) and change the clock reset from $\{y\}$ to $\{x_2\}$. We update $\mathcal{GS}'$ by adding $x_{1,0} \mapsto (x_2, x_1 > 1)$, where x_2 is the clock that is reset on the current transition, and $x_1 > 1$ is its guard. If a later transition also refers to $x_{1,0}$, it can be synchronized with the current transition, so the delay between them is valid. After adding the transition we update $\mathcal{NG}'$ by building the diagonal constraint $x_1 - x_2 > 1$, subtracting x_2 (the clock reset on the current transition) from all clocks in the current guard $x_1 > 1$. $\mathcal{NG}'$ will be conjuncted to all following transitions, so these later transitions are only enabled, if x_1 was greater than 1 at the time of the *button?* transition. The tuple $(q_t' = \mathbf{q_3}, q_i' = (q_{1,3}, \mathbf{q_{2,4}}, q_{3,1}), i' = \mathbf{2}, e' = \mathbf{0}, \mathcal{CR}' = \{x \mapsto x_{1,0}, \mathbf{y \mapsto x_2}, z \mapsto x_0\}, \mathcal{GU}' = \{\mathbf{x_{1,0} \mapsto x_1 > 1}\}, \mathcal{GS}' = \{\mathbf{x_{1,0} \mapsto (x_2, x_1 > 1)}\}, \mathcal{NG}' = \{\mathbf{x_1 - x_2 > 1}\})$ is stored in P_3. For the second *button?* transition there already exists a transition with the same label in the tree, so the two transitions are determinized after the second transition is updated. The determinization works via disjunction, as illustrated in Figure 4.10 (d). The produced state tuple is $(q_t = q_3, q_i = (q_{1,3}, \mathbf{q_{2,3}}, q_{3,1}), i = \mathbf{2}, e = \mathbf{0}, \mathcal{CR}' = \{x \mapsto x_{1,0}, y \mapsto x_0, \mathbf{z \mapsto x_2}\}, \mathcal{GU}' = \{x_{1,0} \mapsto x_1 > 1\}, \mathcal{GS}' = \{\mathbf{x_{1,0} \mapsto (x_2, x_1 > 1 \wedge x_1 < 3)}\}, \mathcal{NG}' = \{\mathbf{x_1 - x_2 > 1 \wedge x_1 - x_2 < 3}\})$. The determinization via disjunction produces a weaker guard, that is enabled if any of the original guards were enabled. The diagonal constraint stored in $\mathcal{NG}'$ in the two state tuples are later attached to the following transitions (see Figure 4.10 (e)), to ensure that $t!$ is only enabled, if $x_1 < 3$ was enabled in the combined *button?* transition. The combined transition is also enabled for any value $x_1 \geq 3$, but then only $c!$ can be taken later on. Note that the c and t transitions in Figure 4.10 (e) are internal, and will be removed when processed. □

The next three subsections give details on processing silent, internal and external transitions. Six operations are equally called in the first step of every of these stages. They are shown in Algorithm 9 and explained below:

Add invariants to guard: First, we build a conjunction of the invariants of all currently selected locations in the NTA. This conjunction is added to the guard of the processed transition, to make sure that traversing the transition is valid in all automata. The invariants are also added to the determinized tree, after they are updated the same way as guards in the next steps. For simplicity we will not mention them separately. Thus no trace in the tree can violate any of the original invariants. We only consider traces ending in discrete steps, neglecting time progress in the leaves of the tree.

Apply $\mathcal{NG}$: $\mathcal{NG}$ stores constraints (like the enabling guard) that need to be attached to all transitions

Algorithm 9 Updates needed for all transitions.

```
Input: $(g_i, \mathcal{CR}, \mathcal{GU}, \mathcal{GS}, \mathcal{NG})$
Output: $\{(g', \mathcal{CR}', \mathcal{GS}')\}$
*Add invariants to guard*
*Apply* $\mathcal{NG}$
$g' \leftarrow$ *substitute according to* $\mathcal{CR}(g)$
$\forall x$ s.t. $x \in \mathcal{GS} \wedge x \in g' : g' \leftarrow g' \wedge$ *apply* $\mathcal{GS}(\mathcal{GS}, x, g')$
$g' \leftarrow$ *substitute according to* $\mathcal{GU}(g')$
$\mathcal{CR}' \leftarrow$ *update* $\mathcal{CR}$
$\forall x$ s.t. $x \in \mathcal{GU} \wedge x \in g' : \mathcal{GS}' \leftarrow$ *update* $\mathcal{GS}(g', x)$
```

directly following the transition that produced it. Applying $\mathcal{NG}$ is done by conjuncting all its constraints to the guard.

Update $\mathcal{CR}$: Every clock x that is reset on the currently processed transition will be added to $\mathcal{CR}$. If the current transition is a silent transition (or one produced by synchronizing two communication transitions) x will in the future be substituted by the clock $x_{i,e}$, otherwise it will be substituted by x_i. Thus, if for example the two clocks x and y are reset on an observable transition on Depth 5, both $x \leftarrow x_5$ and $y \leftarrow x_5$ will be added to $\mathcal{CR}$, where previous entries for x and y are deleted. Note that all clocks $x_{i,e}$ will be removed in later steps, leaving only the clocks x_1–x_k in the final automaton.

Substitute according to $\mathcal{CR}$: All clocks that appear in the guard of a processed transition t_i need to be substituted according to the function $\mathcal{CR}$. In the first steps, this means that all clocks are substituted by x_0, later on, as $\mathcal{CR}$ will be updated, the substitution may change. This corresponds to the clock renaming step of Section 4.1.2.

Substitute according to $\mathcal{GU}$: If a clock $x_{j,e}$ is already defined in $\mathcal{GU}$ this means it was reset on a silent transition that was processed earlier in the trace leading to the current transition. If such a clock appears in the guard g of the currently processed transition t_i, the guard constraints referencing $x_{j,e}$ will be updated using the constraints stored in $\mathcal{GU}(x_{j,e})$. This update removes the lower bound $x_{j,e} > m$ from g and for each lower bound $x > n$ in $\mathcal{GU}(x_{j,e})$ it adds the constraint $x > n + m$ to g via conjunction. Upper bounds are treated equally, and $=$ is treated as $\leq$ and $\geq$. Note that the built constraint contains strict inequality as long as one of the original constraints contained strict inequality. This step corresponds to the updating of future guards from Section 4.2.

Update $\mathcal{GS}$: $\mathcal{GS}$ needs to be updated every time we process a transition that refers to a clock that was reset on an internal or silent transition. It is used (see next item), to synchronize all transitions that refer to that clock. For these synchronization constraints, $\mathcal{GS}$ needs to store the constraints referring to the clock and the clock that is reset on the current transition.

Apply $\mathcal{GS}$: $\mathcal{GS}$ needs to be applied to every transition referring in its guard to a clock $x_{j,e}$ that was reset by a silent transition (despite the first one, where it is only updated) to synchronize these transitions. The synchronization is done by building the following constraint: for each upper bound $x_{j,e} < n$ stored in $\mathcal{GS}(x_{j,e})$, linked to the clock y (the clock reset on the transition from which we extracted $x_{j,e} < n$) and each lower bound $x_{j,e} > m$ in the current guard, we build the constraint $y > m - n$, ensuring that the time that passed between the two transitions is not larger than the difference between the lower bound of the first transition and the upper bound of the later one. Assume the constraint $3 \leq x$ of one transition, and the constraint $x \leq 5$ at a later transition. Then at most 2 $(5 - 3)$ seconds may pass between the two transitions. Similar constraints are built for the lower bounds in $\mathcal{GS}(x_{j,e})$ and the upper bounds in the guard. More details on the constraints can be found in Section 4.2, in the step for synchronizing future guards.

4.5.1 ϵ-Closure

Given a tuple $(q_t, q_i, i, e, \mathcal{CR}, \mathcal{GU}, \mathcal{GS}, \mathcal{NG})$, Algorithm 10 depicts the workflow of processing all silent transitions that can be taken in any location $q_{(i,j)} \in q_i$ (Lines 2–3). Most of this workflow correlates to the step of removing silent transitions presented in Section 4.2. The main difference is that the previous approach had to traverse the whole subtree beneath each silent transition, to update it. Contrary to that, we only keep all constraints we need for the updating stored, so we can update the future transitions at the time we process them during unfolding.

Update transition: In the first step, the guard of the transition is updated according to the constraints of previous silent transitions or non-determinism. Details are given in Algorithm 9. This step also computes $\mathcal{CR}'$ and $\mathcal{GS}'$.

Update $\mathcal{GU}$: Every time a silent transition is processed, we store all upper and lower bound constraints occurring in its guard, mapped to the clock that was reset on the silent transition. The constraints are later used to update guards that refer to that clock to ensure that their timing constraints are still valid.

Algorithm 10 ϵ-Closure.

```
Input: $(q_t, q_i, i, e, \mathcal{CR}, \mathcal{GU}, \mathcal{GS}, \mathcal{NG})$
Output: $\{(q_t, q_i', i, e+1, \mathcal{CR}', \mathcal{GU}', \mathcal{GS}', \mathcal{NG}')\}$
 1: $P \leftarrow \emptyset$
 2: for each location $q_{(i,j)} \in q_i$ do
 3:     for each transition $t_i = (q_{(i,j)}, \epsilon, g_i, \rho_i, q_{(l,j)})$ do
 4:         $(g', \mathcal{CR}', \mathcal{GS}') \leftarrow$ update transition$(t_i)$
 5:         $\mathcal{GU}' \leftarrow$ update $\mathcal{GU}$
 6:         $en \leftarrow$ calculate enabling guard$(g')$
 7:         $\mathcal{NG}' \leftarrow$ update enabling guard$(en) \cup \{$taken guard$\}$
 8:         $P \leftarrow P \cup \{(q_t, q_i[q_{(i,j)} \backslash q_{(l,j)}], i, e+1, \mathcal{CR}', \mathcal{GU}', \mathcal{GS}', \mathcal{NG}')\}$
 9:     end for
10: end for
11: return $P$
```

Calculate Enabling Guard: When removing a silent transition, we can express the upper bound until when it is enabled via diagonal constraints, by pairwise comparing the upper and lower limits of all clocks appearing in the guard, to identify the time interval when all clocks are enabled at once. The exact details on building the constraints can be found in Section 4.2.

Update Enabling Guard: In the approach for single timed automata we built a bypass transition for every silent transition. That was a copy of the transition preceding the silent one, augmented by the enabling guard and leading to the target location of the silent transition. During determinization, this transitions was removed again and only the enabling guard, attached to the following transitions, remained. Now we can avoid building the transition, by immediately attaching the enabling guard to the following transitions: we create the constraint $\mathcal{NG}'$ by subtracting in the enabling guard the clock that was reset on the preceding transition from all clocks appearing in the enabling guard. Let $x_2 < 5$ be an enabling guard, and x_3 be the clock reset on the preceding transition, then $x_2 - x_3 < 5$ is the processed enabling guard. This builds a time invariant that, in all following transitions, can decide whether the enabling guard was enabled at the time of the preceding transition or not. $\mathcal{NG}'$ will be attached to all transitions that can leave the new target states in the NTA. We also add the *taken guard* to $\mathcal{NG}'$, that is a conjunction of all lower bounds in the guard. This ensures that no following transition can be taken earlier than allowed by the silent transition.

Storing the next tuple: Finally, the next state tuple is created and stored in P, the set of tuples returned to the main algorithm (Line 8). This tuple reflects the next combination of locations in the NTA that needs to be processed. As no externally observable transition was processed, it is still linked to the same location in the tree and i is not increased. In the set of current locations in the NTA, only the location $q_{i,j}$ in the j-th automaton (the automaton containing the processed silent transition) changes to $q_{l,j}$. Finally, e is increased by one.

4.5.2 Internal Transitions

Internal transitions are processed similarly to silent transitions. The main difference is that internal transitions are only processed pairwise, and only if an action is enabled in one automaton as an input and in another as an output. Algorithm 11 shows the applied steps. Lines 3-9 collect all input and output transitions with label $a \in \Sigma_i$ that are enabled in the different automata. Then, for each combination of an input and an output transition (Line 10) we build a combined transition t (Line 11), where the action label is ϵ, the guard is the conjunction of both guards and the set of reset clocks is the union of the two set ρ_i and ρ_o. This transition is then processed equally to silent transitions, until Line 16, where we update the current locations of both automata that were involved in the synchronization.

Algorithm 11 Processing internal transitions in the NTA.

```
Input: $(q_t, q_i, i, e, \mathcal{CR}, \mathcal{GU}, \mathcal{GS}, \mathcal{NG})$
Output: $\{(q_t, q_i', i, e+1, \mathcal{CR}', \mathcal{GU}', \mathcal{GS}', \mathcal{NG}')\}$
$P \leftarrow \emptyset$
for each $a \in \Sigma_i$ do
    $I \leftarrow \emptyset; O \leftarrow \emptyset$
    for each location $q_{(i,j)} \in q_i$ do
        for each transition $t_i = (q_{(i,j)}, a, g_i, \rho_i, q_{(h,j)})$ do
            if $isOutputTransition(t_i)$ then $O \leftarrow O \cup \{(t_i, j)\}$
            else $I \leftarrow I \cup \{(t_i, j)\}$ end if
        end for
    end for
    for each $(t_o, k) \in O$ and each $(t_i, l) \in I$ do
        $t = (q, \epsilon, g_o \wedge g_i, \rho_o \cup \rho_i, q') \leftarrow$ combine trans.$(t_i, t_o)$
        $(g', \mathcal{CR}', \mathcal{GS}') \leftarrow$update transition$(t)$
        $\mathcal{GU}' \leftarrow$ update $\mathcal{GU}$
        $en \leftarrow$ calculate enabling guard$(g')$
        $\mathcal{NG}' \leftarrow$ update enabling guard$(en) \cup$ taken guard
        $P \leftarrow P \cup (q_t, q_i[q_{(i,k)} \backslash q_{(h,k)}, q_{(i,l)} \backslash q_{(h,l)}], i, e+1, \mathcal{CR}', \mathcal{GU}', \mathcal{GS}', \mathcal{NG}')$
    end for
end for
return $P$
```

4.5.3 External Transitions

For each enabled external transition we perform the following steps, as presented in Algorithm 12:
Update transition: First we update the transition according to the constraints of the previous silent transitions, as done for silent and internal transitions.
External transition in the tree: If there already exists a transition with the same label in the current location of the tree, we disjunct the current guard to it, to ensure that the transition is always enabled when the currently processed transition is (Line 7). If it does not yet exist, we create a new location in the tree (Line 9). Then, we create a new transition leading to this new location, with the label of the currently processed transition, resetting the clock x_i and g (the guard we gain from updating the transition) as guard (Line 9).
Update $\mathcal{NG}$: We update $\mathcal{NG}$ by building diagonal constraints in g. This is done by subtracting x_i from every clock that appears in g. These diagonal constraints can in later transitions decide, whether g was enabled in the current transition. By attaching them to the following transitions, we can ensure that even though we possibly disjuncted the guard g to other guards (and thus weakened the guard), the following transitions will only be enabled *iff* g was enabled. Details can be found in the section about determinization (Section 4.3).

4.6 Restrictions

The discussed algorithms face several restrictions. In this section, we want to discuss the most severe ones:

- **State-space explosion.** As in the last chapter, the first and most obvious limitation is the state-space explosion. Due to the explicit unfolding, there is an exponential increase in the number of locations and transitions. While our on-the-fly algorithm already drastically reduces the amount

Algorithm 12 Processing external transitions in the NTA.

```
Input: $(q_t, q_i, i, e, \mathcal{CR}, \mathcal{GU}, \mathcal{GS}, \mathcal{NG})$
Output: $\{(q'_t, q'_i, i+1, 0, \mathcal{CR}', \mathcal{GU}', \mathcal{GS}', \mathcal{NG}')\}$
P ← ∅  ($P \leftarrow \emptyset$)
for each location $q_{(i,j)} \in q_i$ do
    for each action $a \in \Sigma^I_e \cup \Sigma^O_e$ do
        for each transition $t_i = (q_{(i,j)}, a, g_i, \rho_i, q_{(l,j)})$ do
            $(g, \mathcal{CR}', \mathcal{GS}') \leftarrow$ update transition$(t_i)$
            if $\exists t_t = (q_t, a, g_t, \rho_t, q'_t)$ then
                $g_t \leftarrow g_t \vee g$
            else
                $q'_t \leftarrow$ new location; $t_t \leftarrow$ new transition$(q_t, a, g, \{x_i\}, q'_t)$
            end if
            $\mathcal{NG}' \leftarrow$ update $\mathcal{NG}$(g)
            $P \leftarrow P \cup (q'_t, q_i[q_{(i,j)} \backslash q_{(l,j)}], i+1, 0, \mathcal{CR}', \mathcal{GU}, \mathcal{GS}', \mathcal{NG}')$
        end for
    end for
end for
return $P$
```

of iterations through the state-space that are needed for the determinization process, the size of the resulting tree stays the same and may pose an obstacle for further processing.

In Chapter 6 we describe how the tree can be pruned, while still creating *tioco*-conform partial models, and we describe a symbolic approach for the test-case generation, that includes building a symbolic tree and turned out to be efficient. Both approaches help reducing the state-space explosion.

- **Weakening of Timed Automata Definition.** By keeping diagonal constraints and disjunction in our final timed automata, they do not comply to the standard definition of timed automata anymore. The main drawback of this is that our timed automata cannot be processed by tools that use *Difference Bounded Matrices* (DBMs), as for instance UPPAAL.

  However, in the context of bounded model-checking with SMT-solvers, neither disjunction nor diagonal constraints pose any problem, and the model-based mutation testing works well with this extended form of timed automata.

- **Data variables.** During the determinization, we merge transitions with the same label. This can be done, since transitions on same depth of the tree always reset the same clock. However, it restrains the usage of data variables, as the merged transitions may assign different values to a data variable, and during the merging, either one of the assignments would be lost or we would need to allow non-deterministic assignments. Thus, the algorithm is restricted to timed automata without data variables.

  This could only be solved by completely leaving the structure of timed automata, and directly creating a deterministic SMT-formula for the step relation, that could later be used for e.g. test-case generation.

4.7 Implementation

All algorithms discussed in this chapter were implemented as an additional feature of the tool MoMuT::TA. As already mentioned, this is a tool that was originally implemented by the author of this thesis, and reimplemented by Willibald Krenn [113]. The additional features were added to the reimplementation, but programmed by the author of this thesis.

The algorithms were implemented in Scala (Version 2.10.3). The determinization algorithm uses the SMT-solver Z3 [73] for checking satisfiability of guards.

4.8 Proofs

4.8.1 Proof of Theorem 4.1 [Silent Transitions Removal]

Given a non-deterministic timed automaton with silent transitions A in the form of a finite tree, we need to show that our algorithm of removing the silent transitions results in an equivalent timed automaton, that is, $\mathcal{L}(O(A)) = \mathcal{L}(A)$. We will show that if A' is the result of removing one first silent transition then A and A' are equivalent: for every timed trace of A there is an equivalent timed trace of A' and vice versa, in the sense that the corresponding observable timed traces are identical.

So, let $\tau_{s,0}$ be a first silent transition on a path γ that starts at the initial location. Let $\tau_{s,0}$ be from location q_s to location $q_{s,0}$, let q_{s-1} be the location that leads to q_s and let q_{s+1} be a location that follows $q_{s,0}$ on the path. Let A' be the automaton that results after removing τ and performing the steps as in Algorithm 8. Clearly, for every run that does not pass through $\tau_{s,0}$ there is an identical run in the other automaton. Thus, we restrict ourselves to runs though $\tau_{s,0}$. We will mostly restrict ourselves to strict inequalities, as the extension to the other cases (strict inequality versus weak inequality or weak inequality versus weak inequality) is straight forward.

4.8.1.1 $\mathcal{L}(A) \subseteq \mathcal{L}(A')$.

Let ρ be a run on A through γ. We need to show that there exists a run ρ' on A' with the same observable trace as of ρ. The run ρ' will go through the same locations and transitions as does ρ, except for the part q_{s-1}, τ_s, q_s, $\tau_{s,0}$, $q_{s,0}$ in A which will be replaced by the bypass q_{s-1}, τ_s', $q_{s,0}$ in A' as in Figure 4.4. The dates of the transitions will also be the same, except for the silent transition that is missing in ρ'. That is, if t_s, $t_{s,0}$ and t_{s+1} are the dates of ρ at the transitions τ_s, $\tau_{s,0}$ (the silent transition) and τ_{s+1} then the corresponding transitions of ρ' will take place at t_s (the time of the bypass) and t_{s+1}.

We first need to show, that the guard of the bypass transition, $g_s' = g_s \wedge eg(\tau_{s,0})$, is satisfied at time t_s. The enabling guard consists of unary constraints and binary constraints. The unary constraints are simply the upper bound limits of the guard of the silent transition. The time of the bypass t_s lies before the time of the silent transition $t_{s,0}$, and the upper bounds are satisfied at $t_{s,0}$. Thus, they also have to be satisfied at the sooner time t_s.

The binary constraints are built by comparing the upper bounds of all clocks with the lower bounds of all other clocks, where $x_j < n_j$ and $m_i < x_i$ build the constraint $n_j - x_j > m_i - x_i$. This constraint ensures, that at time t_s the delay needed to reach the upper bound of x_j (which would disable the silent transition), is higher than the delay needed to reach the lower bound of x_i (which enables the silent transition). As the silent transition is enabled at $t_{s,0}$, we know that all lower bounds can be be reached, without violating any of the upper bounds. Thus, the binary constraints are satisfied.

We have seen that all the constraints of $eg(\tau_{s,0})$ are satisfied at time t_s and so the constraint g_s' of ρ' is satisfied at t_s and the transition τ_s' can be taken.

The next step is to show that the transition τ_{s+1} with guard g'_{s+1} of ρ' from location $q_{s,0}$ to location q_{s+1}, as well as the next transitions τ_{s+j}, $j = 2, \ldots, p$, with guards g'_{s+j} can be taken at the same dates t_{s+j} on which τ_{s+j} are taken in ρ on guards g_{s+j}, $j = 1, \ldots, p$.

If the silent transition happens to be on an exact time: $x_i = n_i$ then the update of the future guards that refer to the clock $x_{s,0}$ that was reset at $\tau_{s,0}$ is clear: each occurrence of $x_{s,0}$ is replaced by $x_i - n_i$, and we are done. Hence, suppose that there are no exact constraints at the silent transition.

We write the guard $g'_{s,0}$ of the silent transition $\tau_{s,0}$ as:

$$g'_{s,0} = 0 \leq x_s \wedge \bigwedge_{i=2,\ldots,r} m_i < x_i < n_i, \tag{4.3}$$

where for some of the clocks x_i there may be only a lower bound or only an upper bound constraint.

The constraints on $x_{s,0}$ at the transitions τ_{s+j}, $j = 1, \ldots, p$ contain $0 \leq x_{s,0}$ in τ_{s+1} and are of the general (strict inequalities) form $m_{s+j} < x_{s,0} < n_{s+j}$ in τ_{s+j}. The corresponding updated constraints of A' at time t_{s+j}, $j = 1, \ldots, p$, are

$$\bigwedge_{i=1,\ldots,r} m_i + m_{s+j} < x_i < n_i + n_{s+j}. \tag{4.4}$$

First, we need to show that the taken guard $tg(\tau_{s,0})$ is satisfied at time t_{s+1}. The taken guard is the constraint $0 \leq x_{s,0}$. After the update of the future guards this constraint is replaced by the conjunction of all the lower bound constraints $m_i < x_i$ of $g'_{s,0}$. But since these lower bound constraints are satisfied at the time $t_{s,0}$ of the silent transition (in ρ) then clearly they are satisfied at t_{s+1}, $t_{s+1} \geq t_{s,0}$, that is, the updated taken guard $tg(\tau_{s,0})$ is satisfied in ρ'.

Let us look at the other updated future constraints. Since at the time of the silent transition $x_{s,0} = 0$ and $m_i < x_i$ then at time t_{s+j} when $m_{s+j} < x_{s,0}$ we have $m_i + m_{s+j} < x_i$. With a similar argument for the upper bound constraints, we see that the constraints of (4.4) are satisfied in ρ'.

Also the part of the synchronization rules is clear since it refers to the possible minimum and maximum time difference between every two transitions on which $x_{s,0}$ occurs, and since the run ρ goes through these transitions it assures that these constraints can be satisfied. So, for example, the synchronization constraint $m_{s+j} - n_{s+i} < x_{s+i} < n_{s+j} - m_{s+i}$ that is added to the guard g_{s+j} of τ_{s+j}, refers to the time difference $t_{s+j} - t_{s+i}$ between the transition τ_{s+i} and the transition τ_{s+j}, $i < j$.

Note that the synchronization with the constraint $0 \leq x_{s,0}$ of τ_{s+1} results in adding to τ_{s+j}, $j = 1, \ldots, p$ the constraint $x_{s+1} < n_{s+j}$, that is $t_{s+j} - t_{s+1} < n_{s+j}$, which clearly is satisfied since $t_{s+j} - t_{s,0} < n_{s+j}$.

We showed that the observable trace of ρ' is the same as that of ρ and this completes the proof of $\mathcal{L}(A) \subseteq \mathcal{L}(A')$.

4.8.1.2 $\mathcal{L}(A') \subseteq \mathcal{L}(A)$.

Let ρ' be a run on A' going through the bypass τ'_s. We will show that there exists a run ρ through $\tau_{s,0}$ in A with the same observable trace as of ρ'.

The first thing we need to check is that the silent transition $\tau_{s,0}$ can be taken, given that the enabling guard $eg(\tau_{s,0})$ is satisfied at time t_s. The unary constraints $x_j < n_j$ ($x_j \leq n_j$) of $eg(\tau_{s,0})$ guarantee that each of the constraints in the guard $g'_{s,0}$ of the silent transition $\tau_{s,0}$ can be satisfied separately at some time that is equal to or is later than t_s. Then, in order to show that all the constraints could be satisfied simultaneously, it suffices to show that the minimum upon the time delays to the upper bound constraints of the clocks appearing in $g'_{s,0}$ is greater than the maximum upon the time delays to the lower bound constraints in $g'_{s,0}$ (the 'greater' should be replaced by 'greater or equal' in case both the maximum and minimum come from weak inequalities):

$$\min_j(n_j - x_j) > \max_i(m_i - x_i). \tag{4.5}$$

But this condition is equivalent to the condition that $n_j - x_j > m_i - x_i$ at time t_s for every i, j, which is exactly the conjunction of diagonal constraints

$$\bigwedge_{i \neq j} x_j - x_i < n_j - m_i \tag{4.6}$$

of $eg(\tau_{s,0})$.

Thus, we know that the silent transition $\tau_{s,0}$ can be taken in the run ρ at some time $t_{s,0}$ after a delay of $M = \max_i(m_i - x_i)$ from t_s (this delay is not negative since we introduced the constraint $0 \leq x_s$) and before a delay of $N = \min_j(n_j - x_j)$.

It remains to show that the transitions $\tau_{s+1}, \ldots, \tau_{s+p}$ on guards $g_{s+1}, \ldots, g_{s+p}$ of ρ can be taken at the same dates $t_{s+1}, \ldots, t_{s+p}$ as the corresponding transitions on guards $g'_{s+1}, \ldots, g'_{s+p}$ are taken in ρ'.

To be more specific, it suffices to prove that there exists $t_{s,0}$ with the following conditions:

1. $t_s \leq t_{s,0} \leq t_{s+1}$;
2. $g'_{s,0}$ is satisfied at $t_{s,0}$;
3. the constraints on $x_{s,0}$ are satisfied at $t_{s+1}, \ldots, t_{s+p}$, with $x_{s,0}$ reset at $t_{s,0}$.

For the second condition the constraints of $g'_{s,0}$ that should be satisfied at time $t_{s,0}$ are

$$\bigwedge_{i=1,\ldots,r} m_i < x_i(t_{s,0}) < n_i. \tag{4.7}$$

Equivalently, at each time t_{s+j}, $j = 1, \ldots, p$:

$$\bigwedge_{i=1,\ldots,r} m_i + t_{s+j} - t_{s,0} < x_i(t_{s+j}) < n_i + t_{s+j} - t_{s,0}, \tag{4.8}$$

or,

$$\bigwedge_{i=1,\ldots,r} m_i - x_i(t_{s+j}) + t_{s+j} < t_{s,0} < n_i - x_i(t_{s+j}) + t_{s+j}. \tag{4.9}$$

For the third condition the constraints on $x_{s,0}$ that should be satisfied at times $t_{s+1}, \ldots, t_{s+p}$ are $m_{s+j} < x_{s,0}(t_{s+j}) < n_{s+j}$ for $j = 1, \ldots, p$. The constraint here at time t_{s+1} is $0 \leq x_{s,0}(t_{s+1})$ possibly conjuncted with other constraints (for convenience we wrote all constraints as strict inequalities). This is equivalent to

$$\bigwedge_{j=1,\ldots,p} m_{s+j} < t_{s+j} - t_{s,0} < n_{s+j} \tag{4.10}$$

or

$$\bigwedge_{j=1,\ldots,p} -n_{s+j} + t_{s+j} < t_{s,0} < -m_{s+j} + t_{s+j}. \tag{4.11}$$

We need to show that the constraints on $t_{s,0}$ of (4.9) and (4.11) do not define an empty set. This condition is equivalent to showing that the set S_1 of the above expressions to the left of $t_{s,0}$ is smaller than the set S_2 of the expressions to the right of $t_{s,0}$ (equivalently that the maximum of S_1 is smaller than the minimum of S_2), where

$$S_1 = \{m_i - x_i(t_{s+j}) + t_{s+j} \,|\, i = 1, \ldots, r,\ j = 1, \ldots, p\} \cup \{-n_{s+j} + t_{s+j} \,|\, j = 1, \ldots, p\}, \tag{4.12}$$

and

$$S_2 = \{n_i - x_i(t_{s+j}) + t_{s+j} \mid i = 1, \dots, r,\ j = 1, \dots, p\} \cup \{-m_{s+j} + t_{s+j} \mid j = 1, \dots, p\}. \tag{4.13}$$

There are two types of expressions in S_1 and two types of expressions in S_2, hence we need to check that the following 4 cases are satisfied.

4.8.1.3 Case 1: $m_i - x_i(t_{s+j}) + t_{s+j} < n_{i'} - x_{i'}(t_{s+j'}) + t_{s+j'}$.

This inequality is equivalent to

$$m_i - x_i(t_{s,0}) + t_{s,0} < n_{i'} - x_{i'}(t_{s,0}) + t_{s,0}, \tag{4.14}$$

or to

$$m_i - x_i(t_{s,0}) < n_{i'} - x_{i'}(t_{s,0}). \tag{4.15}$$

The latter is equivalent to

$$x_{i'}(t_s) - x_i(t_s) < n_{i'} - m_i, \tag{4.16}$$

which is (4.6), the enabling guard $eg(\tau_{s,0})$ that is satisfied at time t_s of the run ρ'.

4.8.1.4 Case 2: $m_i - x_i(t_{s+j}) + t_{s+j} < -m_{s+j'} + t_{s+j'}$.

This inequality is equivalent to

$$m_i - x_i(t_{s+j'}) + t_{s+j'} < -m_{s+j'} + t_{s+j'}, \tag{4.17}$$

$$m_i - x_i(t_{s+j'}) < -m_{s+j'}, \tag{4.18}$$

$$m_i + m_{s+j'} < x_i(t_{s+j'}). \tag{4.19}$$

The last inequality is no other than one of the left inequalities of (4.4), which are the updated future constraints in A' of the reset clock $x_{s,0}$, and thus are given to be satisfied.

4.8.1.5 Case 3: $-n_{s+j'} + t_{s+j'} < n_i - x_i(t_{s+j}) + t_{s+j}$.

This inequality is equivalent to

$$-n_{s+j'} + t_{s+j'} < n_i - x_i(t_{s+j'}) + t_{s+j'}, \tag{4.20}$$

$$-n_{s+j'} < n_i - x_i(t_{s+j'}), \tag{4.21}$$

$$x_i(t_{s+j'}) < n_i + n_{s+j'}. \tag{4.22}$$

The last inequality is one of the right inequalities of (4.4), which are the updated future constraints in A' of the reset clock $x_{s,0}$, and thus are given to be satisfied.

4.8.1.6 Case 4: $-n_{s+i} + t_{s+i} < -m_{s+j} + t_{s+j}$.

This inequality is equivalent to

$$m_{s+j} - n_{s+i} < t_{s+j} - t_{s+i}. \tag{4.23}$$

The inequality certainly holds when $i = j$. When $i < j$ we can write this inequality with the clock x_{s+i} that is reset at time t_{s+i} in A':

$$m_{s+j} - n_{s+i} < x_{s+i}(t_{s+j}). \tag{4.24}$$

But the last inequality can be found in the first row of Table 4.3 which contains the synchronization constraints of the updated future constraints in A' of the reset clock $x_{s,0}$.

Similarly, when $j < i$ we need to satisfy the inequality

$$x_{s+j}(t_{s+i}) = t_{s+i} - t_{s+j} < n_{s+i} - m_{s+j}, \tag{4.25}$$

which can be found in the fourth row of Table 4.3.

We showed that the set of possible time values $t_{s,0}$ for the silent transition in ρ is not empty, that is, there is a solution to the sets of inequalities (4.9) and (4.11) in the indeterminate $t_{s,0}$ (the extension to weak inequalities is straight forward).

To complete the proof it remains to show that the solution for $t_{s,0}$ satisfies the first condition, that is, that $t_s \le t_{s,0} \le t_{s+1}$. Well, the left inequality $t_s \le t_{s,0}$ comes from satisfying the inequality $m_i - x_i(t_{s+j}) + t_{s+j} \le t_{s,0}$ of (4.9) with $x_i = x_s$ and $m_i = m_s = 0$ (it refers to augmenting the silent transition guard with the constraint $0 \le x_s$). This inequality is equivalent to $0 - x_s(t_s) + t_s \le t_{s,0}$ or $t_s \le t_{s,0}$ since x_s was reset at time t_s.

The right inequality comes from satisfying the inequality $t_{s,0} \le -m_{s+1}+t_{s+1}$ of (4.11) with $m_{s+1} \ge 0$, that is, $t_{s,0} \le t_{s+1}$.

4.8.2 Proof of Theorem 4.2 [Determinization]

The deterministic property of $D(A)$ follows from the fact that when merging α-transitions into τ_{acc} and $\tau_{\neg acc}$ then the guard of $\tau_{\neg acc}$ is a conjunction of some guard with the negation of the guard of τ_{acc}. Hence, different runs will induce different time traces.

In general, by merging locations of A in $D(A)$ we may only expand the language and conclude that $\mathcal{L}(A) \subseteq \mathcal{L}(D(A))$. On the other hand, the new constraints introduced in $D(A)$ may restrict the language. So, let us examine the new transformed constraints and show that they do not impose additional restrictions. Suppose the guard of transition τ contains the constraint $x \sim n$ and that y is reset on τ. Then, at the time t_0 of τ, the constraint $x(t_0) - y(t_0) \sim n$ holds. But also at time $t_1 > t_0$, the constraint $x(t_1) - y(t_1) \sim n$ holds since x and y progress at the same rate. Hence, for any run through τ in A there exists a corresponding run in $D(A)$ with the same trace because the additional constraints of the form $x - y \sim n$ that are added to the future guards are satisfied automatically by all runs in $D(A)$ that satisfy the guard of τ. Thus, it remains $\mathcal{L}(A) \subseteq \mathcal{L}(D(A))$.

To show that the language of $D(A)$ does not contain accepting traces that are not in the language of A it suffices to show that when a transition in a merged location of $D(A)$ is enabled then the corresponding original transition in A is enabled. But this is indeed the case since for each transition of $D(A)$ we first copy to its guard the transformed guard of the transition that leads to it, and this transformed guard contains all the history: the transformed guards of the path that leads to this transition. That is, by induction one shows that since the record of paths of level n are passed to paths of level $n + 1$ then it holds for every level.

5 Case Studies and Results

Parts of this chapter are based on our publications at FORMATS 2015 [123], A-MOST 2015 [15] and TASE 2016 [16].

This chapter will first present our industrial case studies and then provide experimental results for the methodologies and algorithms presented in the last chapters. First, we will present results of the basic test-case generation approach for deterministic systems, when applied to the specification of a car alarm system. The study will focus on runtime and the ability to catch bugs in real Java implementations. Next, we will present the performance of the bounded determinization approach, where we will first show some purely scientific examples, including one that is a network of timed automata, where we only perform the determinization. Then, we show the complete workflow including silent transition removal, determinization and test-case generation performed on the two industrial case studies. All experiments were run on a MacBook Pro with a 2.8 GHz Intel CoreI i7 Processor and 8 GB RAM.

5.1 Industrial Case Studies

In this section we present the two case studies we used for evaluation of the previously defined test-case generation technique and the silent transition removal approach. We will first present the specification of a car alarm system that was developed in a previous project of our research group and then present the specification of an adjustable speed limiter that was developed within CRYSTAL.

5.1.1 Car Alarm System

In this section we present the timed automata specification of a Car Alarm System (CAS) [5, 149]. The car alarm system is a model inspired by Ford's demonstrator developed in the EU FP7 project MOGENTES[4]. We developed the TAIO model of the CAS from the requirements provided by Ford, given below:

CAS Requirements.

Arming: The system is armed $20sec$ after the vehicle is locked and the bonnet, luggage compartment and all doors are closed;

Alarm: The alarm sounds for $30sec$ if an unauthorized person opens the door, the luggage compartment or the bonnet. The hazard flasher lights flash for $5min$;

Deactivation: The anti-theft alarm system can be deactivated at any time, even when the alarm is sounding, by unlocking the vehicle from outside.

The TAIO model of the car alarm system consists of 16 locations, 25 transitions, 11 invariants and 5 clocks and can be seen on the left-hand side of Figure 5.1. The model contains two *mixed states* (the locations labeled with q_1 and q_2), where inputs and outputs are enabled simultaneously (at the points in time were $c = 20$ and $e = 300$, respectively). These states do not pose a problem during test-case execution, but if a test case is generated that executes an input at that exact point of time, the system under test may have already produced the output, forcing the test to deliver an *inconclusive* verdict, as the test purpose can not be reached anymore. Thus, in general it makes sense to avoid mixed states, where possible, e.g. by adding the guard $e < 300$ to the $unlock?$ transition leaving q_2.

[4] `http://www.mogentes.eu`

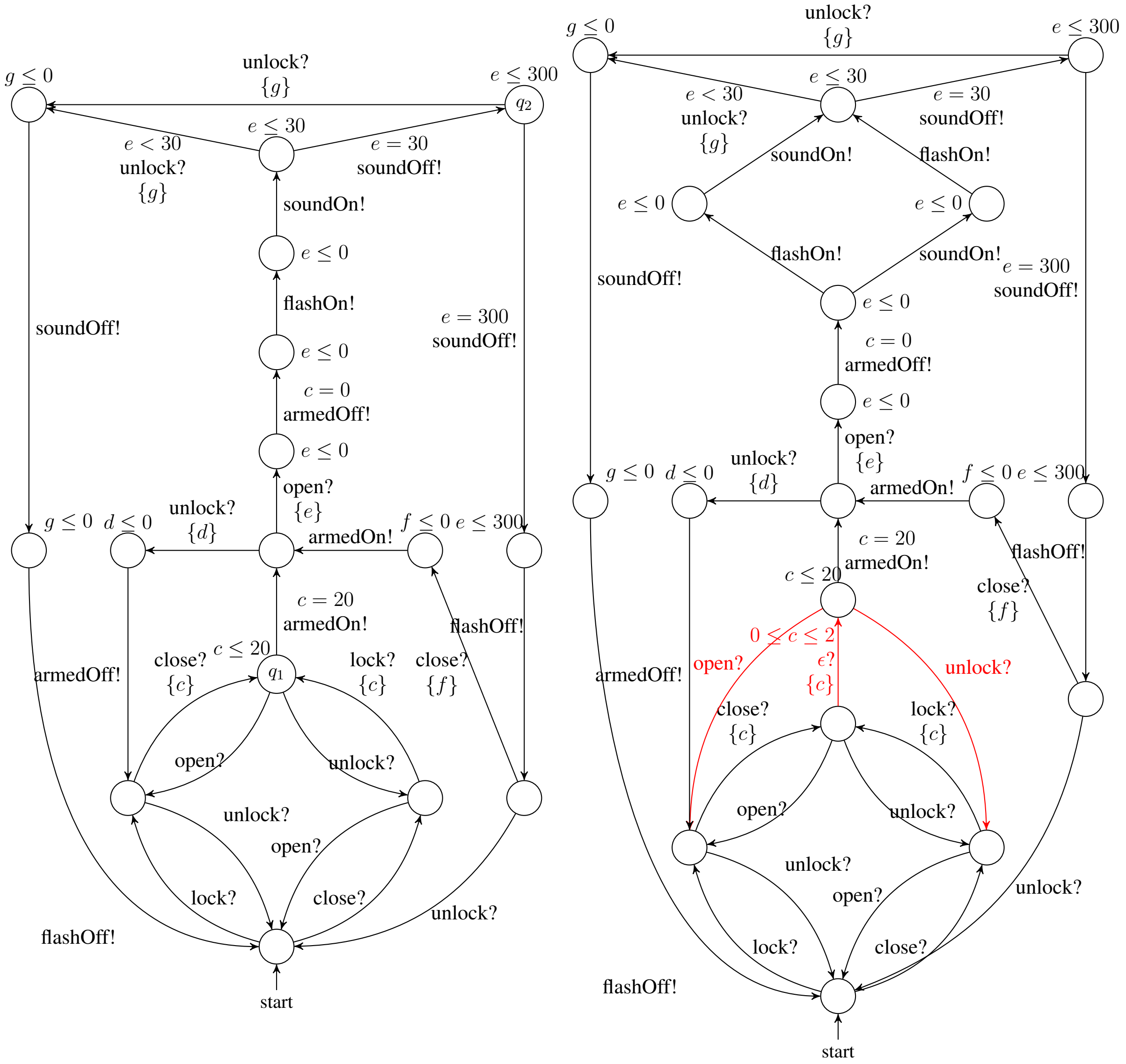


Figure 5.1: A deterministic and a non-deterministic version of the car alarm system.

Additionally, we developed a non-deterministic version, where we introduced a silent transition that adds a non-deterministic delay of up to two seconds before the timer of the alarm starts. The concrete non-deterministic model can be seen on the right-hand part of Figure 5.1. It contains one location and three transitions more than the deterministic variant.

Both models consist of four inputs (*lock?*, *unlock?*, *close?* and *open?*) and six outputs (*armedOn!*, *armedOff!*, *soundOn!*, *soundOff!*, *flashOn!* and *flashOff?*). They use 5 clocks, even though the same functionality could as well be modeled with one clock. We experimented with both variants and found that the clocks did not significantly increase the runtimes. Thus we stayed with the multiple clocks, to show that the capabilities of the approach.

5.1.2 Adjustable Speed Limiter

In this subsection we present an industrial case study, that was provided as a use case by Volvo within the European Artemis Project Crystal. The use case evolves around an adjustable speed-limiter (*ASL*),

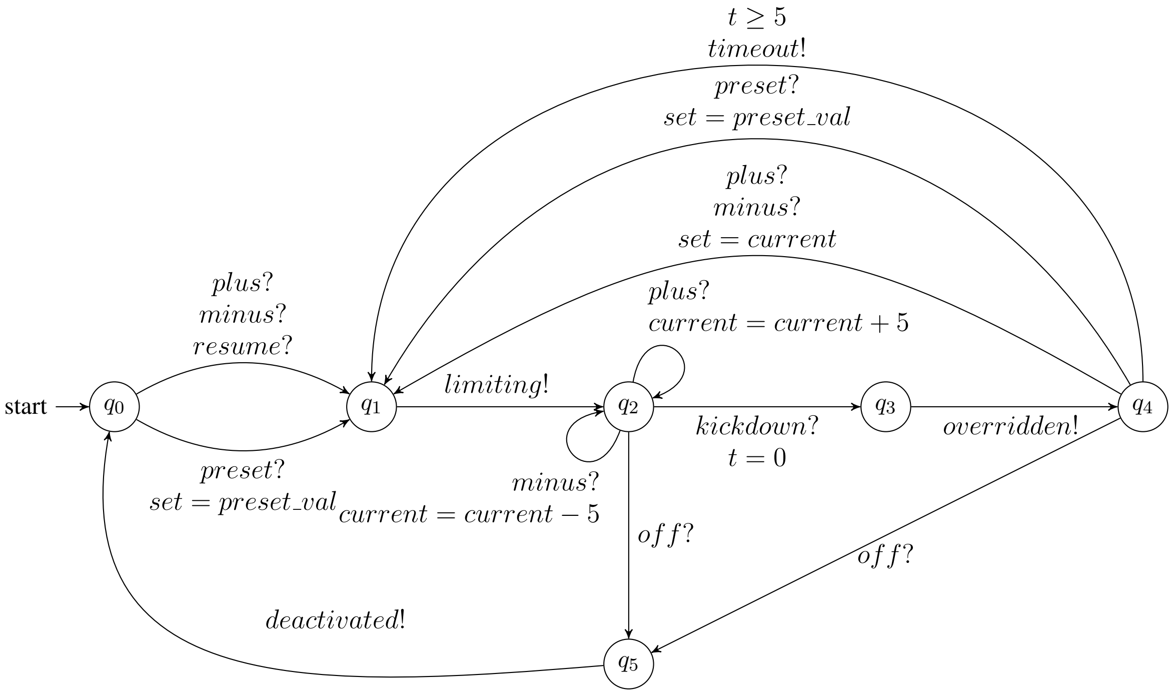


Figure 5.2: Deterministic version of the speed limiter, self-loops without variable updates are omitted.

that limits the actual car speed according to an internally stored value.

The speed limiter can assume three operating states: *deactivated*, *limiting* and *overridden*. In the *limiting* state the device is active, and *overridden* means that the user temporarily deactivated it by a kickdown of the gas pedal. In its initial state, the *ASL* is *deactivated*.

After receiving any of the inputs *preset?*, *plus?*, *resume?* or *minus?*, the speed limiter switches from *deactivated* to *limiting*. There it stays, until the user either turns it off via the *off?* input, or overrides it via a *kickdown?* input. The kickdown triggers a timed transition back to the *limiting* state, that is executed automatically after a certain delay, if there was no manual state change in between.

The current speed limit is stored internally. It can be modified by three inputs: *preset?* sets it to a predefined constant value, *plus?* increases the limit and *minus?* decreases it. However, plus and minus only change the limit, if the system is limiting, otherwise they only trigger a state change towards *limiting*.

We performed experiments on two models of different levels of abstraction of the speed limiter. The first version is deterministic, and the second one contains non-determinism and silent transitions.

Deterministic Version.

A timed automata model of the deterministic version can be seen in Figure 5.2. It contains additional outputs that are activated before each state change, denoting the new state. In this model, the time-triggered state change from the *overridden* location back to the *limiting* state is observable, and labeled by *timeout!*. The delay for the timeout was parametrized in the requirements, the delay of 5 chosen arbitrarily for this model.

Non-Deterministic Version.

For the second series of experiments, we concentrate on the state-change mechanics, neglecting the actual value of the current limit. The only important information is whether the limit is zero, lower than the predefined value that is set by the *preset?* input, equal to it, or higher. Thus, we applied qualitative abstraction to the limit and encoded these four value ranges in the locations.

Figure 5.3 gives an impression of the case study. Note that the first column of locations contains the *deactivated* states, the second column contains the *limiting* states and the third contains the *overridden* states. The first row indicates that the current limit is higher than the preset, the second row indicates that it is equal, the third that it is lower and the bottom row indicates that the limit is zero. The figure is a little simplified for presentational purposes: in the real automaton each state has an ”on entry”-transition. That is, each state consists of two locations, where all ingoing transitions lead to the first location, and there is an output transition labeled by the name of the state, leading to the second location. For example, there is a *limiting!* output for the *limiting* state. All outgoing transitions only leave the second location. These ”on entry”-transitions were neglected in the figure to ease comprehension. Additionally, for every location besides q_7 there exists a transition with label *preset?* leading to q_7, as the *preset?* input both turns the state to *limiting* and sets the limit to the predefined value. These transitions were omitted to reduce the amount of crossing and overlaying transitions.

In location q_{10}, where the current limit is higher than the preset value, a *minus?* transition may non-deterministically stay in q_{10} or lead to q_7, where the current limit equals the preset. The same non-determinism also appears in q_4. In this model we also considered the timeout a silent transition, changed its lower bound to 8 and gave it an upper bound of 10.

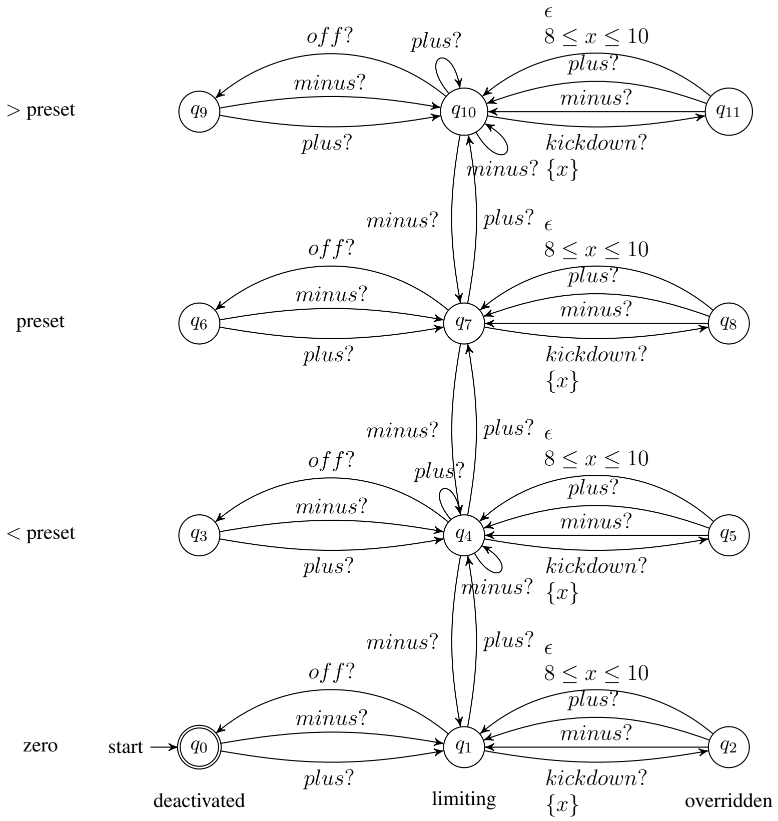


Figure 5.3: The non-deterministic version of the ajustable speed limiter, simplified for readability.

5.1.3 Overview

Table 5.1 shows the summarized characteristics of the different models. Both models have a deterministic and a non-deterministic variant.

Table 5.1: Characteristics of the two case studies.

Case study	# Locations	# Transitions	# Clocks
CAS (det.)	16	25	5
CAS (non-det.)	17	28	5
ASL (det.)	6	16	1
ASL (non-det.)	12	38	1

5.2 Test-Case Generation from Deterministic Models

In this section we will present our test-case generation on deterministic models, that did not need any preprocessing. We will first present the results for the car alarm system, and than discuss a study we did for the CRYSTAL project on the adjustable speed limiter.

5.2.1 Car Alarm System

In the first evaluation of our test-case generation approach, we applied our mutation testing tool to the CAS example. This experiment was the first one we conducted, and was used to evaluate the mutation operators and the solving approach, to check which of the techniques we should use in the following experiments. We first generated all the mutants (1099) and for each mutant checked whether it tioco-conforms to the original CAS model, by effectively doing the k-bounded language inclusion check. We set the maximal bound k to 20 for the k-bounded language inclusion check. We generated test cases from all the non-conforming mutants. The whole procedure took 62.3 minutes and produced a total of 628 test cases. 471 mutants are tioco conform to the specification and therefore did not produce any test cases. Table 5.2 shows the runtimes for the language inclusion check applied on the car alarm system and a single equivalent mutant. The first row presents the results gained using the standard solving technique of Z3 and the second row shows the results when using the incremental solving capabilities (see Section 3.5). Given the advantage of the incremental approach, all future experiments were performed using the incremental approach.

In order to evaluate our mutation testing framework, we used an existing implementation of the CAS [3], developed in Java. The implementation consists of 4 public methods, *open*, *close*, *lock* and *unlock*, and 2 internal methods, *setState* and the constructor. The CAS implementation simulates time elapse with a *tick* method. We also used the 38 implementation mutants of the CAS described by Aichernig et al. [3]. They were produced using the Java mutation tool μJava[5]. Applying all mutation operators of μJava to all methods except *tick* resulted originally in 72 mutants. Some of the mutants were equivalent to the original implementation or to other mutants. After filtering them out, the total of 38 unique faulty implementations were derived. Table 5.3 shows the total numbers of implementation mutants and equivalent ones. Both the correct and the 38 faulty CAS implementations were used to evaluate the effectiveness of the test cases we generated.

We developed a test driver in order to execute generated tests on the CAS implementation. We integrated quiescence in the test driver, which is responsible to detect prolonged absence of outputs. We

[5] http://cs.gmu.edu/~offutt/mujava/

Table 5.2: Computation time for k-bounded language inclusion check.

k	5	10	15	20
Standard Solving	$0.1s$	$40.1s$	$115.2s$	$279.5s$
Incremental Solving	$0.1s$	$0.3s$	$0.6s$	$1.0s$

set the maximal timeout that the driver is allowed to wait for an output action to 400 time units. If the timeout is reached without observing an action, the test outputs a verdict **pass** if the test is in the last location with the invariant being true or **inc** otherwise. The test driver immediately emits an input action when the associated transition becomes enabled. If the timeout is reached before the transition labeled by the input action becomes enabled, the test driver gives the **inc** verdict. Note that we executed tests on a Java implementation which models time passage as discrete ticks. We can currently interface our test driver to any simulated implementation model with an arbitrary model of time, as long as time is simulated and communicated in form of time stamps.

We say that a faulty implementation is *killed* if at least one test case reaches the verdict **fail** during a test execution. We analyzed the effectiveness of our mutation operators with respect to their ability to kill faulty implementations. Table 5.4 summarizes the results on effectiveness of mutation operators, where each row provides the number of mutants, the number of resulting test cases, the average number of faulty implementations killed per test case and the *mutation score* of a mutation operator. The average number of faulty implementations killed per test case is built by executing all test cases of a mutation operator on all implementations, checking for each test how many implementations are killed and computing the average. The tests gained by mutation operator M1 kill an average of 12.5 faulty implementations per test. Tests that kill many different faulty implementations should be executed first during test-case execution, to reduce the number of executed tests until a bug is found. Mutation score is the measure which gives the percentage of faulty implementations killed by mutants resulting of a single mutation operator. A high value indicates that the tests are rather different to each other, and cover different paths, thus being able to detect more faults. It is interesting to note, that the tests gained by mutation operators M1 and M2 kill comparatively few implementations per individual test case, but all tests combined achieve a very high mutation score. This shows, that they achieve their high mutation score mostly due to their big number (267 and 165 tests), but the individual tests are not very strong on average.

We achieved a total of 100% mutation score for the combined mutation operators. The highest mutation score is achieved by the "change target" operator $M2$, at the price of generating 375 mutants and 267 test cases. Evaluation results also show that most of the faulty implementations are killed by $M2$-mutants which contain self-loops. We also observed that 3 faulty implementations were only killed by "sink state" mutants ($M7$).

Following the above observations, we conducted another experiment in which we only applied "sink

Table 5.3: Injected faults into the CAS implementation.

	Mutants	Equiv.	Pairwise Equiv.	Different Faults
SetState	6	0	1	5
Close	16	2	6	8
Open	16	2	6	8
Lock	12	2	4	6
Unlock	20	2	8	10
Constr.	2	0	1	1
Total	72	8	26	38

Table 5.4: Mutation analysis of mutation operators. The list of mutation operators can be seen in Section 3.1.

	M1	M2	M3	M4	M5	M6	M7	M8	**Total**
Model Mutants [#]	139	375	375	24	25	11	25	125	1099
TCs [#]	139	267	165	6	3	11	25	12	628
av. Kills per TC [#]	12.5	13.2	12.4	16.3	16.3	17.8	17.8	13.8	13
Mutation Score [%]	71	94.7	92.1	57.9	47.4	60.5	89.5	57.9	**100**

state" and "self-loop" mutations, resulting in only 50 mutants. All mutants were shown to be non tioco-conformant to original models, generating 50 test cases in just $56s$. In addition, combining the test cases of these two operators resulted in 100% mutation score. These results indicate that a smart choice of a small subset of mutation operators can achieve high mutation scores while considerably reducing test-case generation and execution times. However, we decided to use all mutation operators for the following experiments, as the good results achieved by this subset may be specific to this use case.

5.2.2 Adjustable Speed Limiter

We used the deterministic model of the adjustable speed limiter for assessing the quality of the test suite that was already in use at Volvo, by checking which of our model mutants can be killed by the test suite. Therefor, we translated the already existing test suite into timed automata test cases. As the specified part is only a subcomponent, and the original test suite was designed to test the whole system, only 9 of the test cases were relevant to this part.

We then generated the mutants for the ASL and performed conformance checks between the existing test cases and the mutants. This can be done, by considering the test case as the specification and the mutants as implementation and checking whether the mutants conform to the test cases. Only for the mutants that were not yet killed by the existing test cases, we performed the test-case generation. Analyzing which mutants were already killed took 363 seconds for 393 mutants. The test-case generation took 49.6 seconds for the unkilled 248 mutants. Details are given in Table 5.5. Processing all 248 mutants took 49.6 seconds, with an average of 0.2 seconds per mutant. The longest equivalence check for a single mutant tool 0.5 seconds. Thus, the test-case generation was very fast, and there were no statistical outliers.

The approach produced 11 additional test cases. Grischa Liebel from Chalmers discussed these additional test cases with engineers at Volvo and identified three main reasons for the additional tests:

- **Self-Transitions.** As self-transitions were implicitly modeled in the textual requirements, the provided test cases did not take them into account, and they where thus not tested. If any of them was implemented wrongly, i.e. setting a wrong target state, or changing the variables, this was not tested. Six of the new test cases were introduced into the test suite of Volvo for testing self-transitions.
- **Missing Combinations.** Three of the test cases contained combinations of events that were not covered by the original test cases, e.g. triggering the $plus?$ input after the $minus?$ input in the

	depth	mean	min	max	Q_1	Q_2	Q_3	total
Test-case generation	6	0.2	0.13	0.5	0.18	0.19	0.2	49.6

Table 5.5: Runtimes of the test-case generation for the deterministic adjustable speed limiter. All numbers are given in seconds.

overridden state. The engineers pointed out, that analyzing the corresponding mutants, to figure out why the combination makes sense, was very useful.

- **Missing Test Cases.** Two of the test cases were generated to test the automatic timeout of the overridden state, that was not covered at all by the existing tests. This was most likely caused by the level of abstraction we chose and some of the less abstract test cases for the complete system probably covered the time out behavior. However, the test cases we received for analysis did not.

5.3 Test-Case Generation from Non-Deterministic Models

In this section we present the results of applying our complete workflow to non-deterministic timed automata specifications. However, the first subsection (Section 5.3.1) will present some scientific examples, where we only perform the determinization and silent transition removal. The two following subsections will show results for the non-deterministic versions of the car alarm system and the adjustable speed limiter. However, since we did not have any non-deterministic implementations, we only performed the determinization and test-case generation, but could not evaluate the quality of the test suites.

5.3.1 Scientific Studies

We would like to start the evaluation with two small examples that were introduced in related work on determinization and silent transition removal to show that these operations are not possible in general.

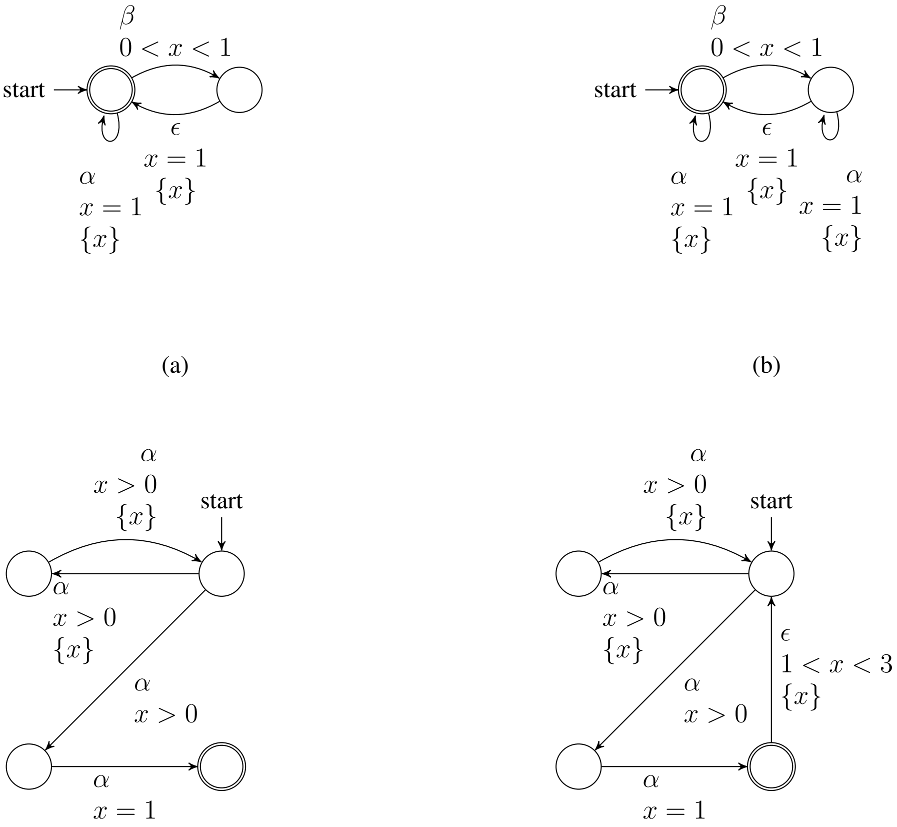


Figure 5.4: The four timed automata used in Study 1 and Study 2.

Depth	Number of locations				Runtime (sec.)			
	unfolded	std. det.	new det.	on-the-fly	ϵ-removal	std. det.	new det.	on-the-fly
Partial 2								
2	8	7	7	7	0.1	0.3	0.1	0.1
5	78	63	63	63	0.4	0.5	0.4	0.2
9	1,278	1,023	1,023	1,023	16,011.2	6.7	7.2	1.0
2	9	8	8	8	0.2	0.2	0.2	0.1
5	177	135	84	63	0.8	0.9	1.3	0.7
9	8,361	4,364	3,609	1,023	20,969.0	71.2	88.3	9.6
Partial 2								
2	5	5	4	4	-	0.1	0.1	0.1
5	11	10	8	8	-	0.2	0.3	0.1
10	21	21	16	16	-	0.3	0.3	0.1
25	51	50	38	38	-	0.5	0.9	0.2
50	101	100	76	76	-	0.7	391.6	0.3
2	5	5	4	4	0.1	0.1	0.1	0.01
5	24	26	8	8	0.2	2.1	0.4	0.3
10	140	661	16	16	0.5	1,945.1	2.1	0.5

Table 5.6: Runtime and number of locations for the automata of r Study 1, i.e. the automata of Fig. 5.4 (a) (first three rows) and Fig. 5.4 (b) (rows 4-6) and Study 2, i.e. Figure 5.4 (c) (rows 7-12) and Figure 5.4 (d) (last three rows).

The following studies compare the numbers of locations and the runtimes of (a) the silent transition removal, (b) a standard determinization algorithm that works by splitting non-deterministic transitions into several transitions that contain every possible combination of their guards (as e.g. the algorithm applied by Wang et al. [173]), (c) the new determinization algorithm introduced in Section 4.3 and (d) its on-the-fly version, as introduced in Section 4.5. The runtimes presented in the rows labeled by determinization do not include the removal of silent transitions.

Study 1. The first example, taken from Diekert et al. [75], is the timed automaton illustrated in Figure 5.4 (a), where the silent transition cannot be removed, as there is no unbounded observable automaton with the same language. We then added another α-transition (Figure 5.4 (b)), which causes non-determinism after removing the silent transition.

Study 2. The second example is taken from Baier et al. [39] and is illustrated in Figure 5.4(c). We modified the automaton by adding a silent transition (Figure 5.4(d)).

The results of the two studies are shown in Table 5.6, including for both examples the results before and after modification. As can be seen on both examples, the number of locations is increasing exponentially with the depth of the unfolding, even though the determinization may reduce the number a little, if transitions can be merged. The on-the-fly algorithm can reduce them even further by building a directed acyclic graph instead if a tree The advantage of the on-the-fly algorithm with respect to runtime also shows very well, with a runtime reduction from over $20,000$ seconds to ten seconds, in the most extreme case.

Networks of Timed Automata. Now we want to analyze the NTA of a coffee machine with three components, that was already introduced in Chapter 2 in Figure 2.2. It contains non-determinism, as the buttons for ordering the different drinks are all labeled by ”button” and silent transitions which are enabled when the preparation of a drink is finished.

Table 5.7 shows the results of applying the on-the-fly algorithm for networks of timed automata in terms of size of the unfolded network and calculation time for different depths. We extended the 2nd

Depth	Number of locations				Runtime (sec.)			
	2 drinks	3 drinks	4 drinks	5 drinks	2 drinks	3 drinks	4 drinks	5 drinks
NTA	13	16	19	22	-	-	-	-
10	37	93	191	343	0.1	0.1	0.3	0.4
20	381	3,279	16,384	58,593	0.4	1.5	5.3	11.6
30	4,093	147,620	1,747,160	timeout	1.7	26.3	281.5	timeout

Table 5.7: Runtimes of the on-the-fly algorithm for different versions and depths of the coffee machine NON-DET(TA).

and 3rd automata in the network, to enable selecting a broader variety of drinks (still all buttons are non-deterministically labeled by $button?$), producing automata with 2–5 choices. As expected, we experience an exponential increase in complexity depending on depth and amount of choices (2–5 drinks). Fortunately, in most realistic cases lower depths are sufficient, e.g. here, Depth 3 covers all observables.

We also performed conformance checks with an equivalent mutant (thus it needs to explore the complete state-space, as there is no counter example allowing an early termination) for the coffee machine with 2 drinks. It takes 2.3 seconds on Depth 10, 105.8 seconds on Depth 20 and runs into a timeout (2 hours) on Depth 30.

5.3.2 Car Alarm System

We now want to evaluate the approach on the non-deterministic car alarm system that was already presented in Section 5.1.1. The results of the determinization are given in Table 5.8. We were able to perform the removal of silent transitions and the new determinization up to Depth 12, and the standard determinization up to Depth 8. The on-the-fly algorithm still only took ten seconds for Depth 12, but it also shows a $10-$ times increased runtime on Depth 12 compared to Depth 8.

However, the size of the determinized automaton is beyond the capabilities of our test-case generation tool and cannot be processed anymore. We thus split the original model into two $tioco$-conform partial models, where the first one captures the different variants of locking, unlocking, closing and opening the doors, up to the first arming transition. The second one only contains one direct path to the armed state, but covers the rest of the system. Both partial models are illustrated in Figure 5.5. Most of the branching is kept in the first smaller system, and the main functionality in the second and larger system.

The runtimes for the determinization of the partial models, performed with the on-the-fly algorithm, can be seen in Row 2 and 4 of Table 5.9. The approach was very fast, and the times for the individual mutants were very similar. For the first partial model the determinized mutants consisted of a mean of 242 mutants, with a minimum of 63 locations and a maximum of 1014. For the second partial model, the determinized mutants consisted of a mean of 44 locations, a minimum of 13 locations and a maximum of 148. The variations manly depended on whether the mutation added early branching to the determinized

Depth	Number of locations				Runtime (sec.)			
	unfolded	std. det.	new det.	on-the-fly	ϵ-removal	std. det.	new det.	on-the-fly
2	8	8	8	8	0.108	0.2	0.1	0.0
5	153	139	83	81	0.4	1.0	0.8	0.2
8	2,062	1,973	757	739	4.1	129.0	11.6	0.9
12	78,847	-	14,009	13,545	10,592.3	-	4,832.1	10.2

Table 5.8: Runtime and number of locations for the Car Alarm System, modified by adding a silent transition causing a 0-2 seconds delay.

Model	Depth	# Mutants	Test-Case Generation						
			Mean	Max	Min	Q_1	Q_2	Q_3	total
Det. Partial 1	8	220	0.15	0.53	0.10	0.13	0.14	0.15	32.6
TCG Partial 1	8	-	6.86	55.94	0.21	2.76	5.53	7.62	$1,406.87$
Det. Partial 2	12	1263	0.11	0.31	0.1	0.11	0.11	0.12	143.02
TCG Partial 2	12	-	1.04	3.09	0.08	0.46	1.20	1.46	$1,223.17$

Table 5.9: Computation time for the determinization and conformance checks on the partial models of the *non-deterministic* version of the car alarm system. All times are given in seconds.

tree, or removed branching. For instance a mutation changing the source location of a transition to the initial location increases the size of the mutants.

We then performed the model-based mutation testing on the two models, with all mutation operators

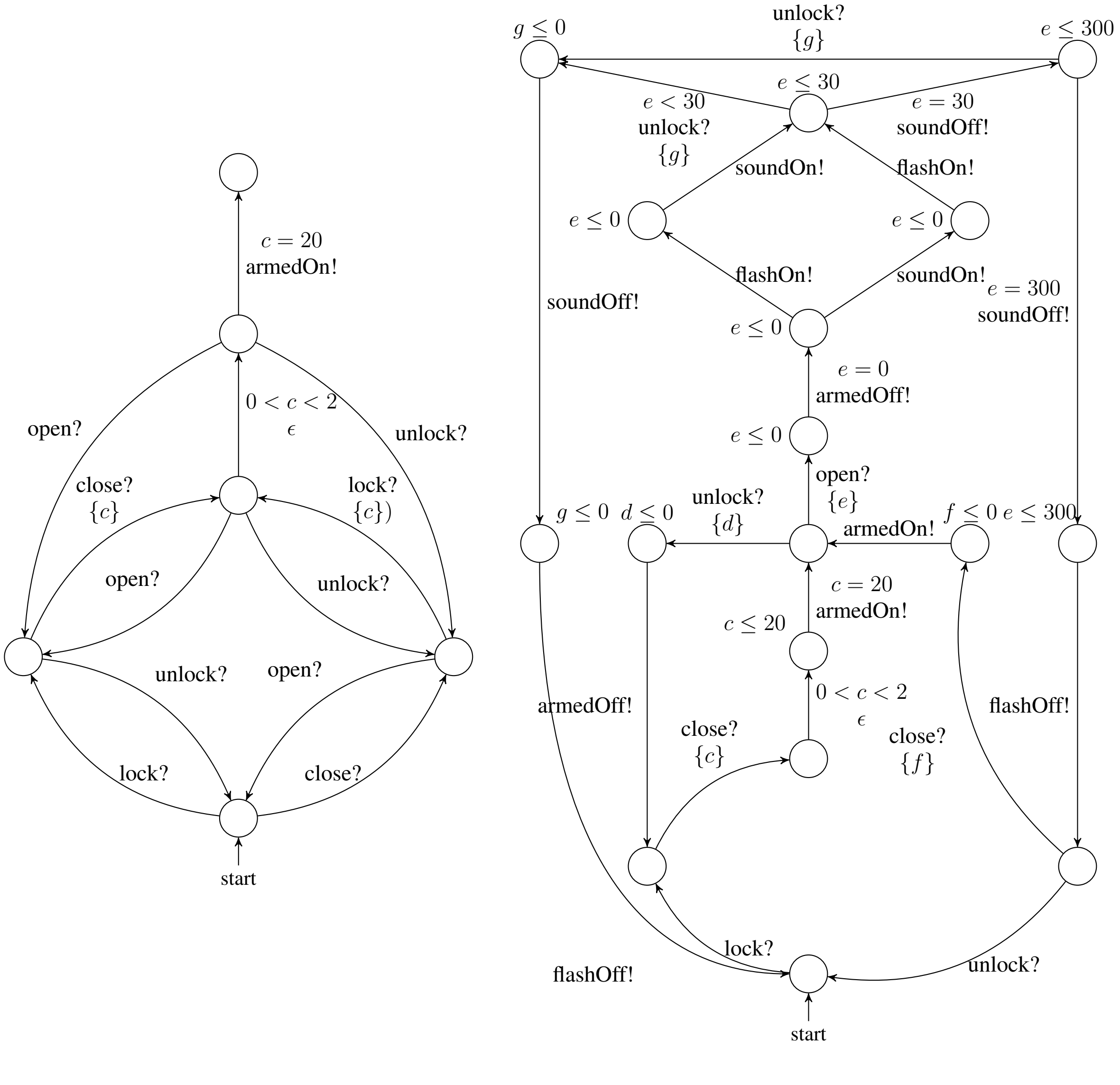


Figure 5.5: Partial models of the car alarm system with silent transitions.

described in Section 3.1. The results of applying the approaches to these models are illustrated in Row 1 and 3 of Table 5.9, including the mean time, the maximal and minimal time and he quartiles Q_1, Q_2 and Q_3. The complete runtime for the first partial model was 23.45 minutes for 220 mutants on Depth 8. The approach created 41 test cases and classified 179 mutants as equivalent. For the second partial model, the bounded model checking took 20.38 minutes for $1,263$ mutants on Depth 12. This time, 637 test cases were generated and 626 mutants were classified as equivalent.

The runtime of the test-case generation exceeds the runtime of the determinization significantly, identifying the test-case generation as the bottleneck of the approach. The high runtime of the first partial model is party caused by some statistical outliers, where the check almost takes a minute, but also the median values (expressed by Q_3) is rather high, indicating that the check took more than 7 seconds for at least half of the mutants. The runtime of the second model is caused by the sheer number of mutants, while the conformance check for the individual mutants only took an average of 1.04 seconds.

For the first model, 95% of the time was spent on equivalent mutants and for the second model it was 62% of the time. These times indicate, that reworking the mutation operators to produce fewer equivalent mutants would be a valuable future task.

5.3.3 Adjustable Speed Limiter.

This section will present our results on the non-deterministic speed limiter. We produced 342 mutants from its original automaton, using the mutation operators described in Section 3.1. 40 of these mutants contained loops of silent transitions and were neglected from the following process. The remaining 302 mutants were unfolded to depth ten, together with the original specification. Then, the silent transition removal and the determinization were performed on all of them, using the *on-the-fly* algorithm. The on-the-fly algorithm took an average of 0.2 seconds per mutant, with a maximum of 1.0 and a minimum of 0.14 seconds. The unfolded determinized mutants contained an average of 620 locations, with a minimum of 337 and a maximum of 1065 locations. The correct specification contained 606 locations. The different amount of locations is caused by the fact, that the mutations may change the branching of the tree and they may change the amount of non-determinism (and thus the amount of locations in the trees that can be merged) of the mutants.

Then we started the test-case generation, which means applying *tioco*- conformance checks between the determinized mutants and the determinized specification. The checks took an average of 63.3 seconds, with a minimum of 0.2 and a maximum of 201.9 seconds. In total, the test-case generation took 5.4 hours, and produced 128 test cases. All runtimes are summarized in Table 5.10, including the quartiles (Q_1, Q_2, Q_3). Again, the equivalent mutants took a high percentage of the total time, with 74%. This confirms the need to select suitable mutation operators for each case study.

The results show that the determinization only takes a fraction of the time needed for test-case generation. However, due to the increased amount of locations and transitions in the explicit unfolding, the runtime of the test-case generation was significantly increased, compared to being executed on a deterministic model of equivalent size that was not unfolded. The most efficient way to combine determinizing and test-case generation would most likely be to integrate the test-case generation into the on-the-fly algorithm, which will be discussed in the future work section.

	Depth	Mean	Min	Max	Q_1	Q_2	Q_3	Total
Determinization	10	0.2	0.14	1.0	0.16	0.18	0.20	60.4
Test-case generation	10	63.3	0.17	201.9	17.1	76.7	94.2	5.4h

Table 5.10: Runtime of determinization and test-case generation for the speed limiter. All numbers are given in seconds, unless otherwise noted.

6 Additional Contributions

Parts of this chapter are based on our publications at SAFECOMP 2014 [10], AMOST 2015 [15] and a Festschrift paper for Frank de Boer [18].

In the previous chapters we saw how timed automata can be used for model-based mutation testing, and how the models can be determinized, to enable the testing from non-deterministic models. Those were the two core topics of Part I. In this chapter, we will present three smaller topics, that slightly expand the capabilities of the test-case generation. The first presented topic in Section 6.1 will show how model mutants can be used to debug faulty implementations, by giving the developer a hint where the fault might be located, and how it looks like. In the second section (Section 6.2) we present how pruning can be used to generate smaller partial models, that still conform to the original and reduce the state space. These partial models enable the efficient processing of models that would otherwise be too complex. In the last section of this chapter, Section 6.3, we will show how to translate timed automata to timed action systems, and compare the presented test-case generation approach with a symbolic one, that runs on timed action systems.

6.1 Debugging via Mutations

Testing and debugging are both important tasks of the development process in the automotive industry. In Chapter 3, we already motivated and discussed the use of model mutations for test-case generation. This section will give a short overview how a very similar technique, reusing many of the already available operations, can also be used for debugging, that is, for the tracing from a failing test case to the actual fault in the system.

Consider a set of very long regression tests, where at some point during execution at least one test case fails. In that case, either the test case or the test setup could be faulty or the implementation can be incorrect. In this section, we will consider the second case. Now the failing test case must not necessarily lead directly to the fault, and thus does not provide many details on its location. Especially in larger products where several designers are working on the SUT, it is important to identify the erroneous part as fast as possible. Therefor the verification engineer has to go through the design and probe outputs / signals / variables related to the faulty test case to judge their correctness. This debugging process is a difficult and time consuming task which in practice is up to now mostly done manually. Within this section we propose a method for speeding up the debugging process by using model-mutants to provide a higher degree of automation.

First, we will discuss some related work, as an introduction to the topic: model-based software debugging (MBSD) [147, 128] is an automated debugging approach with the goal of identifying model components that might be responsible for faulty behavior. MBSD relies on a set of test cases that specify the correct behavior and one or more models that reflect incorrect behavior. Usually the models are divided into a set of components, e.g. the set of code statements. Then the goal of MBSD is to find minimal sets of components (called ”diagnoses”) that, if assumed to be faulty, explain the fault in the implementation. Several different model notations have already been used for model based debugging [127], using formal textual specifications for the models, usually created automatically from the source code. The most commonly used models are Dependency-based Models, Value-based Models and Abstraction-based Models. Other common approaches rely on satisfiability checking and worst-case analysis of several different models [127].

Papadakis and Le Traon [140] used mutants for mutation-based fault-localization, where they show that by investigating the locations indicated by the mutants, they are able to detect 90% of all faults, by investigating only 10% of the source code.

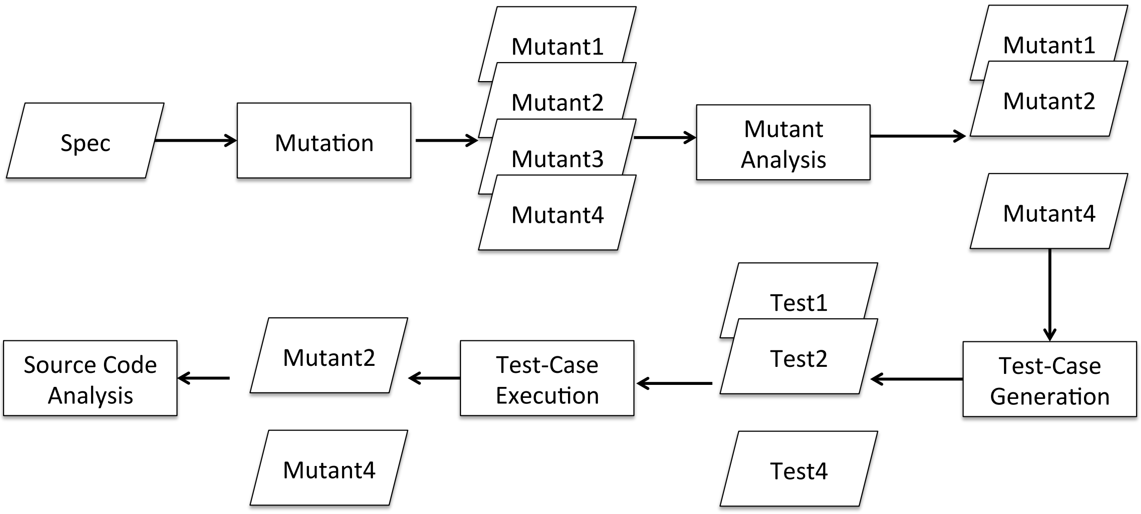


Figure 6.1: Workflow of the model-based debugging approach.

Wotawa [176] introduced mutation debugging, using code mutations as possible diagnoses for faulty implementations. Model-based debugging is used to determine possibly faulty components and the mutation algorithm is only executed on these candidates, to speed up the process.

Nica et al. [132] propose a method for combining debugging, testing and mutants to reduce the set of possible fault candidates. Contrary to our work, they use white-box methods: by mutating the faulty code, they try to find mutants behaving correctly, while we mutate the correct model, trying to find mutants that show the same faulty behavior as the implementation.

6.1.1 Model-Based Mutation Debugging

Model-based mutation debugging (MBMD) starts with a situation that is very common in model-based testing: we are given a specification model (assumed to be correct), a faulty implementation and a random test case that conforms to the model, but fails on the implementation. As already mentioned, if the test case is not minimal, it does not give a lot of feedback on which part of the implementation is faulty, as the incorrect output might be triggered by a fault that was invoked much earlier in the test execution.

Via model mutation, as described in Section 3.1 we can create model mutants representing possible implementation faults. In different filtering steps we can select a small subset of those mutants showing the same faulty behavior as the SUT. These mutants are therefore likely to represent the implemented fault and can be seen as ”mutant diagnoses” for the faulty implementation. The technique relies mostly on operations that were already implemented for the testing approach.

Our approach consists of several steps, each of which will be explained in detail in the next sections. The basic concept behind the approach works as illustrated in Figure 6.1 and is described in the following. The whole procedure is done automatically by our framework. Only the final step, the analysis of the source code, steered by the final mutants, needs to be done manually.

- **Mutant Generation:** First, we create a set of all model mutants our framework supports (Mutant 1, Mutant 2, Mutant 3, Mutant 4 in Figure 6.1). Details on the different supported mutation operators can be found in Section 3.1. Note that we generated fewer mutants, by only applying each mutation operator once per transition/location. Thus, instead of e.g. changing the target location of a transition once for each other location in the automaton and creating multiple mutants for it, we only changed it to one randomly chosen location.

- **Mutation Analysis:** Next, the random test case can be compared to the mutants. If a mutant shows any faulty behavior along the path of the test case, the mutant is said to be killed by the test case. The killed mutants are stored for the next step. All mutants that are not killed are disregarded (Mutant 3 in Figure 6.1), because either their mutations did not lie along the path of the initial test case or did not introduce any faults. The mutation analysis is implemented as a language inclusion check of timed traces (see Section 3.3.2), where the test case is seen as the specification, and we check whether the mutants conform to it. This is possible because our abstract test cases are timed automata traces in sequential form, that can be seen as partial models of the complete specification.
- **Test-Case Generation:** Then, we use our model-based mutation testing technique as described in Chapter 3 for creating minimal test cases (Test 1, Test 2, Test 4) for the selected mutants (Mutant 1, Mutant 2, Mutant 4). The test cases reflect the shortest input/output sequence leading to the faulty output of the mutants.
- **Test-Case Execution:** Next, by executing the test cases on the faulty implementation, we can identify the subset of test cases (and their corresponding mutants) that still are able to find the bug. Some of the test cases cover the bug, but contain several unnecessary steps afterwards. By discarding these test cases and their corresponding mutants, an even smaller set of test cases (Test 2, Test 4) and mutants (Mutant 2, Mutant 4) can be achieved.
- **Source Code Analysis:** Finally, the remaining subset of mutants consists of those mutants that reflect the behavior of the faulty implementation the best. Each mutant reflects a specific implementation fault at a specific location. By examining these code fragments, the location of the bug can usually be traced easily.

We will illustrate the approach on the same car alarm system, that was already introduced in Section 5.1.1. For convenience, we show it is specification once more in Figure 6.2(a). Additionally to the model, we have a Java implementation, a tool to generate random test cases and our model-based mutation test case generation tool MoMuT::TA. Now let us assume a fault in our implementation, skipping the effect of the *unlock?* signal after the alarm went silent. The corresponding model mutant mimicking this implementation fault can be seen in Figure 6.2(b). Also assume the untimed abstract test case TC_1[6]: *close? - lock? - armedOn! - open? - armedOff! - flashOn! - soundOn! - soundOff! - soundOff!- flashOff! - unlock? - lock? - close? - armedOn!* to return the verdict **fail** after this sequence. Now of course, if it was a real random test case, it could be much longer and the trace would not lead straight towards the fail. But already in this simple version, it is hard to trace the exact location of the fault, that could have been introduced anywhere along the path.

Our method provides mutant diagnoses for the bug which can be created without access to the source code, are simple to understand, illustrated as UPPAAL models and give information about both the possible locations and the possible types of fault.

Example 6.1. Applying MBMD to the CAS:

- Mutant Generation: we produce the whole set of model mutants. For the CAS the total number of mutants is 296.
- Mutation Analysis: we filter out all mutants that conform to the test case and keep only those that are killed. E.g. all mutations on the *unlock?* signal while the alarm is still active are not within the scope of the test case.
- Test-case Generation: we can produce minimal test cases for each remaining mutant, leading straight to the fault. A mutation of the *armedOn!* signal might for example produce the test case TC_2 *lock? - close? - armedOn!* to make sure that the implementation fault corresponding to this specific mutation would be detected in the implementation.

[6] For presentation purposes, the test case does not include any timing information.

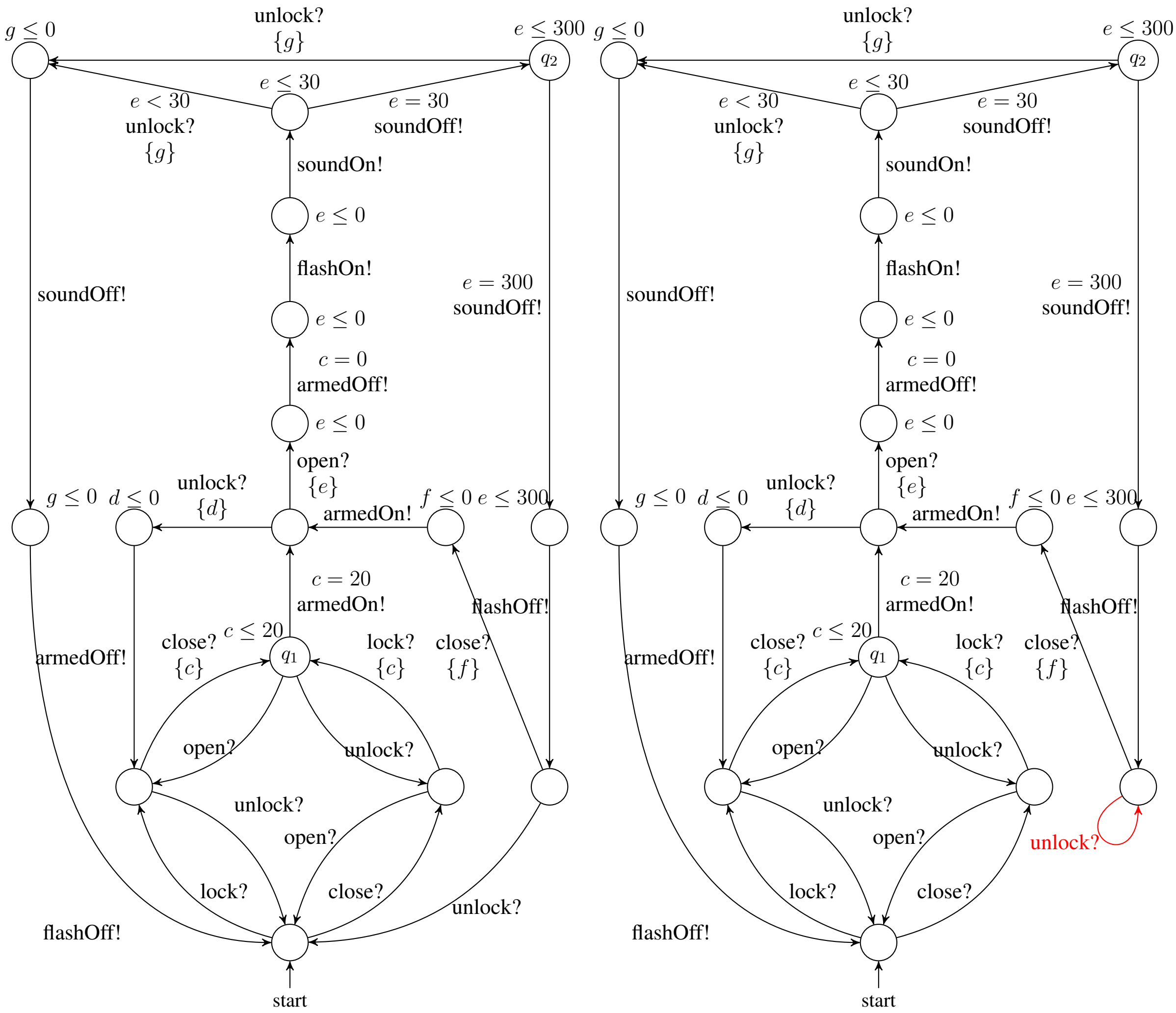


Figure 6.2: Car alarm system: correct TAIO specification (a) and a mutant (b).

- Test-case Execution: after executing the new test cases on the faulty implementation, mutants with test cases that cannot reach the implementation fault are filtered out. TC_2 does not reach the fault, therefore it would be filtered out, as well as its corresponding mutant. Only the shortest test cases and their mutants pass this selection step and are presented to the user as the final set of mutant diagnoses.
- Source Code Analysis: For this specific fault, only two of our mutants remain at the end. Both represent implementation faults of the *unlock?* signal, deactivating its functionality after the alarm went silent. Both lead to the same faulty implementation statement. Figure 6.2 (b) shows one of them. □

6.1.2 Experimental Results

To validate our approach we conducted several experiments on the Java implementation of the car alarm system and the 38 faulty versions, that we presented in Section 5.2. As mentioned, the implementation consists of four public methods, *open*, *close*, *lock* and *unlock*, and two internal methods, *SetState* and the constructor. Elapse of time is simulated with a *tick* method. The faulty implementations were generated with the mutation tool μJava. Since none of the automatically generated faulty implementations contained any timing errors, we additionally created six of those: we generated two faulty implementations

Table 6.1: Possible fault models derived by the remaining mutant diagnoses.

Model Mutation	Mutated Transitions	Corresponding Implementation Fault
Invert Reset	*lock?*	A wrong clock reset during the *lock?* signal.
Self Loop	*close?/lock?*	The *close?/lock?* signal has no effect.
Sink Location	***close?* / *lock?***	**The *close?/lock?* signal leads to a quiescent state.**
Change invariant	-	The *armedOn!* signal is delayed longer than allowed by the specification.
Change source	*close?/lock?*	The *close?/lock?* signal is enabled in a wrong internal state & disabled in the right one.
Change target	*close?/lock?*	The *close?/lock?* signal leads to a wrong internal state.
Negate Guard	*close?/lock?*	The *close?/lock?* signal is disabled.

for each of the three signals *armedOn*, *flashOff* and *soundOff*, one where the signal is delayed and one where the signal is triggered too early.

We applied our debugging method to each of the faulty implementations in a separate experiment: in each experiment, we used the specification model shown in Figure 6.2(a), one of the faulty implementations and a random test case of length 50, generated from the model by our tool MoMuT::TA. If the random test case passed on the faulty implementation, new test cases were generated until one failed on it. All experiments used the same model mutants, which were produced from the specification model by our tool chain. The total number of timed automata model mutants for the CAS is 296.

This subsection is split into three parts: The first two will show two of the experiments in detail. They represent the two most demonstrative special cases, a mutation that can be reached from the initial state, and a mutation that needs ten preceding signals to be reached. Then in the third part, we will give an overview on how well our method performed, presenting the average values of the 44 experiments.

Experiment 1. The first experiment was started with a random test case of length 50. The mutated code of the faulty implementation is shown in Listing 6.1: the introduced implementation fault (negating the state variable in Line 10) causes the *close?* signal to lead to an incorrect internal state that can never be left. The bug only occurs if *close?* is triggered from the initial state.

By doing the tioco-conformance check between the model mutants and the test case, the number of possible mutants was already reduced to 108. Hence, 188 mutants were disregarded either because they

```
1  public static final int OpenAndUnlocked = 1;
2  public static final int ClosedAndUnlocked = 2;
3  public static final int OpenAndLocked = 3;
4  public static final int ClosedAndLocked = 4;
5  public static final int SilentAndOpen = 5;
6  ...
7  public void Close() {
8    switch (m_state) {
9      case CarAlarmSystemState.OpenAndUnlocked :
10       SetState(-CarAlarmSystemState.ClosedAndUnlocked);
11       break;
12     ...
```

Listing 6.1: Code mutation of the *close?* signal leading to a wrong internal state, by negating the state variable in the initial state.

```
1 public void Unlock() {
2         switch (m_state) {
3         case -CarAlarmSystemState.SilentAndOpen :
4             SetState(CarAlarmSystemState.OpenAndUnlocked);
5             break;
6          ...
```

Listing 6.2: Code mutation of the *unlock?* signal. The switch condition cannot evaluate to true, because of the incorrect negation.

were equivalent to the specification, or because the mutation was not covered by the random test case.

Executing MoMuT::TA on the remaining mutants took 724 seconds and produced the corresponding set of test cases. A total of 51 out of the 108 test cases were able to kill the faulty implementation. 17 of these test cases are minimal and all of the minimal test cases are identical, consisting of the trace *close? - lock? - armedOn!*. All test cases longer than this trace and their mutants are disregarded.

Observing that all test cases contain the trace *close? - lock? - armedOn!* and none of them contains *lock? - close? - armedOn!* lets us conclude two things: The fault is not located at the *armedOn!* signal and it is state dependent. The fault is either located in the *close?* signal, when triggered from the initial state, or in the *lock?* signal, when triggered from the *closed* and *unlocked* state.

Since there is no possible output between the *close?* and the *lock?* signal, the test cases cannot provide more information on the location of the bug.

However, using this information, one can discard all mutants with mutations in the *armedOn!* signal, further reducing the 17 mutants to 11. This is however the first step that requires manual input, while the execution so far can be done automatically. Table 6.1 presents the implementation faults represented by the remaining mutant diagnoses. The bold row shows the model mutant representing the actual implementation fault, which could easily be found with this information.

Experiment 2. Part of the second experiment has already been discussed in the introduction of this section, yet here we present the full results. The exact fault is shown in Listing 6.2. The switch condition for the *silent and open* state in the mutated *unlock* signal (Line 3) has been negated and can never evaluate to true, hence the unlock method has no effect after the alarm went silent.

Our initial test case was produced randomly with a length of 50 and is able to kill our faulty implementation. The tioco - conformance check between the model mutants and the test case reduced the total amount of mutants to 127, taking 742 seconds. Hence, 169 mutants were disregarded because the test case did not cover any unspecified output on them.

In the next step, we produced the corresponding test cases with our model-based mutation testing technique, obtaining 127 test cases in 68 seconds. Due to the fact that the mutation is not easy to reach, only 2 of the test cases were able to kill the faulty implementation, therefore only two mutant diagnoses remained in the final set. One of them can be seen in Figure 6.2(b). It was produced with the change target mutation operator, which created a ”self loop”for the *unlock?* transition leaving the *silent and open* state. Such loops mimic the behavior of implementation faults that completely disable the functionality of a signal. The second mutant diagnosis remaining was created via the ”Negate Guard” mutation operator, disabling the guard of the *unlock?* signal. Both diagnoses are valid explanations for the faulty behavior.

Average results. Applying the whole procedure to a faulty implementation took an average of 835 seconds. The final set of mutant diagnoses contained an average of 13 mutants. The reason for this high value is that in black box methods at least one observable has to be reached before a difference

Table 6.2: Characteristics of the generated mutant diagnoses and minimal test cases.

# Faulty impls.	9	1	1	3	1	4	1	2	3	4	2	1	5	2	5	**Avg.**
# Mutant diagnoses	30	29	28	17	16	13	10	9	8	7	6	5	3	2	1	**12.65**
Average minimal test case length	3	3	3	3	3	3	10	9	8	6.5	4	11	3	13	5	**4.95**

in behavior can be detected. Consequently, the minimal length of a test case for the CAS is three and most implementation faults that lie within these three steps show the same behavior with respect to tioco. Therefore, all model mutations that lie within these steps are seen as possible explanations for the faults. Faults that are harder to reach and therefore usually harder to detect, can be identified far better by our approach.

An average of 14 mutants were selected as diagnoses for the timed faults of the six manually created faulty implementations. Several of these diagnoses contained mutations of the (time dependent) guard of transitions and mutations on time invariants. This shows that MBMD also supports the debugging of timing faults. Table 6.2 shows the relation between the faulty implementations and the amount of mutant diagnoses produced by them. The nine faulty implementations that produce 30 possible mutant diagnoses all contain a fault introduced within the first three transitions. The last row shows the average length of the minimal test cases per cardinality of the diagnosis set. It highlights the fact that, in general, deeper faults generate fewer possible explanations.

6.2 Pruning of Trees

In Chapter 4 we presented a bounded method to remove silent transitons and determinize timed automata, by unfolding the automata and bounding the length of the traces we investigate. As mentioned, one of the downsides to this technique is an exponential state-space explosion caused by the unfolding. Consider the example presented in Figure 6.3. It is once more a timed automata specification of a car alarm system, however it differs from the previous examples as it is split into four communicating timed automata. The first two automata handle the locks and the doors. If an input is triggered, they pass on an internal signal to the third automaton, that monitors the locks and doors, to arm the system, if the doors are closed and locked for twenty seconds. The last automaton handles the activation and manual or time-triggered deactivation of the alarms, that are triggered if the doors are violently opened while the system is armed. The network contains four inputs (*lock, unlock, close, open*), five internal signals (*locked, unlocked, closed, opened*) that become hidden after building the product, and six output signals (*soundOn, soundOff, flashOn, flashOff, armedOn, armedOff*). Altogether, the example contains only 23 locations. Yet, unfolding its product to the observable depth two already creates 11 locations, without taking into account the locations that can only be reached by traces ending with internal transitions. Unfolding it one step further creates a total of 50 locations. The number of locations grows exponentially, e.g. on Depth 12 it is already higher then three million locations.

At this point, applying our test-case generation is not feasible anymore. In this section, we propose two methods that avoid the generation of the whole tree and use meaningful sub-trees instead.

The first method applies the mutation to the unfolded tree, instead of applying it to the original specification, as done in our initial process. Consequently, the position of the mutation in the tree is known, and the check whether the mutation violates $tioco$ conformance has to be applied only to the sub-tree beneath.

The second method works with heuristic-based pruning of the tree. Due to the fact that we are using $tioco$, which supports partial models, pruning away transitions with input labels leads to legal partial models. The test cases that can be produced via pruning the system are more adaptive than the tests we generated in our previous test-case generation approach, as they still contain every trace that can be invoked by the remaining inputs and do not lead to $inconclusive$ verdicts if the execution takes a different path than intended by our tool.

6.2.1 Mutation on the Tree

Originally, it was planned to mutate the original specification before unfolding, to produce fewer mutants and avoid the state-space explosion of the unfolding during the mutation. The whole test-case generation process is illustrated in Figure 6.4 (a). It is applied on N mutants, producing n tests, where $N \geq n$, due to possibly equivalent mutants.

However, the unfolded structure can also be utilized to improve the mutation and language inclusion process: the unfolding already traverses the whole state space that is needed for the test-case generation. By mutating the unfolded specification, this information can be used: as the mutations are introduced on purpose and systematically, their position in the unfolded state-space is known. The only thing that needs to be done is the check whether a mutation invokes a violation of the $tioco$-conformance relation in the sub-tree beneath it, or not. Thus, the language inclusion check does not need to start at the root of the tree, but should rather start at the mutation and explore only the subtree beneath the mutated action. This reduces the investigated state space drastically. The updated process can be seen in Figure 6.4 (b). Note that this approach produces M mutants and m tests, where $M \geq m$ and $M > N$ since the mutation on the bigger unfolding produces more mutants than the mutation on the smaller specification.

In order to apply this approach, two steps need to be executed: First, as the unfolded automata might contain infeasible paths, a reachability-analysis is needed to check whether the mutated location or transition can actually be reached from the initial location. If the path from the initial state to the mutated location is in the form of a tree, i.e. every location along the path has exactly one incoming transition, the reachability is comparatively easy and only the guards and clock resets along the trace have to be checked for satisfiability. The constraints that are created on the clocks need to be stored, so they can be attached to the initial state of the tioco check. If there are several traces leading to the mutated state, there is not only one constraint, but one per different trace. As only one of them has to be

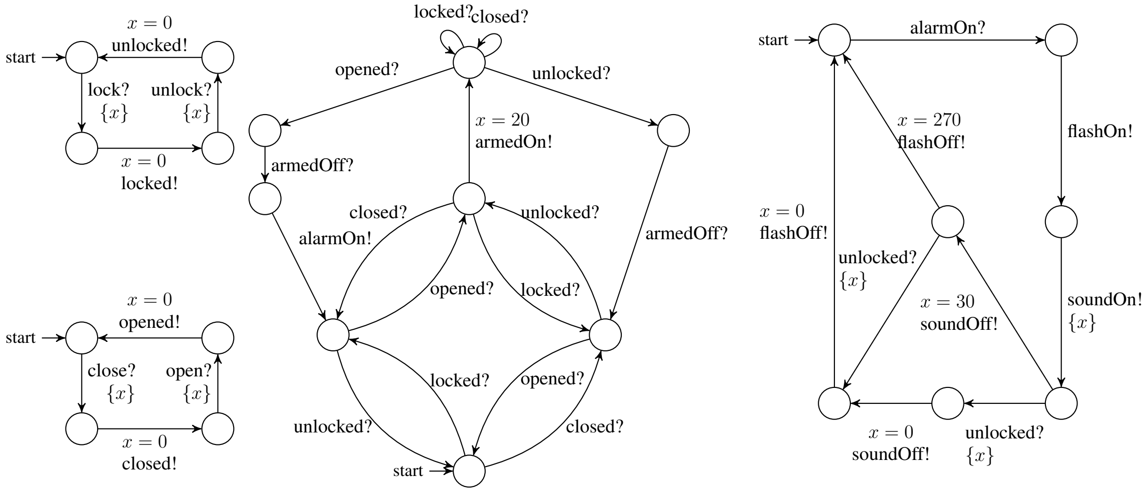
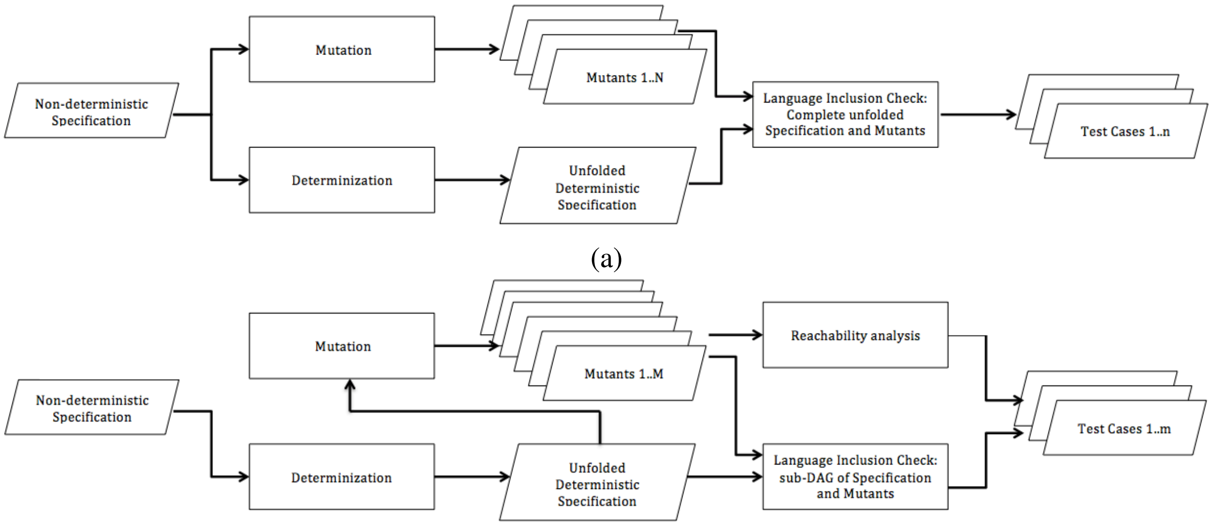


Figure 6.3: A network of four automata, specifying a car alarm system: The first two automata handle the locks and doors, the third triggers the arming of the system and the fourth handles the alarms.


Non-deterministic Specification
Mutation
Determinization
Mutants 1..N
Unfolded Deterministic Specification
Language Inclusion Check: Complete unfolded Specification and Mutants
Test Cases 1..n
Non-deterministic Specification
Mutation
Determinization
Mutants 1..M
Unfolded Deterministic Specification
Reachability analysis
Language Inclusion Check: sub-DAG of Specification and Mutants
Test Cases 1..m

(a)

(b)

Figure 6.4: Our test-case generation process: (a) illustrates the original process (b) shows the updated process, where the language inclusion is only applied to a subtree of the specification and the mutants.

satisfiable for the mutated state to be reachable, a disjunction of all these constraints is stored.

After finishing the reachability analysis, the tree can be pruned, leaving only the subgraph below the mutation. Then the language inclusion check can be applied, with the mutated location as initial location, to see if the mutation propagates to a real failure. The only change to the classical check is that the clocks are not set to zero at the initial location, but are defined by the constraints calculated in the last step.

If a counterexample is found, the test-case generator merges the trace(s) calculated in the reachability check with the counter-example found by the language inclusion, to gain a time adaptive test case from the initial state to the $tioco$-violation.

This check naturally enables us to reach far higher depths in the k-bounded language inclusion of the trees than could be achieved otherwise, as the exponential growth of the complexity only starts after the mutation.

6.2.2 Pruning the Inputs

Partial specifications are valid resources for test-case generation, as long as the partial models still conform to the complete specification. The conformance relation $tioco$ allows the underspecification of inputs, thus by pruning inputs in the tree, the $tioco$ conformance is not violated, while removing any output transitions would. There are several possibilities for the pruning approach:

- Prune according to a manually predefined *test purpose*.
- At each depth, pick a subset N of all inputs, either randomly or according to some predefined distribution, and prune every input not contained in N.
- At each depth, only allow *exactly one* random controllable and prune the rest.

Note that these pruning options can already be applied during the determinization, thus the pruning does not only decrease the complexity of the test-case generation, but can already increase the efficiency of the unrolling and determinizing. In the following, we want to present these approaches in detail:

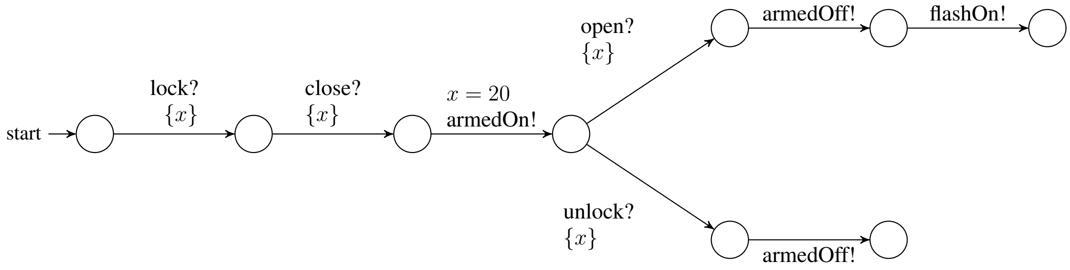


Figure 6.5: The tree of the car alarm system pruned according to the test purpose $\{lock\}$, $\{close\}$, $\{\}$, $\{open, unlock\}$, $\{\}$, $\{\}$.

Test purpose. Our test purposes we have in mind are defined as sets of inputs for each depth, so that at each step only the defined inputs are explored. The test purposes have to be defined by the test engineer, which requires some knowledge about the system, but ensures that the tree only covers the parts relevant for the user. For the presented car alarm system, a test engineer might want to avoid those parts of the tree that start with alternating $locking$ and $unlocking$ of the doors. A well chosen test purpose to avoid this is $\{lock\}, \{close\}, \{\}, \{open, unlock\}, \{\}, \{\}$. This prunes the tree to locking and closing the doors in the first two steps, and avoids any inputs in the third step (thus the empty set). Thus, in the third step only outputs are received, and the tree only covers the branch that arms the alarm system. Opening the door in the fourth step will cause the alarms to start and unlocking it will trigger the transition for unarming the system, so both important branches of the tree are covered. The pruned car alarm system can be seen in Figure 6.5.

The empty sets at the end of the test purpose are used to complete the test case with all outputs that are immediately triggered after the test purpose. Due to these empty sets, in the final steps no new input is triggered, but all outputs are still captured. Contrary, if the test purpose should only be used to prune the first few steps of the tree, and the unfolding should be continued afterwards, it suffices to add sets with all inputs, until the desired depth is reached.

Automated picking of inputs. Picking a set of inputs N for each depth, either per randomization or according to some distribution, is very similar to the manual approach in the previous paragraph. The main advantage is that no knowledge about the specification is needed and the partial model can be created purely automatically.

The random approach of selecting the enabled inputs needs the least effort, even though it needs to ensure in some way, that the chosen inputs are actually enabled in some parts of the tree in the current depth. Choosing the inputs via statistical measures helps steering the partial model in the right direction. If locking and closing the doors has a higher priority than opening and unlocking, the probability for arming the system in the selected partial model is very high.

Alternatively, the inputs can also be chosen for each location in the tree individually, instead of using the same inputs at all locations of each depth. This would facilitate specifying the priority of the inputs in the specification, dependent on the current location. Thus, for instance, the priority of $lock$ and $close$ might be higher in the first few locations, and might decrease as the inner parts of the model are explored.

One input per depth. Choosing exactly one input per depth is a special case of the approach described in the previous paragraph, where $|N|$ is set to one. Like in the last paragraph, the selection can be done randomly or steered by heuristics. The partial models constructed by this approach are in fact already adaptive test cases [99]: they provide one fixed sequence of inputs, and contain every branching caused by the outputs. Given a test driver that can handle adaptive tests, theses tests could immediately be executed on a SUT.

While these partial models cannot be used for generating a test suite via model-based mutation testing, as they already are in the form of test cases, the model-mutations still can be used to assess the quality of these random tests, i.e. to check how many mutants are killed by the adaptive test cases. The mutation analysis could also be used as a stopping criterion for the test-case generation, indicating when enough test cases have been generated.

Note that we distinguish between two types of non-determinism: non-deterministic automata, where two transitions with the same label are enabled at the same time, and non-deterministic systems, where the system can chose non-deterministically between different outputs. While our test-case generation approach is not able to handle the first case, the second one did not cause problems. However, our previous test cases only contained one specific trace through the system, and hence unexpected outputs led the test-case execution to an $inconclusive$ verdict. The test cases we generate by pruning the tree are fully adaptive and contain every trace that can be invoked by the chosen inputs.

6.3 Translation to Timed Action Systems

In this section, we show how to translate timed automata into *timed action systems*, an extended version of Back and Kurkio-Suonios [37] action systems with support for time. We then show how to apply a symbolic conformance check to timed action systems, according to the *symbolic timed input-output conformance (stioco)*. The approach was developed together with Martin Tappler, who developed and implemented the conformance check and did all the experiments on the timed action systems. In this section we will only give an excerpt, the full documentation can be found in [18].

6.3.1 Timed Action Systems

Action systems were introduced for modeling distributed systems. In more recent work, they have been used as a modeling formalism for mutation-based test-case generation for reactive systems [14, 4]. An event-centered view of action systems has been taken in this context, for deriving test cases and for checking of *ioco* conformance between action systems. More concretely, for model-based mutation testing each action is assigned a label and an action type, which identifies the action as being an output, input or internal action.

For the definition of timed action systems, we also follow this approach. However, the modeling formalism discussed in the following is more restricted with respect to discrete actions than other variants of the action systems formalism. Nevertheless, we also extend traditional action systems by explicitly accounting for time, which is inspired by timed automata.

In our approach, an action system defines a set of actions and corresponding guarded commands, a set of state variables and an initialisation for these variables. An action defines a set of parameters and has an action type. For each action, the corresponding guarded command defines the conditions in which the action may be executed and the effect of the action execution. The guarded commands may access state variables and the parameters of the corresponding action. There may be several actions same label and if multiple actions share the same label, they must also have the same parameters and action type.

During the execution of action systems, at each step an enabled action is chosen non-deterministically and executed. Through this the state is continuously updated until the execution terminates, when none of the actions is enabled. An action is enabled if the guard of its corresponding guarded command is satisfiable.

In order to enable the modeling of time, we extend action systems by adding clock variables as in timed automata. In between the execution of two discrete actions, the system may wait for certain amounts of time, which increases the values of the clock variables. This act of waiting will also be referred to as delay in the following. To be able to define the conditions for the actual waiting time, we add time invariants to action systems. The time invariant of an action system must hold in all states and consists of several clauses. A clause defines a time constraint which must hold if the state variables satisfy the condition defined by the clause. Finally, guarded commands may define conditions using clocks and may reset clocks.

Several time extensions for action systems have already been proposed: Fidge and Wellings [85] proposed timed action systems, assuming time-consuming actions and discrete time. Westerlund and Plosila [175] proposed action systems based on continuous time, where each action system contains a clock to measure the time since start of the system. Again, time is considered to be consumed by actions, and may not pass between them. Wabenhorst [170] proposes a formalism combining time-consuming actions and an additional wait action executed if none of the other actions are enabled. In contrast to these proposals, we consider actions that take zero time, followed by delays. This keeps our definition of timed action systems very close to timed automata.

Kurki-Suonio [118] proposed a time extension to action systems, using, equal to our approach, zero time actions, but using only one global variable to track time. Each action has a parameter specifying its time of execution. They can only be executed if the global time is smaller or equal to their time of execution. If an action is chosen, it raises the global time to its time of execution. Contrary to this approach, we use invariants instead of deadlines for limiting time progress and we support multiple clocks, allowing for more complex time constraints.

Seceleanu and Seceleanu [151] proposed a new definition of a parallel product for action systems, which can also be applied to continuous timed action systems. However, the scope of the paper focused mainly on the product of multiple action systems, while our approach is currently only applied to single action systems.

6.3.2 Conventions

Generally, we assume the usage of two-sorted logic, where one sort d is defined for discrete data and the other sort t for time-related formulas and terms. We further require that the constant 0_t of sort t and the binary addition $+_t$ for pairs of sort t must be defined. In addition, the relations $\leq, <, =, >\geq$ must be defined for all pairs of sorts d and t, i.e. any comparison between time and data must be possible. Note that in practise, we allow for more sorts in our models, such as user-defined enumeration sorts, but we use a type checker to ensure that only meaningful comparisons are performed.

We will denote the set of terms containing variables from a set X by $Te(X)$ and first-order formulas containing free variables from the same set by $Fr(X)$. The function $free(\varphi)$ maps a formula φ to the set of all free variables in φ.

The set $CC(X, Y)$ denotes the set of clock constraints, with clock variables in X and constraint operands in $Y \cup Te(\emptyset)$. A clock constraint is of the form $x \otimes y$, with $x \in X$, $y \in Y \cup Te(\emptyset)$ and $\otimes \in \{\leq, <, =, >\geq\}$, i.e. it is comparison between a clock variable and a variable or a constant term.

The set of all total functions from A to B shall be denoted by B^A. The substitution of variables shall be denoted by $g[\sigma]$, where σ is a function from variables to terms and g is some formula or term. Hence, the signature of $[\sigma]$ is given by $[\sigma] : Te(X) \cup Fr(X) \rightarrow Te(X) \cup Fr(X)$, where X is a set of variables. The term f_X denotes the domain restriction of a function f to the set X.

Sequences containing $e_1, e_2, \ldots, e_n$ will be denoted by $\langle e_1 \cdot e_2 \cdots e_n \rangle$ and the concatenation of two sequences σ_1 and σ_2 will be denoted by $\sigma_1\hat{}\sigma_2$.

```
clocks[Real]{ c;d;e;f;g }
init{
  location := OpenAndUnlocked;}
invariant{
  if location == ClosedAndLocked then c <= 20;
  ... }
actions{
  !armedOn() if location == ClosedAndLocked and c == 20 then {
    location := Armed; };

  ?open() resets e if location == Armed then {
    location := BeforeAlarm; };
  ... }
```

Figure 6.6: A snippet of the timed action system model of the CAS.

6.3.3 Syntax

In the following, we define the syntax for action systems. For a trace-based definition of their semantics, we refer to our publication [18]. The definitions are inspired by the work of Frantzen et al. [87] and von Styp et al. [168], who use symbolic timed automata. Since symbolic timed automata are similar to regular ones, our version of *stioco* can be seen as an extension of the original definition [168], as we also allow internal actions.

Figure 6.6 illustrates the structure of the concrete syntax of timed action systems and models a part of the car alarm system. It specifies five real-valued clocks, that the initial state of the system shall be `OpenAndUnlocked`, that the system must not wait longer than 20 time units in state `ClosedAndLocked` and defines the actions. The actions are labeled with `armedOn` and `open`. These two events are fully defined through two and three actions respectively. In the following, we present the abstract syntax of timed action systems.

Definition 6.1 (Abstract Syntax of Timed Action Systems)
A timed action system is a tuple $\mathcal{TAS} = \langle \mathcal{V}, \mathcal{I}, \mathcal{C}, \Lambda_I, \Lambda_U, \iota, Inv, A\rangle$, where $\mathcal{V}$ is the set of state variables, $\mathcal{I}$ is the set of parameter variables and $\mathcal{C}$ is the set of clock variables, with $\mathcal{V}$, $\mathcal{I}$, $\mathcal{C}$ being mutually disjoint. $\Lambda = \Lambda_I \cup \Lambda_U$ is the set of action labels, with Λ_I being the set of input action labels and Λ_U being the set of output action labels. The constant $\tau \notin \Lambda$ denotes an internal action and we set $\Lambda_\tau = \Lambda \cup \{\tau\}$. The initialisation of the action system is $\iota \in Te(\emptyset)^{\mathcal{V}}$. Inv is the time invariant of $\mathcal{TAS}$, which is of the form $\bigwedge_i dc_i \to cc_i$, with $dc_i \in Fr(\mathcal{V})$ and $cc_i \in CC(\mathcal{C}, \mathcal{V})$ for all i. The set $A \subseteq \Lambda_\tau \times Fr(\mathcal{V} \cup \mathcal{I}) \times CC(\mathcal{C}, \mathcal{V}) \times Te(\mathcal{V} \cup \mathcal{I})^{\mathcal{V}} \times \mathcal{P}(\mathcal{C})$ is the set of all actions. For $a = (\lambda, g, g_c, up, r) \in A$, λ is called label, g is called guard, g_c is the clock guard, up is the update mapping, defined by assignments in the guarded command and r is a set of clocks, which are reset by executing a.

Before we define semantics for timed action systems, we introduce two requirements and two auxiliary functions. These are similar to the requirements defined for symbolic transition systems by Frantzen et al. [87]. The functions $arity$ and $para$ associate each action with its number of parameters and a tuple containing its parameters respectively.

1. For all actions λ, $para$ maps λ to a tuple of distinct parameter variables and for $(\lambda, g, g_c, up, r) \in A$ it holds that $free(g) \subseteq \mathcal{V} \cup para(\lambda)$ and $up \in Te(\mathcal{V} \cup para(\lambda))^{\mathcal{V}}$.
2. As for τ-edges of timed transition system, we disallow the definition of parameter variables for internal actions of timed action systems, i.e. for all τ-actions, it must hold that $arity(\tau) = 0$.

Example 6.2 (Abstract Syntactical Representation of the CAS). The CAS defined in Figure 6.6 is a timed action system$\langle \mathcal{V}, \mathcal{I}, \mathcal{C}, \Lambda_I, \Lambda_U, \iota, Inv, A\rangle$, where $\mathcal{V} = \{location\}$, $\mathcal{I} = \{\}$, $\mathcal{C} = \{c, d, e, f, g\}$,

$\Lambda_I = \{open, \ldots\}$, $\Lambda_U = \{armedOn, \ldots\}$, $\iota = \{location \mapsto OpenAndUnlocked\}$, $Inv = (location = ClosedAndLocked) \rightarrow c \leq 20 \wedge \ldots$ and $A = \{o, a, \ldots\}$. With actions $o = (open, location = Armed,$ true, $\{location \mapsto BeforeAlarm\}, \{e\})$ and $a = (armedOn, location = ClosedAndLocked,$ $c = 20$,$\{location \mapsto Armed\}, \{\})$. Parts omitted in Figure 6.6 are represented by dots. □

The trace-based semantics must fulfill four requirements: a trace must (1) start with a delay, (2) consist of alternating sequences of discrete actions and delays, and (3) end in a delay. The first two requirements are placed on the semantics in correspondence to the definition of traces by von Styp et al. [168]. Conversely, the third requirement serves to simplify conformance checking while it does not limit generality as zero delays are possible. Additionally, (4) a trace should handle internal actions appropriately: consider the concrete timed trace $ct = \langle 1 \cdot !a \cdot 2 \cdot \tau \cdot 3 \cdot ?b \cdot 0 \rangle$. For checking *tioco* conformance one is only interested in observable traces of the specification [116]. Thus, we would project ct to the set of observable input and output actions, erasing the τ-action and summing up the two consecutive delays: $ct' = \langle 1 \cdot !a \ \cdot \ 5 \cdot ?b \ \cdot \ 0 \rangle$.

In the symbolic setting, we use symbolic traces where constant time delays are replaced by symbolic delay variables. As common in symbolic execution, these symbolic delays are defined via constraints. We distinguish between two kinds of delay variables: observable delays t_i, which are part of the observable trace and unobservable delays $d_{i,j}$ that appear only in constraints. Observable delays are always defined in terms of unobservable delays. For example, the symbolic trace $st = \langle d_1 \cdot !a \cdot d_{2,1} \cdot \tau \cdot d_{2,2} \cdot ?b \cdot d_3 \rangle$ including an unobservable τ-action would be projected to an observable trace $st' = \langle t_1 \cdot !a \cdot t_2 \cdot ?b \cdot t_3 \rangle$ with the constraints $t_1 = d_1$, $t_2 = d_{2,1} + d_{2,2}$ and $t_3 = d_3$. Note that while observing the delay t_2, it is not possible to distinguish between the internal delays $d_{2,1}$ and $d_{2,2}$.

So far, we only considered delays. For the trace-based semantics we need to update the state of variables and clocks along a trace and collect the constraints: discrete and time guards of actions, time invariants and constraints which express that consecutive unobservable delays sum up to observable delays. In addition, it is necessary to keep track of the set of unobservable delays along a trace, because we will hide these via existential quantification for the conformance check.

6.3.4 Conformance Checking.

Since the *stioco* conformance relation for timed action systems is very similar to the definition of *stioco* of von Styp et al. [168], we will not give the full definition, but rather list three important differences:

- We use the semantics discussed above. As unobservable delays along a trace are relevant for conformance, symbolic states and symbolic observations consider these as well. Hence, states and observations are tuples, where one tuple element contains the unobservable delays which have been collected before reaching a symbolic state or before observing some symbolic observation.
- The symbolic observation of delays needs to be adapted as well, i.e. a symbolic counterpart of the $elapse(s)$-function [116] must be defined, which maps a state s to the set of delays, which can be executed without executing an observable action. Hence, a symbolic $elapse(s)$-function could be defined as a trace, which consists of only one delay, executed in state s.
- The original *stioco* definition uses a function Φ, which gives a condition for observing some observation after a given trace σ. To account for internal actions, this function needs to existentially quantify over the sets of unobservable delays collected along σ.

The symbolic conformance check is implemented in the same fashion as the *sioco* conformance check for untimed action systems [23], which is itself inspired by the *ioco* conformance checker used in [6, 4]. More concretely, it performs a bounded depth-first search for *unsafe states*, which are states in which non-conformance may be observed. For this purpose, both mutant and specification are symbolically executed in parallel, such that they synchronise on observable actions, but execute internal actions independently from each other. In order to ensure input-enabledness of the mutant, which is a requirement for *stioco*, we perform an angelic completion for the mutant. Hence, we implicitly add self-loops to states for all non-specified inputs. At each step, a conformance check is performed and if non-conformance is detected, the trace leading to the current state and the satisfiable non-conformance condition are returned.

6.3.5 Encoding Timed Automata using Timed Action Systems

In order to encode a timed automaton as a timed action system, we essentially create a timed action system having the same set of state variables plus one additional state variable representing the current location and having the same set of transitions/actions. The procedure for translating timed automata into timed action systems can be structured as follows:

1. Create a timed action system with a set of state variables corresponding to the automatons locations, and the same clocks and action labels.
2. Create a set of constants Loc, where each constant represents a location in the timed automaton. Define a function rep, which maps locations to their respective constants.
3. Add an additional state variable called $location$, which takes values in Loc, and rename an existing variable with the same name, if such a variable exists. Initialise $location$ with $rep(l_0)$, where l_0 is the initial location of the timed automaton.
4. For each transition of the timed automaton with source location l and target location l':
   (a) Create an action with equivalent guards, clock resets, state updates and label.
   (b) Add $location = rep(l)$ to the guard via conjunction and add $location \mapsto rep(l')$ to the state update.
5. Initialise the time invariant to $\top$, then for each invariant i of a location l: Add the clause $rep(l) = location \rightarrow i$ to the time invariant of the timed action system via conjunction.

Any timed action system that was built according to this structure, can also be encoded as a timed automaton, by reverting the steps above.

6.3.6 Comparison of Symbolic Execution and Bounded Model-Checking

We now want to compare the symbolic conformance check developed by Martin Tappler to the bounded model-checking approach described in Chapter 3 [23]. We apply the checks to different variants of the car alarm system, containing model elements such as silent transitions and data variables, that can be challenging for the conformance checks. In all the experiments we use the following settings: we translated from timed automata to timed action systems as closely as possible: The different models contain the same number of states and transitions and the same sets of clocks and variables. We used eight different mutation operators (similar to those in Section 3.1, excluding the changing of action labels, that would have been problematic to implement for the timed action systems), that were implemented equally for both types of models. However, due to the different modeling styles, the amount of mutants did vary slightly in some cases. All experiments were run on a MacBook Pro with a 2.8 GHz Intel Core i7 and 8 GB RAM (6 GB were reserved for the Java virtual machine) with the reimplementation of MoMuT::TA, only the car alarm system with PIN code was processed with the initial prototype implementation.

Table 6.3: Computation time for the different conformance checks on the deterministic version of the car alarm system.

Depth	Bounded Model Checking				Symbolic Execution			
	Mean	Median	Max	Min	Mean	Median	Max	Min
12	$1.4s$	$1.1s$	$33s$	$0.07s$	$1.7s$	$0.02s$	$38.83s$	$\sim 0s$

Deterministic Car Alarm System. We first investigate the model in Figure 5.1 (a). It is deterministic and has 5 clocks, 16 locations and 25 transitions. The results of applying both approaches are displayed in Table 6.3. The bounded model checking performed slightly faster and at a very constant rate, without many statistical outliers. The symbolic execution, with the median far below the mean value, was very fast for most of the mutants, however there were some that took significantly longer than the rest, and increased the average processing time. The overall runtime of the bounded model checking was 30.0 minutes for $1,320$ mutants, compared to 27.5 minutes for 968 mutants of the symbolic execution.

Non-Deterministic Car Alarm System. The next model was presented in Figure 5.1 and contains a silent transition that non-deterministically delays the 20 seconds timer responsible for arming the system by up to two seconds. This changes the time constraints for the arming of the system and adds non-determinism for the $unlock$ and $open$ transitions leaving the locations. In Section 5.3.2 we already explained that the unfolded determinized model became too large for our test-case generation approach. Thus, it was split into two $tioco$-conform partial models, where the first one captures the different variants of locking, unlocking, closing and opening the doors, up to the first arming transition. The second one only contains one direct path to the armed state, but covers the rest of the system. Both partial models are illustrated in Figure 5.5.

The results of applying the two different approaches to these models are illustrated in Table 6.4. The difference to the runtimes presented in Section 5.3.2 is based on using a different version of Z3, and in the experiments from Section 5.3.2 we reserved more RAM for the virtual machine. Both approaches discussed in this section were run with the same settings. The overall runtime for the first partial model was 32.8 minutes for 220 mutants for applying the bounded model checking and 48.1 seconds for 168 mutants for the symbolic execution. For the second partial model, the bounded model checking took 34.1 minutes for $1,263$ mutants and the symbolic execution only needed 68.1 seconds for 832 mutants.

The ability of the symbolic approach, to process the models without unfolding them first, clearly gives it an advantage here. Not only is it a lot faster on the partial models, it was also able to process the complete model. Additionally, it has on average even been faster than in the deterministic case. There are two main reasons for this behavior. Firstly, three mutants have not been checked for conformance automatically, because they ran into a timeout (ten minutes), and were excluded from the experiments. However, manual inspection revealed that these mutants conform to the specification. Secondly, the introduction of a silent transition led to a much larger portion of nonequivalent mutants. Aichernig et al. showed that *ioco* checking of equivalent mutants takes significantly longer than *ioco* checking of non-

Table 6.4: Computation time for the different conformance checks on the partial models of the *non-deterministic* version of the car alarm system.

Model	Depth	Bounded Model Checking				Symbolic Execution			
		Mean	Median	Max	Min	Mean	Median	Max	Min
Partial 1	8	$9.7s$	$8.0s$	$85.1s$	$0.3s$	$0.28s$	$0.04s$	$16.78s$	$\sim 0s$
Partial 2	12	$1.6s$	$1.63s$	$37.3s$	$0.08s$	$0.08s$	$0.03s$	$2.28s$	$\sim 0s$
Complete	12	x	x	x	x	$0.79s$	$0.06s$	$360.84s$	$\sim 0s$

Table 6.5: Computation time for the different conformance checks on the deterministic version of the car alarm system, augmented by a *PIN code*.

Depth	Bounded Model Checking				Symbolic Execution			
	Mean	Median	Max	Min	Mean	Median	Max	Min
8	$1.46s$	$0.28s$	$59.41s$	$0.12s$	$0.07s$	$0.05s$	$0.82s$	$\sim 0s$
12	$4.12s$	$0.35s$	$35.41s$	$0.13s$	$0.24s$	$0.05s$	$3.67s$	$\sim 0s$

equivalent mutants [6], thus a lower number of equivalent mutants can explain the reduction in average runtime from $1.7s$ to $0.79s$.

Car Alarm System with PIN Code. This final model treats the ability of processing data variables. The *unlock* and *lock* transitions of the car alarm system are augmented by a PIN code. If the code is entered correctly, the system acknowledges it with a new ack-output, and continues as before. If it was entered incorrectly, the system will start the alarms, after a $nack$-output. This model only uses one clock, whereas five clocks were used in the original car alarm system.

The PIN code did not have any negative influence on both approaches, as illustrated in Table 6.5. For the symbolic execution, the mean conformance check time was even reduced. This was most likely caused by the fact that only one clock was used in this model. Furthermore, there were several more mutants, most of which were non-equivalent.

Altogether, the bounded model checking was applied to $1,702$ mutants and needed 41.4 minutes on depth 8 and 116.8 minutes on depth 12. The symbolic execution was faster, needing 143.0 seconds on depth 8 and 460.8 seconds on depth 12 for $1,918$ mutants. For the reported numbers, we restricted the PIN code to three digits. However, we also applied the experiments with higher values (four and five digits), without any negative consequences.

6.3.7 Summary

In this section, we have introduced timed action systems in a fashion as close to timed automata as possible. We showed how to translate timed automata into timed action systems and defined a symbolic trace semantics for them. Using this semantics, we applied a symbolic conformance check based on the *stioco* conformance relation. We then compared bounded model checking and symbolic execution in the context of test-case generation, applied to different models of a car alarm system. The results showed that symbolic execution was able to handle non-determinism better than the bounded model-checking approach, and that data variables did have no negative influence on both approaches.

Part II

Synchronous Systems

Overview

In the second part of this thesis, we investigate synchronous systems in the context of model-based mutation testing and real-time properties.

We first introduce *requirement interfaces*, which we use as a representative for synchronous formalisms. It enables the modeling of different system views as subsets of requirements. It is a state-transition formalism that supports compositional specification of synchronous data-flow systems by means of assume/guarantee rules, that we call contracts. We associate subsets of contracts to requirement identifiers, to facilitate their tracing to the informal requirements from which the specification is derived. These associations can later on be used to generate links between the work products, connecting severals tools.

Instead of modeling the complete behavior of a system at once, each requirement interface is intended to model a single specific view of the SUT. We define the *conjunction* operation that enables combining different views of the SUT. Intuitively, a conjunction of two requirement interfaces is another requirement interface that requires contracts of both interfaces to hold. We assume that the overall specification of the SUT is given as a conjunction of requirement interfaces modeling its different views. Requirement interfaces are inspired by the synchronous interfaces formalism by Henzinger et al. [66], with the difference that we support *hidden* variables in addition to the interface (*input* and *output*) variables and that the requirement identifiers are part of the formal model. The conjunction operator was first defined by Doyen et al. [78] as shared refinement, while Benveniste et al. [46] establishes the link of the conjunction to multiple viewpoint modeling and requirement engineering.

Once the basic definitions of requirement interfaces are given, we formally define *consistency* for requirement interfaces and develop a bounded consistency checking procedure. In addition, we show that falsifying consistency is compositional with respect to conjunction – the conjunction of an inconsistent interface with any other interface remains inconsistent. Next, we develop a requirement-driven TCG and execution procedure from requirement interfaces, with *language inclusion* as the conformance relation. We present a procedure for TCG from a specific SUT view, modeled as a requirement interface, and a *test purpose*. Here, the test purpose is a formal specification of the target state(s) that a test case should cover. Such a test case can be used directly to detect if the implementation by the SUT violates a given requirement, but cannot detect violation of other requirements in the conjunction that composes the system. Next, we extend this procedure by completing such a partial test case with additional constraints from other view models that enable detection of violations of any other requirement.

Then, we develop a tracing procedure that exploits the natural mapping between informal requirements and our formal model. Thus, inconsistent contracts or failing test cases can be traced back to the violated requirements. We believe that such tracing information provides precious maintenance and debugging information to the engineers.

Next, we show how to apply model-based mutation testing, using the technique to automatically generate a set of test purposes. The corresponding test suite is able to provide fault coverage for a specified set of fault models, under the assumption of a deterministic SUT. The approach includes the following steps: first, we define a set of fault models for requirement interfaces. These are applied to all applicable parts of the contracts in the requirement interface, generating a set of mutants. We then check whether the mutated contract introduces any new behavior. This check is encoded as a test purpose, so we can simply pass it to the previously defined test generation. If the mutation introduces new behavior that deviates from the reference model, it will generate a test, otherwise, the test purpose will be unreachable, and the mutant is considered equivalent.

We illustrate the entire workflow of using requirement interfaces for consistency checking, testing and tracing in Figure 6.7, where the test purpose may be produced by model-based mutation testing, or

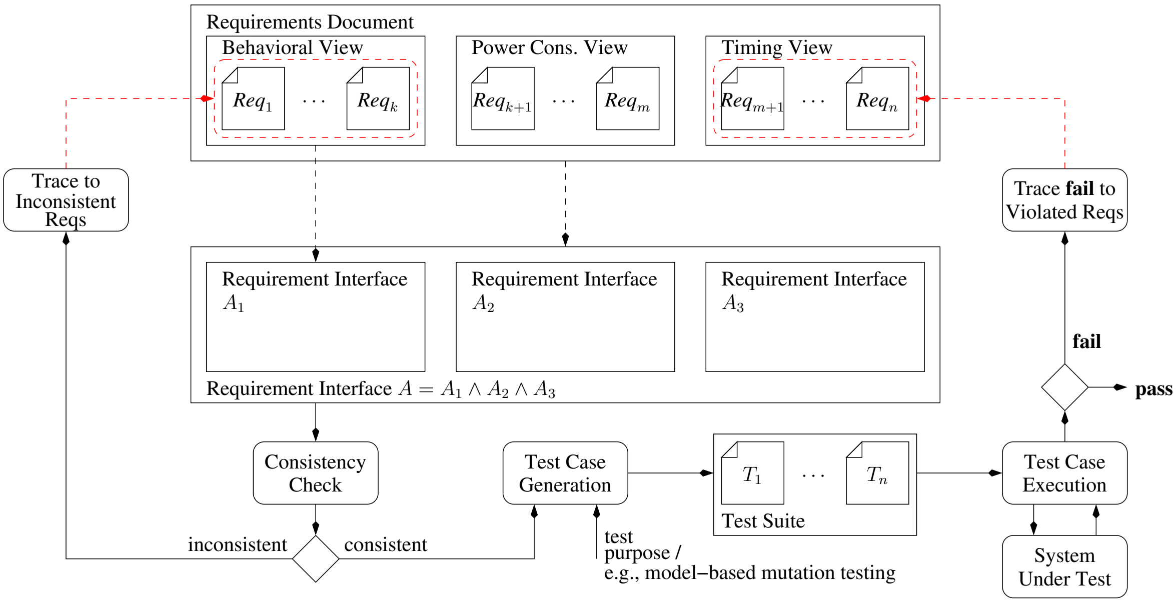

Requirements Document
Behavioral View
Req_1 · · · Req_k
Power Cons. View
Req_{k+1} · · · Req_m
Timing View
Req_{m+1} · · · Req_n
Trace to Inconsistent Reqs
Trace **fail** to Violated Reqs
Requirement Interface A_1
Requirement Interface A_2
Requirement Interface A_3
Requirement Interface $A = A_1 \wedge A_2 \wedge A_3$
fail
pass
Consistency Check
Test Case Generation
T_1 · · · T_n
Test Suite
Test Case Execution
System Under Test
inconsistent
consistent
test purpose / e.g., model–based mutation testing

Figure 6.7: Overview of using requirement interfaces for testing, analysis and tracing.

any arbitrary other technique. The process starts with the requirements document, which in this example shows three different views of the system. These are then formalized into three requirement interfaces, which, if composed together, specify the complete functionality of the system. Then we perform a consistency check, which may be performed on the individual requirement interfaces, or on the conjunction. If they reveal an inconsistency, we need to trace them back to the original requirements (which are directly linked), to see whether the fault lies in the requirements, or was made during the formalization. If the requirement interfaces were consistent, we apply the test-case generation, which may be guided by manually designed test purposes, or by automatically generated ones, like those generated by model-based mutation testing. If the test purposes can be reached, we transform the trace leading there into a test case. The sum of this test cases forms the test suite, which is then executed on the system under test. If all test cases pass, we can consider the system correct, with respect to the test suite and the requirements. If not, we need to examine which requirements were violated, to determine whether the system behaved incorrectly, or the requirements were incorrect.

7 Requirement Interfaces

Parts of this chapter are based on our publication at FMICS 2015 [11].

We introduce *requirement interfaces*, a formalism for the specification of synchronous data-flow systems. Their semantics is given in the form of labeled transition systems (LTS). We define *consistent* interfaces as the ones that admit at least one correct implementation. The *refinement* relation between interfaces is given as *language inclusion*. Finally, we define the *conjunction* of two requirement interfaces as another interface that subsumes all behaviors of both interfaces.

7.1 Syntax

Let X be a set of typed variables. A valuation v over X is a function that assigns to each $x \in X$ a value $v(x)$ of the appropriate type. We denote by $V(X)$ the set of all valuations over X. We denote by $X' = \{x' \mid x \in X\}$ the set obtained by priming each variable in X. Given a valuation $v \in V(X)$ and a predicate φ on X, we denote by $v \models \varphi$ the fact that φ is satisfied under the variable valuation v. Given two valuations $v, v' \in V(X \cup X')$ and a predicate φ on $X \cup X'$, we denote by $(v, v') \models \varphi$ the fact that φ is satisfied by the valuation that assigns to $x \in X$ the value $v(x)$, and to $x' \in X'$ the value $v'(x')$.

Given a subset $Y \subseteq X$ of variables and a valuation $v \in V(X)$, we denote by $\pi(v)[Y]$, the projection of v to Y. We will commonly use the symbol w_Y to denote a valuation projected to the subset $Y \subseteq X$. Given the sets X, $Y_1 \subseteq X$, $Y_2 \subseteq X$, $w_1 \in V(Y_1)$ and $w_2 \in V(Y_2)$, we denote by $w = w_1 \cup w_2$ the valuation $w \in V(Y_1 \cup Y_2)$ such that $\pi(w)[Y_1] = w_1$ and $\pi(w)[Y_2] = w_2$.

Given a set X of variables, we denote by X_I, X_O and X_H three disjoint partitions of X denoting sets of *input*, *output* and *hidden* variables, such that $X = X_I \cup X_O \cup X_H$. We denote by $X_{\text{obs}} = X_I \cup X_O$ the set of *observable* variables and by $X_{\text{ctr}} = X_H \cup X_O$ the set of *controllable* variables. We adopt SUT-centric conventions to naming the roles of variables. A *contract* c on $X \cup X'$, denoted by (φ, ψ), is a pair consisting of an *assumption* predicate φ on $X'_I \cup X$ and a *guarantee* predicate ψ on $X'_{\text{ctr}} \cup X$. A contract $\hat{c} = (\hat{\varphi}, \hat{\psi})$ is said to be an *initial* contract if $\hat{\varphi}$ and $\hat{\psi}$ are predicates on X'_I and X'_{ctr}, respectively, and an *update* contract otherwise. Given two valuations $v, v' \in V(X \cup X')$ and a contract $c = (\varphi, \psi)$ over $X \cup X'$, we say that (v, v') satisfies c, denoted by $(v, v') \models c$, if $(v, \pi(v')[X'_I]) \models \varphi \Rightarrow (v, \pi(v')[X'_{\text{ctr}}]) \models \psi$, where $\Rightarrow$ denotes implication. In addition, we say that (v, v') satisfies the assumption of c, denoted by $(v, v') \models_A c$ if $(v, \pi(v')[X'_I]) \models \varphi$. The valuation pair (v, v') satisfies the guarantee of c, denoted by $(v, v') \models_G c$, if $(v, \pi(v')[X'_{\text{ctr}}]) \models \psi)$. Note that we sometimes use the direct notation $(v, w'_I) \models_A c$ and $(v, w'_{\text{ctr}}) \models_G c$, where $w_I \in V(X_I)$ and $w_{\text{ctr}} \in V(X_{\text{ctr}})$ and for readability we use the concrete syntax $\varphi \vdash \psi$ to denote (φ, ψ) in our examples.

Definition 7.1

A *requirement interface* A is a tuple $\langle X_I, X_O, X_H, \hat{C}, C, \mathcal{R}, \rho \rangle$, where

- X_I, X_O and X_H are disjoint finite sets of *input*, *output* and *hidden* variables, respectively, and $X = X_I \cup X_O \cup X_H$ denotes the set of all variables;
- $\hat{C}$ and C are finite non-empty sets of *initial* and *update contracts*;
- $\mathcal{R}$ is a finite set of *requirement identifiers*;
- $\rho : \mathcal{R} \to \mathcal{P}(C \cup \hat{C})$ is a function mapping requirement identifiers to subsets of contracts, such that $\bigcup_{r \in \mathcal{R}} \rho(r) = C \cup \hat{C}$ and $\mathcal{P}$ is the powerset operator.

We say that a requirement interface is *receptive* if in any state it has defined behaviors for all inputs, that is $\bigvee_{(\hat{\varphi}, \hat{\psi}) \in \hat{C}} \hat{\varphi}$ and $\bigvee_{(\varphi, \psi) \in C} \varphi$ are both valid. This corresponds to input-enabledness of timed automata. A requirement interface is *fully-observable* if $X_H = \emptyset$. A requirement interface is *deterministic* if for all initial contracts $(\hat{\varphi}, \hat{\psi}) \in \hat{C}$, $\hat{\psi}$ has the form $\bigwedge_{x \in X_O} x' = c_{init}$, where c_{init} is a constant of the appropriate type, and for all $(\varphi, \psi) \in C$, ψ has the form $\bigwedge_{x \in X_{\text{ctr}}} x' = f(X)$, where f is a function over X that has the same type as x. Note that we denote by X_O/X_{ctr} the output/controllable variables involved in that contract.

Example 7.1. We use an abstract N-bounded FIFO buffer example to illustrate all the concepts introduced in this chapter. Note that the actual content of the buffer are ignored, as we are only interested in detecting whether the buffer is full, empty, or in between. Let A^{beh} be the behavioral model of the buffer. The buffer has two Boolean input variables enq, deq, i.e. $X_I^{beh} = \{\mathsf{enq}, \mathsf{deq}\}$, two Boolean output variables E, F, i.e. $X_O^{beh} = \{\mathsf{E}, \mathsf{F}\}$ and a bounded integer internal variable $k \in [0, N]$ for some $N \in \mathbb{N}$, i.e. $X_H^{beh} = \{k\}$. The textual requirements are listed below:

r_0: In the initial state, the buffer is empty and the inputs are ignored.

r_1: enq triggers an enqueue operation when the buffer is not full.

r_2: deq triggers a dequeue operation when the buffer is not empty.

r_3: E signals that the buffer is empty.

r_4: F signals that the buffer is full.

r_5: Simultaneous enq and deq (or their simultaneous absence), an enq on the full buffer or a deq on the empty buffer have no effect.

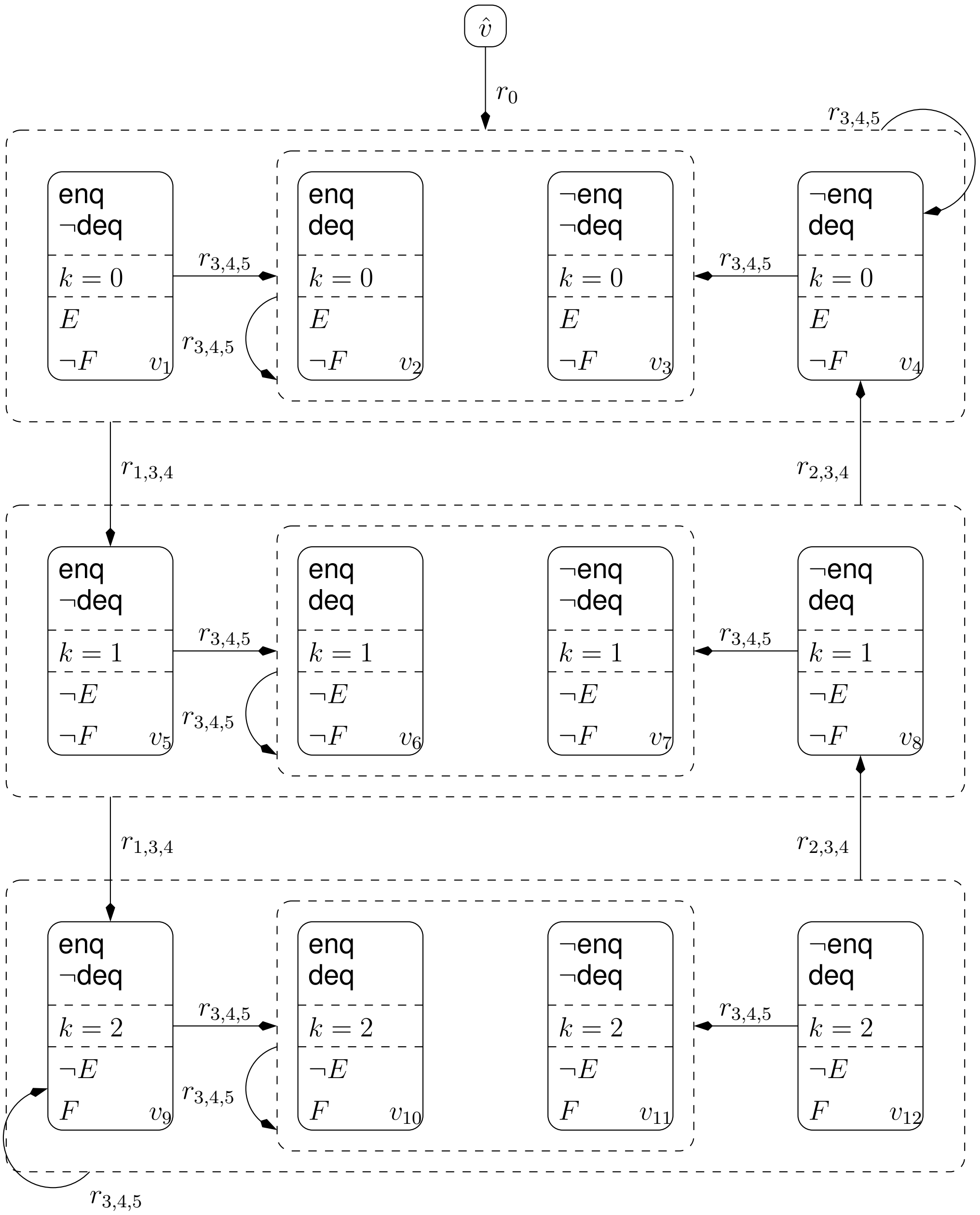
We formally define A^{beh} as $\hat{C}^{beh} = \{c_0\}$, $C^{beh} = \{c_i \mid i \in [1, 5]\}$, $\mathcal{R}^{beh} = \{r_i \mid i \in [0, 5]\}$, $\rho^{beh}(r_i) = \{c_i\}$, $X_I^{beh} = \{\mathsf{enq}, \mathsf{deq}\}$, $X_O^{beh} = \{\mathsf{E}, \mathsf{F}\}$ and $X_H^{beh} = \{\mathsf{N}, \mathsf{k}\}$ where

$$\begin{array}{lcl}
c_0 & : & \mathsf{true} \ \vdash \ (k' = 0) \wedge \mathsf{E}' \wedge \neg \mathsf{F}' \\
c_1 & : & \mathsf{enq}' \wedge \neg \mathsf{deq}' \wedge k < N \ \vdash \ k' = k + 1 \\
c_2 & : & \neg \mathsf{enq}' \wedge \mathsf{deq}' \wedge k > 0 \ \vdash \ k' = k - 1 \\
c_3 & : & \mathsf{true} \ \vdash \ k' = 0 \Leftrightarrow \mathsf{E}' \\
c_4 & : & \mathsf{true} \ \vdash \ k' = N \Leftrightarrow \mathsf{F}' \\
c_5 & : & (\mathsf{enq}' = \mathsf{deq}') \vee (\mathsf{enq}' \wedge \mathsf{F}) \vee (\mathsf{deq}' \wedge \mathsf{E}) \ \vdash \ k' = k
\end{array}$$

Contracts c_0 to c_2 follow the requirements r_0 to r_2 very closely and do not need further explanation. c_3 and c_4 are more interesting, as we are not allowed to reason over primed input variables in the assumption of the contracts. Thus, we could not simply model them as $k' = 0 \ \vdash \ \mathsf{E}'$, but had to shift the implication into the guarantee. The biimplication was only used to emphasize the connection between the empty buffer and the empty signal. Requirement r_5 and contract c_5 are needed, as without them the behavior of the buffer would not be defined for e.g. a dequeue operation on an empty buffer. In that case, the internal variable k might be set to any value arbitrarily. The requirement interface is receptive, since for all reachable states the behavior for any input is defined. It is deterministic, since all output and internal variables are set to a fixed value for any possible combination of inputs and internal values. □

7.2 Semantics

Given a requirement interface A defined over X, let $V = V(X) \cup \{\hat{v}\}$ denote the set of states in A, where a *state* v is a valuation $v \in V(X)$ or the *initial* state $\hat{v} \notin V(X)$. The latter is not a valuation, as the initial contracts do not specify unprimed variables. There is a transition between two states v and v' if (v, v') satisfies all its contracts. The transitions are labeled by the (possibly empty) set of requirement identifiers corresponding to contracts for which (v, v') satisfies their assumptions. The semantics $[[A]]$ of A is the following LTS.



$\hat{v}$
r_0
$r_{3,4,5}$
enq ¬deq $k=0$ E ¬F v_1
enq deq $k=0$ E ¬F v_2
¬enq ¬deq $k=0$ E ¬F v_3
¬enq deq $k=0$ E ¬F v_4
$r_{1,3,4}$
$r_{2,3,4}$
enq ¬deq $k=1$ ¬E ¬F v_5
enq deq $k=1$ ¬E ¬F v_6
¬enq ¬deq $k=1$ ¬E ¬F v_7
¬enq deq $k=1$ ¬E ¬F v_8
enq ¬deq $k=2$ ¬E F v_9
enq deq $k=2$ ¬E F v_{10}
¬enq ¬deq $k=2$ ¬E F v_{11}
¬enq deq $k=2$ ¬E F v_{12}

Figure 7.1: Labeled transition graph $[[A^{beh}]]$ illustrating the semantics of the bounded FIFO specification A^{beh}, where $N = 2$.

Definition 7.2
The semantics of the requirement interface A is the LTS $[[A]] = \langle V, \hat{v}, L, T\rangle$, where V is the set of states, $\hat{v}$ is the initial state, $L = \mathcal{P}(\mathcal{R})$ is the set of labels and $T \subseteq V \times L \times V$ is the transition relation, such that:

- $(\hat{v}, R, v') \in T$ if $v \in V(X)$, $\bigwedge_{\hat{c}\in\hat{C}}(\hat{v}, v') \models \hat{c}$ and $R = \{r \mid (\hat{v}, v) \models_A \hat{c}$ for some $\hat{c} \in \hat{C}$ and $\hat{c} \in \rho(r)\}$;
- $(v, R, v') \in T$ if $v, v' \in V(X)$, $\bigwedge_{c\in C}(v, v') \models c$ and $R = \{r \mid (v, v') \models_A c$ for some $c \in C$ and $c \in \rho(r)\}$.

We say that $\tau = v_0 \xrightarrow{R_1} v_1 \xrightarrow{R_2} \cdots \xrightarrow{R_n} v_n$ is an *execution* of the requirements interface A if $v_0 = \hat{v}$ and for all $1 \leq i \leq n-1$, $(v_i, R_{i+1}, v_{i+1}) \in T$. In addition, we use the following notation:

(1) $v \xrightarrow{R}$ iff $\exists v' \in V(X)$ s.t. $v \xrightarrow{R} v'$;
(2) $v \rightarrow v'$ iff $\exists R \in L$ s.t. $v \xrightarrow{R} v'$;
(3) $v \rightarrow$ iff $\exists v' \in V(X)$ s.t. $v \rightarrow v'$;
(4) $v \overset{\epsilon}{\Rightarrow} v'$ iff $v = v'$;
(5) $v \overset{w}{\Rightarrow} v'$ iff $\exists Y \subseteq X$ s.t. $\pi(v')[Y] = w$ and $v \rightarrow v'$;
(6) $v \overset{w}{\Rightarrow}$ iff $\exists v', Y \subseteq X$ s.t. $\pi(v')[Y] = w$ and $v \rightarrow v'$;
(7) $v \xRightarrow{w_1 \cdot w_2 \cdots w_n} v'$ iff $\exists v_1, \dots, v_{n-1}, v_n$ s.t. $v \xRightarrow{w_1} v_1 \xRightarrow{w_2} \cdots v_n \xRightarrow{w_n} v'$; and
(8) $v \xRightarrow{w_1 \cdot w_2 \cdots w_n}$ iff $\exists v'$ s.t. $v \xRightarrow{w_1 \cdot w_2 \cdots w_n} v'$.

We say that a sequence $\sigma \in V(X_{\text{obs}})^*$ is a *trace* of A if $\hat{v} \overset{\sigma}{\Rightarrow}$. We denote by $\mathcal{L}(A)$ the set of all traces of A. Given a trace σ of A, let A after $\sigma = \{v \mid \hat{v} \overset{\sigma}{\Rightarrow} v\}$. Given a state $v \in V$, let $\text{succ}(v) = \{v' \mid v \rightarrow v'\}$ be the set of successors of v.

Example 7.2. In Figure 7.1, we show the LTS $[[A^{beh}]]$ of A^{beh}. We use the notation $r_{1,2,3}$ to denote the set $\{r_1, r_2, r_3\}$. The outermost dotted groupings consist of all valuations with the same values for the internal and output variables. Each of the dottet groupings contains one valuation for the possible input values. In the initial valuation, all inputs are ignored, thus from the initial valuation we might reach any of the valuations v_1 to v_4. By applying an enqueue afterwards, we reach valuation v_5, with the label $r_{1,3,4}$. A single enqueue would lead us to v_9, a single dequeu to v_4. Any other inputs would lead to v_6 or v_7. For instance, $\hat{v} \xrightarrow{r_0} v_3 \xrightarrow{r_{1,3,4}} v_5 \xrightarrow{r_{3,4,5}} v_6$ is an execution in $[[A]]$ and the trace induced by the above execution is $(\neg\mathsf{enq}, \neg\mathsf{deq}, E, \neg F)$, $(\mathsf{enq}, \neg\mathsf{deq}, \neg E, \neg F)$, $(\mathsf{enq}, \mathsf{deq}, \neg E, \neg F)$.

7.3 Consistency, Refinement and Conjunction

A requirement interface consists of a set of contracts, that can be conflicting. Such an interface does not permit any correct implementation. We say that a requirement interface is *consistent* if it permits at least one correct implementation.

Definition 7.3
Let A be a requirement interface, $[[A]]$ its associated LTS, $v \in V$ a state and $\mathcal{C} = \hat{C}$ if v is initial, and C otherwise. We say that a state $v \in V$ is *consistent*, denoted by $\text{cons}(v)$, if for all $w_I \in V(X_I)$, there exists v' such that $w_I = \pi(v')[X_I']$, $\bigwedge_{c \in \mathcal{C}}(v, v') \models c$ and $\text{cons}(v')$. We say that A is *consistent* if $\text{cons}(\hat{v})$.

Example 7.3. A^{beh} is consistent – every reachable state accepts every input valuation and generates an output valuation satisfying all contracts. Consider now replacing c_2 in A^{beh} with the contract c_m : $\neg\mathsf{enq}' \wedge \mathsf{deq}' \wedge k \geq 0 \ \vdash \ k' = k - 1$, that incorrectly models r_2 and decreases the counter k upon deq even when the buffer is empty, setting it to the value minus one. This causes an inconsistency with the contracts c_3 and c_5, that state that if k equals zero the buffer is empty, and that dequeue on an empty buffer has no effect on k. □

We define the *refinement* relation between two requirement interfaces A^1 and A^2, denoted by $A^2 \preceq A^1$, as *trace inclusion*. Refinement is used for checking whether a more concrete model, or an implementation, conforms to an abstract specification. Cavalcanti et al. [65] gave an overview on different notions of refinement. Henzinger et al. [96] defined compositional refinement for assume-guarantee contracts. In our case, we will use refinement during the model-based mutation testing, described in Section 8.3, to check whether a mutant refines the original requirement interface.

Definition 7.4
Let A^1 and A^2 be two requirement interfaces. We say that A^2 *refines* A^1, denoted by $A^2 \preceq A^1$, if
(1) A^1 and A^2 have the same sets X_I, X_O and X_H of variables; and
(2) $\mathcal{L}(A^1) \subseteq \mathcal{L}(A^2)$.

We use a requirement interface to model a view of a system. Multiple views are combined by *conjunction*. The conjunction of two requirement interfaces is another requirement interface that is either inconsistent due to a conflict between views, or is the greatest lower bound with respect to the refinement relation. The conjunction of A^1 and A^2, denoted by $A^1 \wedge A^2$, is defined if the two interfaces share the same sets X_I, X_O and X_H of variables.

Definition 7.5
Let $A^1 = \langle X_I, X_H, X_O, \hat{C}^1, C^1, \mathcal{R}^1, \rho^1 \rangle$ and $A^2 = \langle X_I, X_H, X_O, \hat{C}^2, C^2, \mathcal{R}^2, \rho^2 \rangle$ be two requirement interfaces. Their conjunction $A = A^1 \wedge A^2$ is the requirement interface $\langle X_I, X_H, X_O, \hat{C}, C, \mathcal{R}, \rho \rangle$, where

- $\hat{C} = \hat{C}^1 \cup \hat{C}^2$ and $C = C^1 \cup C^2$;
- $\mathcal{R} = \mathcal{R}^1 \cup \mathcal{R}^2$; and
- $\rho(r) = \rho^1(r)$ if $r \in \mathcal{R}^1$ and $\rho(r) = \rho^2(r)$ otherwise.

Remark: For refinement and conjunction, we require the two interfaces to share the same alphabet. This additional condition is used to simplify definitions. It does not restrict the modeling – arbitrary interfaces can have their alphabets *equalized* without changing their properties by taking union of respective input, output and hidden variables. Contracts in the transformed interfaces do not constrain newly introduced variables. For requirement interfaces A^1 and A^2, alphabet equalization is defined if $(X_I^1 \cup X_I^2) \cap (X_{\text{ctr}}^1 \cup X_{\text{ctr}}^2) = (X_O^1 \cup X_O^2) \cap (X_H^1 \cup X_H^2) = \emptyset$. Otherwise, $A_1 \not\preceq A_2$ and vice versa, and $A^1 \wedge A^2$ is not defined.

Example 7.4. We now consider a *power consumption* view of the bounded FIFO buffer. Its model A^{pc} has the Boolean input variables enq and deq and a bounded integer output variable pc. The following textual requirements specify A^{pc}:

r_a: The power consumption equals zero when no enq/deq is requested.

r_b: The power consumption is bounded to 2 units otherwise.

The interface A^{pc} consists of $\hat{C}^{pc} = C^{pc} = \{c_a, c_b\}$, $\mathcal{R}^{pc} = \{r_i \mid i \in \{a, b\}\}$, $\rho(r_i) = \{c_i\}$ for $i \in \{a, b\}$, $X_I^{pc} = \{\mathsf{enq}, \mathsf{deq}\}$, $X_O^{pc} = \{\mathsf{pc}\}$ and $X_H^{pc} = \{\}$ where:

$$\begin{array}{lllll} c_a & : & \neg\mathsf{enq} \wedge \neg\mathsf{deq} & \vdash & \mathsf{pc}' = 0 \\ c_b & : & \mathsf{enq} \vee \mathsf{deq} & \vdash & \mathsf{pc}' \leq 2 \end{array}$$

The conjunction $A^{buf} = A^{beh} \wedge A^{pc}$ is the requirement interface where $X_I^{buf} = \{\mathsf{enq}, \mathsf{deq}\}$, $X_O^{buf} = \{\mathsf{E}, \mathsf{F}, \mathsf{pc}\}$, $X_H^{buf} = \{k\}$, $\hat{C}^{buf} = \{c_0, c_a, c_b\}$, $C^{buf} = \{c_1, c_2, c_3, c_4, c_5, c_a, c_b\}$, $\mathcal{R}^{pc} = \{r_i \mid i \in \{a, b, 0, 1, 2, 3, 4, 5\}\}$, and $\rho(r_i) = \{c_i\}$. □

We now show some properties of requirement interfaces.

The conjunction of two requirement interfaces with the same alphabet is the intersection of their traces.

Theorem 7.1
Let A^1 and A^2 be two consistent requirement interfaces defined over the same alphabet. Then either $A^1 \wedge A^2$ is inconsistent, or $\mathcal{L}(A^1 \wedge A^2) = \mathcal{L}(A^1) \cap \mathcal{L}(A^2)$.

Proof: Let A^1 and A^2 be two consistent requirement interfaces defined over the same alphabet. We first show that $A^1 \wedge A^2$ can be inconsistent. For this, we choose A^1 and A^2 such that $X_I^1 = X_I^2 = \{x\}$, $X_O^1 = X_O^2 = \{y\}$, $X_H^1 = X_H^2 = \emptyset$, $\hat{C}^1 = C^1 = \{c^1\}$ and $\hat{C}^2 = C^2 = \{c^2\}$, where $c^1 = \text{true} \vdash y' = 0$

and $c^2 = \text{true} \vdash y' = 1$. It is clear that both A^1 and A^2 are consistent – for any new value of x, A^1 (A^2) updates the value of y to 0 (1). However, $A^1 \wedge A^2$ is inconsistent, since no implementation can satisfy the guarantees of c^1 and c^2 simultaneously ($y' = 0 \wedge y' = 1$).

Assume that $A^1 \wedge A^2$ is consistent. We now prove that $\mathcal{L}(A^1 \wedge A^2) \subseteq \mathcal{L}(A^1) \cap \mathcal{L}(A^2)$. The proof is done by induction on the size of σ.

Base case: $\sigma = \epsilon$. We have that $(A^1 \wedge A^2)$ after $\epsilon = (A^1$ after $\epsilon) \wedge (A^2$ after $\epsilon) = \{\hat{v}\}$.

Inductive step: Let σ be an arbitrary trace of size n such that $\sigma \in \mathcal{L}(A^1 \wedge A^2)$. By inductive hypothesis, $\sigma \in \mathcal{L}(A^1)$ and $\sigma \in \mathcal{L}(A^2)$. Consider an arbitrary w_{obs} such that $\sigma \cdot w_{obs} \in \mathcal{L}(A^1 \wedge A^2)$. Let $V_{1\wedge 2} = \{v \mid \hat{v} \overset{\sigma}{\Rightarrow} v\}$. By the definition of semantics of requirement interfaces, it follows that $V'_{1\wedge 2} = \{v' \mid v \overset{w_{obs}}{\Longrightarrow}_{1\wedge 2} v'$ for some $v \in V_{1\wedge 2}\}$ is non-empty. Let v' be an arbitrary state in $V'_{1\wedge 2}$, hence we have that $v \rightarrow_{1\wedge 2} v'$. Let $C^i_* = \{(\varphi, \psi) \mid (\varphi, \psi) \in C^i$ and $(v, \pi(v')[X'_I]) \models \varphi\}$ for $i \in \{1, 2\}$ denote the (possibly empty) set of contracts in A^i for which the pair (v, v') satisfies its assumptions. By the definition of conjunction and semantics of requirement interfaces, we have that $(v, v') \models \bigwedge_{(\varphi,\psi)\in C^1_*} \psi \wedge \bigwedge_{(\varphi,\psi)\in C^2_*} \psi$. It follows that $(v, v') \models \bigwedge_{(\varphi,\psi)\in C^1_*} \psi$, and $(v, v') \models \bigwedge_{(\varphi,\psi)\in C^2_*} \psi$, hence we can conclude that $v \rightarrow_1 v'$ and $v \rightarrow_2 v'$, hence $\sigma \cdot w_{obs} \in \mathcal{L}(A^1)$ and $\sigma \cdot w_{obs} \in \mathcal{L}(A^2)$, which concludes the proof that $\mathcal{L}(A^1 \wedge A^2) \subseteq \mathcal{L}(A^1) \cap \mathcal{L}(A^2)$.

We now show that $\mathcal{L}(A^1 \wedge A^2) \supseteq \mathcal{L}(A^1) \cap \mathcal{L}(A^2)$. The proof is by induction on the size of σ.

Base case: $\sigma = \epsilon$. We have that $(A^1 \wedge A^2)$ after $\epsilon = (A^1$ after $\epsilon) \wedge (A^2$ after $\epsilon) = \{\hat{v}\}$.

Inductive step: Let σ be an arbitrary trace of size n such that $\sigma \in \mathcal{L}(A^1)$ and $\sigma \in \mathcal{L}(A^2)$. By inductive hypothesis, it follows that $\sigma \in \mathcal{L}(A^1 \wedge A^2)$. Let $\sigma = \sigma' \cdot v$. Consider an arbitrary w_{obs} such that $\sigma \cdot w_{obs} \in \mathcal{L}(A^1)$ and $\sigma \cdot w_{obs} \in \mathcal{L}(A^2)$. It follows that $v \overset{w_{obs}}{\Longrightarrow}_1$ and $v \overset{w_{obs}}{\Longrightarrow}_2$. Let $C^i_* = \{(\varphi, \psi) \mid (\varphi, \psi) \in C^i$ and $(v, w_{obs}) \models \varphi\}$ for $i \in \{1, 2\}$ denote the (possibly empty) set of contracts in A^i for which the pair (v, w_{obs}) satisfies its assumptions. By the definition of conjunction and the semantics of requirement interfaces, we have that there exist v' and v'' such that $(v, v') \models \bigwedge_{(\varphi,\psi)\in C^1_*} \psi$, and $(v, v'') \models \bigwedge_{(\varphi,\psi)\in C^2_*} \psi$. By the assumption that $A^1 \wedge A^2$ is consistent, we have that there exists v' such that $(v, v') \models \bigwedge_{(\varphi,\psi)\in C^1_*} \psi \wedge \bigwedge_{(\varphi,\psi)\in C^2_*} \psi$, $\sigma \cdot w_{obs} \in \mathcal{L}(A^1 \wedge A^2)$, which concludes the proof that $\mathcal{L}(A^1 \wedge A^2) \supseteq \mathcal{L}(A^1) \cap \mathcal{L}(A^2)$. □

The conjunction of two requirement interfaces with the same alphabet is either inconsistent, or it is the greatest lower bound with respect to refinement.

Theorem 7.2
Let A^1 and A^2 be two consistent requirement interfaces defined over the same alphabet such that $A^1 \wedge A^2$ is consistent. Then $A^1 \wedge A^2 \preceq A^1$ and $A^1 \wedge A^2 \preceq A^2$, and for all consistent requirement interfaces A, if $A \preceq A^1$ and $A \preceq A^2$, then $A \preceq A^1 \wedge A^2$.

Proof: Assume that $A^1 \wedge A^2$ is consistent and consider an arbitrary consistent interface A that shares the same alphabet with A^1 and A^2. The proofs that $A^1 \wedge A^2 \preceq A^1$, $A^1 \wedge A^2 \preceq A^2$, and that if $A \preceq A^1$ and $A \preceq A^2$, then $A \preceq A^1 \wedge A^2$ follow directly from Theorem 7.1 and the definition of refinement. □

The following theorem states that the conjunction of an inconsistent requirement interface with any other interface remains inconsistent. This result enables incremental detection of inconsistent specifications.

Theorem 7.3
Let A be an inconsistent requirement interface. Then for all consistent requirement interfaces A' with the same alphabet as A, $A \wedge A'$ is also inconsistent.

Proof: Follows directly from the definition of conjunction, which constrains the guarantees of individual interfaces. □

8 Consistency Checking and Test-Case Generation

Parts of this chapter are based on our publication at FMICS 2015 [11].

In this chapter, we present our test-case generation and execution framework and instantiate it with bounded model checking techniques. For now, we assume that all variables range over finite domains. This restriction can be lifted by considering richer data domains in addition to theories that have decidable quantifier elimination, such as linear arithmetic over reals. Before executing the test-case generation, we should apply a consistency check on the requirement interface, to ensure the generation starts from an implementable specification.

8.1 Bounded Consistency Checking

In order to check k-bounded consistency of a requirement interface A, we unfold the transition relation of A in k steps, and encode the definition of consistency in a straight-forward manner. The transition relation of an interface is the conjunction of its contracts, where a contract is represented as an implication between its assumption and guarantee predicates. Let

$$\hat{\theta} = \bigwedge_{(\hat{\varphi},\hat{\psi}) \in \hat{C}} \hat{\varphi} \Rightarrow \hat{\psi}$$

and

$$\theta = \bigwedge_{(\varphi,\psi) \in C} \varphi \Rightarrow \psi$$

Then, the k-bounded consistency check for A corresponds to checking the satisfiability of the formula

$$\forall X_I^0 . \exists X_{\text{ctr}}^0 \ldots \forall X_I^k . \exists X_{\text{ctr}}^k . \, \theta^0 \wedge \theta^1 \wedge \cdots \wedge \theta^k \text{ where}$$

$\theta^0 = \hat{\theta}[X' \backslash X^0]$ and $\theta^i = \theta[X' \backslash X^i, X \backslash X^{i-1}]$, $1 \leq i \leq k$, where $X \backslash Y$ means substituting X by Y in the formula.

To implement the consistency check, it can be transformed to a satisfiability problem, which can then be solved by an SMT-solver, as e.g. Z3. The first step is to construct a symbolic representation of the initial contracts and the transition relation.

The transition relation is then unfolded for each step by renaming the occurrence of each variable such that it is indexed by the corresponding step. In each step i the undecorated variables are indexed with $i - 1$, while the decorated variables are indexed with i thus keeping the relation between the valuations of each step. Given a set X of variables, we denote by X^i the copy of the set, in which every variable is indexed by i. The conjunction of all instances of the step relation up to a certain depth is an open formula, leaving all variables free. The consistency check is bounded by a certain depth k.

Example 8.1. Consider the mutated contract c_m presented in Example 7.3, that mutates the dequeue contract. The system is consistent in its initial valuation if for any possible input the conjunction of the contracts is satisfiable. For any inputs other than a single dequeue, this holds producing the valuations. $(\neg\mathsf{enq}, \neg\mathsf{deq}, E, \neg F, k = 0)$, $(\mathsf{enq}, \neg\mathsf{deq}, \neg E, \neg F, k = 1)$ and $(\mathsf{enq}, \mathsf{deq}, E, \neg F, k = 0)$. However, for a single dequeue, contract c_m requires k to become -1, while contract c_5 requires it to be set to 0. Thus, there does not exist a valid valuation of the controllable variables, and the system is detected to be inconsistent. □

8.2 Test-Case Generation

A *test case* is an experiment executed on the SUT I by the *tester*. We assume that I is a black-box that is only accessed via its observable interface. We assume that I can be modeled as an input-enabled, deterministic requirement interface. The restriction to deterministic implementations is for presentation purposes only, the technique is general and can also be applied to non-deterministic systems. Without loss of generality, we can represent I as a total sequential function $I : V(X_I) \times V(X_{\text{obs}})^* \to V(X_O)$. This means, that the behavior of the implementation can be represented by its history of observable inputs and outputs, $V(X_{\text{obs}})^*$, which together with a new input $V(X_I)$ leads to a new output $V(X_O)$. A test case T_A for a requirement interface A over X takes a history of actual input/output observations $\sigma \in \mathcal{L}(A)$ and returns either the next input value to be executed or a verdict. Hence, a test case can be represented as a *partial* function $T_A : \mathcal{L}(A) \to V(X_I) \cup \{\textbf{pass}, \textbf{fail}\}$.

We first consider the problem of generating a test case from A. The test-case generation procedure is driven by a *test purpose*. Here, a test purpose is a condition specifying the target set of states that a test execution should reach. Hence, it is a formula Π defined over X_{obs}.

Given a requirement interface A, let $\hat{\phi} = \bigvee_{(\hat{\varphi},\hat{\psi})\in\hat{C}} \hat{\varphi} \wedge \bigwedge_{(\hat{\varphi},\hat{\psi})\in\hat{C}} \hat{\varphi} \Rightarrow \hat{\psi}$ and $\phi = \bigvee_{(\varphi,\psi)\in C} \varphi \wedge \bigwedge_{(\varphi,\psi)\in C} \varphi \Rightarrow \psi$. The predicates $\hat{\phi}$ and ϕ encode the transition relation of A, where $\hat{\phi}$ depicts the transition relation of the initial contracts, with the additional requirement that at least one assumption must be satisfied. Thus we avoid input vectors for which the test purpose can be trivially reached due to under-specification. A test case for A that can reach Π is defined iff there exists a trace $\sigma = \sigma' \cdot w_{obs}$ in $\mathcal{L}(A)$ such that $w_{obs} \models \Pi$. The test purpose Π can be reached in A in at most k steps if

$$\exists X^0, \ldots, X^k . \phi^0 \wedge \ldots \wedge \phi^k \wedge \bigvee_{i \le k} \Pi[X_{\text{obs}} \backslash X^i_{\text{obs}}],$$

where $\phi^0 = \hat{\phi}[X' \backslash X^0]$ and $\phi^i = \phi[X' \backslash X^i, X \backslash X^{i-1}]$ represent the transition relation of A unfolded in the i-th step.

Example 8.2. Consider our example of the abstract buffer, and the test purpose $\Pi = F$. The formula for reaching the test purpose in at most 2 steps is presented below. The first line is the initial contract, the next ten lines are the step relation of the first and the second steps, followed by the test purpose, which must hold for $k = 0$ or $k = 1$ or $k = 2$.

$\mathsf{true} \vdash (k_0 = 0) \wedge \mathsf{E}_0 \wedge \neg\mathsf{F}_0 \wedge$
$\mathsf{enq}_1 \wedge \neg\mathsf{deq}_1 \wedge k_0 < 2 \vdash k_1 = k_0 + 1 \wedge$
$\neg\mathsf{enq}_1 \wedge \mathsf{deq}_1 \wedge k_0 > 0 \vdash k_1 = k_0 - 1 \wedge$
$\mathsf{true} \vdash k_1 = 0 \Leftrightarrow \mathsf{E}_1 \wedge$
$\mathsf{true} \vdash k_1 = 2 \Leftrightarrow \mathsf{F}_1 \wedge$
$(\mathsf{enq}_1 = \mathsf{deq}_1) \vee (\mathsf{enq}_1 \wedge \mathsf{F}_0) \vee (\mathsf{deq}_1 \wedge \mathsf{E}_0) \vdash k_1 = k_0 \wedge$
$\mathsf{enq}_2 \wedge \neg\mathsf{deq}_2 \wedge k_1 < 2 \vdash k_2 = k_1 + 1 \wedge$
$\neg\mathsf{enq}_2 \wedge \mathsf{deq}_2 \wedge k_2 > 0 \vdash k_2 = k_1 - 1 \wedge$
$\mathsf{true} \vdash k_2 = 0 \Leftrightarrow \mathsf{E}_2 \wedge$
$\mathsf{true} \vdash k_2 = 2 \Leftrightarrow \mathsf{F}_2 \wedge$
$(\mathsf{enq}_2 = \mathsf{deq}_2) \vee (\mathsf{enq}_2 \wedge \mathsf{F}_1) \vee (\mathsf{deq}_2 \wedge \mathsf{E}_1) \vdash k_2 = k_1 \wedge$
$(F_0 \vee F_1 \vee F_2)$

Given A and Π, assume that there exists a trace σ in $\mathcal{L}(A)$ that reaches Π. Let σ_I be a projection to inputs, s.t. $\sigma_I = \pi(\sigma)[X_I] = w_I^0 \cdot w_I^1 \cdots w_I^k$. We first compute $\omega_{\sigma_I,A}$ (see Algorithm 13), a formula characterizing the set of output sequences that A allows on input σ_I. The formula $\omega_{\sigma_I,A}$ can be seen as a monitor for A under input σ_I.

```
Algorithm 13 OutMonitor
Input: $\sigma_I = w_I^0 \cdot w_I^1 \cdots w_I^k$, $A$
Output: $\omega_{\sigma_I,A}$
1: $\omega^0_{\sigma_I,A} \leftarrow \hat{\phi}[X'_I\backslash w^0_I, X'_{\text{ctr}}\backslash X^0_{\text{ctr}}]$    ▷ Substitution in step relation for initial contracts
2: for $i = 1$ to $k$ do
3:     $\omega^i_{\sigma_I,A} \leftarrow \phi[X_I\backslash w_I^{i\text{-}1}, X'_I\backslash w^i_I, X_{\text{ctr}}\backslash X^{i\text{-}1}_{\text{ctr}}, X'_{\text{ctr}}\backslash X^i_{\text{ctr}}]$    ▷ Substitution in norm. step relation
4: end for
5: $\omega^*_{\sigma_I,A} \leftarrow \omega^0_{\sigma_I,A} \wedge \ldots \wedge \omega^k_{\sigma_I,A}$    ▷ Auxiliary variable
6: $\omega_{\sigma_I,A} \leftarrow \mathbf{qe}(\exists X^0_H, X^1_H, \ldots, X^k_H . \omega^*_{\sigma_I,A})$    ▷ Quantifier elimination
7: return $\omega_{\sigma_I,A}$
```

Let $\hat{\phi} = \bigwedge_{(\hat{\varphi},\hat{\psi})\in\hat{C}} \hat{\varphi} \Rightarrow \hat{\psi}$ and $\phi = \bigwedge_{(\varphi,\psi)} \varphi \Rightarrow \psi$. For every step i, we represent by $\omega^i_{\sigma_I,A}$ the allowed behavior of A constrained by σ_I (Lines $1-4$). The formula $\omega^*_{\sigma_I,A}$ (Line 5) describes the transition relation of A, unfolded to n steps and constrained by σ_I. However, this formula refers to the hidden variables of A and cannot be directly used to characterize the set of output sequences allowed by A under σ_I. Since any implementation of hidden variables that preserves correctness of the outputs is acceptable, it suffices to existentially quantify over hidden variables in $\omega^*_{\sigma_I,A}$. After eliminating the existential quantifiers, which can e.g. be done by Z3, using the **qe** strategy [73], we obtain a simplified formula $\omega_{\sigma_I,A}$ over output variables only (Line 6).

Example 8.3. Using the test purpose $\Pi = F$ on our abstract buffer, we get the input sequence

$$\sigma \;=\; \begin{array}{l} (*,*) \\ (\mathsf{enq}, \neg\mathsf{deq}, \neg\mathsf{E}, \neg\mathsf{F}) \\ (\mathsf{enq}, \neg\mathsf{deq}) \end{array}$$

and the output monitor

$$\sigma \;=\; \begin{array}{l} (\mathsf{E}, \neg\mathsf{F}) \\ (\neg\mathsf{E}, \neg\mathsf{F}) \\ (\neg\mathsf{E}, \mathsf{F}) \end{array}$$

```
Algorithm 14 $T_{\sigma_I,A}$
Input: $\sigma_I = w_I^0 \cdots w_I^k$, $A$, $\sigma = w^0_{obs} \cdots w^k_{obs}$
Output: {pass, fail}
$\omega_{\sigma_I,A} \leftarrow$ OutMonitor$(\sigma_I, A)$    ▷ Produce output monitor
for $i = 0$ to $k$ do
    $w^i_O \leftarrow \pi(w^i_{obs})[X_O]$
end for
$\omega^{0,k}_{\sigma_I,A} \leftarrow \omega_{\sigma_I,A}[X^0_O\backslash w^0_O, \ldots, X^k_O\backslash w^k_O]$    ▷ Substitute outputs by outputs observed by SUT
if $\omega^{0,k}_{\sigma_I,A}$ = true then    ▷ Output minitor satisfied
    return pass
else if $\omega^{0,k}_{\sigma_I,A}$ = false then    ▷ Output monitor not satisfied
    return fail
end if
```

Let $T_{\sigma_I,A}$ be a test case, parameterized by the input sequence σ_I and the requirement interface A from which it was generated. It is a partial function, where $T_{\sigma_I,A}(\sigma)$ is defined if $|\sigma| \leq |\sigma_I|$ and for all $0 \leq i \leq |\sigma|$, $w_I^i = \pi(w_{obs}^i)[X_I]$, where $\sigma_I = w_I^0 \cdots w_I^k$ is the input sequence and $\sigma = w_{obs}^0 \cdots w_{obs}^k$ are the outputs observed from the SUT. Algorithm 14 gives a constructive definition of the test case $T_{\sigma_I,A}$. It starts by producing the output monitor for the given input sequence (Line 1). Then it substitutes all output variables in the monitor, by the outputs observed from the SUT (Lines 2-5). If the monitor is satisfied by the outputs, it returns the verdict $pass$, otherwise it returns $fail$.

8.2.1 Incremental Test-Case Generation:

So far, we considered test case generation for a complete requirement interface A, without considering its internal structure. We now describe how test cases can be *incrementally* generated when the interface A consists of multiple views, e.g. $A = A^1 \wedge A^2$. Within this section, we consider two views for the sake of simplicity.

The advantage of the incremental approach is a huge improvement with regards to the runtime of the test-case generation. In Section 11 we will present that even on the abstract buffer, this approach can already result in notable runtime improvements. The work presented in this section was mostly driven by Stefan Tiran. Additional work he did on the topic of incremental test-case generation was later published in an additional paper by Aichernig et al. [21]. In that paper they present a railway interlocking system, and show that the incremental approach for test-case generation can reduce the runtime from over an hour to about eight minutes.

Let Π be a test purpose for the view modeled with A_1. We first check whether Π can be reached in A^1, which is a simpler check than doing it on the conjunction $A^1 \wedge A^2$. If Π can be reached, we fix the input sequence σ_I that steers A^1 to Π. Instead of creating the test case T_{σ_I,A^1}, we generate $T_{\sigma_I,A^1 \wedge A^2}$, which keeps σ_I as the input sequence, but collects output guarantees of A^1 and A^2. Such a test case steers the SUT towards the test purpose in the view modeled by A^1, but is able to detect possible violations of both A^1 and A^2.

We note that test-case generation for fully observable interfaces is simpler than the general case, because there is no need for the quantifier elimination, due to the absence of hidden variables in the model. A test case from a deterministic interface is even simpler as it is a direct mapping from the observable trace that reaches the test purpose – there is no need to collect constraints on the output since the deterministic interface does not admit any freedom to the implementation on the choice of output valuations.

Example 8.4. Consider the requirement interface A_{beh} for the behavioral view of the 2-bounded buffer presented in Example 7.1, and the test purpose F. Our test-case generation procedure gives the input vector σ_I of size 3 such that $\sigma_I[0] = w_I^0 = \{\mathsf{enq}, \mathsf{deq}\}$, $\sigma_I[1] = w_I^1 = \{\mathsf{enq}, \neg\mathsf{deq}\}$ and $\sigma_I[2] = w_I^2 = \{\mathsf{enq}, \neg\mathsf{deq}\}$. The observable output constraints for σ_I (which are encoded in OutMonitor) are $\mathsf{E} \wedge \neg\mathsf{F}$ in Step 0, $\neg\mathsf{E} \wedge \neg\mathsf{F}$ in Step 1 and $\neg\mathsf{E} \wedge \mathsf{F}$ in Step 2. Together, the input vector σ_I and the associated output constraints form the test case $T_{\sigma_I,beh}$. By using the incremental test-case generation procedure, we can extend $T_{\sigma_I,beh}$ to a test case $T_{\sigma_I,buf}$ that also takes into account the power consumption view of the buffer (presented in Example 7.4), resulting in output constraints $\mathsf{E} \wedge \neg\mathsf{F} \wedge \mathsf{pc} \leq 2$ in Step 0, $\neg\mathsf{E} \wedge \neg\mathsf{F} \wedge \mathsf{pc} \leq 2$ in Step 1 and $\neg\mathsf{E} \wedge \mathsf{F} \wedge \mathsf{pc} \leq 2$ in Step 2. □

8.2.2 Test-Case Execution

Let A be a requirement interface, I a SUT with the same set of variables as A, and $T_{\sigma_I,A}$ a test case generated from A. Algorithm 15 defines the test-case execution procedure TestExec that takes as input I and $T_{\sigma_I,A}$ and outputs a verdict **pass** or **fail**. Note that, like in Part I, we perform offline testing, thus

Algorithm 15 TestExec

```
Input: $I$, $T_{\sigma_I,A}$
Output: $\{\textbf{pass}, \textbf{fail}\}$
 1: $\mathsf{test_{in}} \ : \ V(X_I) \cup \{\textbf{pass}, \textbf{fail}\}$
 2: $\mathsf{test_{out}} \ : \ V(X_O)$
 3: $\sigma \leftarrow \epsilon$
 4: $\mathsf{test_{in}} \leftarrow T_{\sigma_I,A}(A, \sigma)$          ▷ Retrieve next input from test case
 5: while $\mathsf{test_{in}} \notin \{\textbf{pass}, \textbf{fail}\}$ do
 6:     $\mathsf{test_{out}} \leftarrow I(\mathsf{test_{in}}, \sigma)$          ▷ Stimulate SUT and retrieve its output
 7:     $\sigma \leftarrow \sigma \cdot (\mathsf{test_{in}} \cup \mathsf{test_{out}})$
 8:     $\mathsf{test_{in}} \leftarrow T_{\sigma_I,A}(A, \sigma)$          ▷ Check output monitor for correctness
 9: end while
10: return $\mathsf{test_{in}}$
```

generating the tests first, and executing them afterwards. TestExec gets the next test input $test_{in}$ from the given test case $T_{\sigma_I,A}$ (Lines 4, 8), stimulates at every step the SUT I with this input and waits for an output $test_{out}$ (Line 6). The new inputs/outputs observed are stored in σ (Line 7), which is given as input to $T_{\sigma_I,A}$. The test case monitors if the observed output is correct with respect to A. The procedure continues until a **pass** or **fail** verdict is reached (Line 5). Finally, the verdict is returned (Line 10).

Proposition 8.1

Let A, $T_{\sigma_I,A}$ and I be an arbitrary requirement interface, a test case generated from A and an implementation, respectively. Then, we have that:

1. *if $I \preceq A$, then TestExec$(I, T_{\sigma_I,A}) =$ **pass**;*

   Proof: We first proof the loop invariant that if $I \preceq A$, then $test_{in} \neq$ **fail** and $\sigma \in \mathcal{L}(A)$. In Line 6 the next input $test_{in}$ is by definition of the test case $T_{\sigma_I,A}$ the next valid input in σ_I. The extended trace in Line 7 is a trace of I. If $I \preceq A$ this extended trace is by definition of refinement also a trace of A. In this case, by definition of the test case $T_{\sigma_I,A}$ the next input $test_{in}$ of Line 8 will be either the **pass** verdict or the next input of σ_I. Hence, the invariant holds. Consequently, when the loop terminates the **pass** verdict is returned. □

2. *if TestExec$(I, T_{\sigma_I,A}) =$ **fail**, then $I \npreceq A$.*

   Proof: By negation we obtain the proposition: if $I \preceq A$, then TestExec$(I, T_{\sigma_I,A}) \neq$ **fail**. This follows directly from the loop invariant established above. □

Proposition 8.1 immediately holds for test cases generated incrementally from a requirement interface of the form $A = A^1 \wedge A^2$. In addition, we notice that a test case T_{σ_I,A^1}, generated from a single view A^1 of A does not need to be extended to be useful, and can be used to incrementally show that a SUT does not conform to its specification. We state the property in the following corollary, that follows directly from Proposition 8.1 and Theorem 7.2.

Corollary 8.1

*Let $A = A^1 \wedge A^2$ be an arbitrary requirement interface composed of A^1 and A^2, I an arbitrary implementation and T_{σ_I,A^1} an arbitrary test case generated from A^1. Then, if TestExec$(I, T_{\sigma_I,A^1}) =$ **fail**, then $I \npreceq A^1 \wedge A^2$.*

Algorithm 16 3-buffer implementation I.

```
Input: enq, dec
Output: E, F, pc
wait for inputs                          ▷ We receive both enq and deq simultaneously
if $\neg$enq $\wedge$ $\neg$dec then
    pc $\leftarrow 0$                    ▷ Update power consumption
else
    pc $\leftarrow 1$                    ▷ Update power consumption
end if
while true do
    wait for inputs                      ▷ We receive both enq and deq simultaneously
    if enq $\wedge$ $\neg$dec $\wedge$ $k < 3$ then
        $k \leftarrow k + 1$
    else if $\neg$enq $\wedge$ dec $\wedge$ $k > 0$ then
        $k \leftarrow k - 1$
    end if
    if $\neg$enq $\wedge$ $\neg$dec then
        pc $\leftarrow 0$                ▷ Update power consumption
    else
        pc $\leftarrow 1$                ▷ Update power consumption
    end if
    if $k = 3$ then                      ▷ Full buffer
        F $\leftarrow$ true; E $\leftarrow$ false
    else if $k = 0$ then                 ▷ Empty buffer
        F $\leftarrow$ false; E $\leftarrow$ true
    else
        F $\leftarrow$ false; E $\leftarrow$ false
    end if
end while
```

Example 8.5. Consider the abstract implementation I of a 3-buffer, as illustrated in Algorithm 16. We assume that the power consumption is updated directly in a PC variable. Although I is correctly implementing a 3-buffer, it is a faulty implementation of the 2-buffer specified in Example 7.1 and Example 7.4. In fact, when I already contains two items, the buffer is still not full, which is in contrast with requirement r_4 of a 2-buffer. Executing tests $T_{\sigma_I,beh}$ and $T_{\sigma_I,buf}$ from Example 8.4 will both result in a **fail** test verdict. □

8.2.3 Traceability

Requirement identifiers as first-class elements in requirement interfaces facilitate traceability between informal requirements, views and test cases. A test case generated from a view A^i of an interface $A = A^1 \wedge \ldots \wedge A^n$ is naturally mapped to the set $\mathcal{R}^i$ of requirements. In addition, requirement identifiers enable tracing violations caught during consistency checking and test-case execution back to the conflicting/violated requirements.

Tracing inconsistent interfaces to conflicting requirements: when we detect an inconsistency in a requirement interface A defining a set of contracts C, we use QuickXPlain, a standard conflict set detection algorithm [106], in order to compute a minimal set of contracts $C' \subseteq C$ such that C' is inconsistent.

Once we compute C', we use the requirement mapping function ρ defined in A, to trace back the set $\mathcal{R}' \subseteq \mathcal{R}$ of conflicting requirements.

*Tracing **fail** verdicts to violated requirements:* in fully observable interfaces, every trace induces at most one execution. In that case, a test case resulting in **fail** can be traced to a unique set of violated requirements. This is not the case in general for interfaces with hidden variables. A trace that violates such an interface may induce multiple executions resulting in **fail** with different valuations of hidden variables, and thus different sets of violated requirements. In this case, we report all sets to the user, but ignore internal valuations that would introduce an internal requirement violation before inducing the visible violation.

We propose a tracing procedure *TraceFailTC*, presented in Algorithm 17, that gives useful debugging data regarding violation of test cases in the general case. The algorithm takes as input a requirement interface A and a trace $\sigma \notin \mathcal{L}(A)$. The trace σ that is given as input to the algorithm is obtained from executing a test case for A that leads to a **fail** verdict. The algorithm runs a main loop that at each iteration computes a *debugging pair*, that consists of an execution $\tau = \pi(\sigma)[X_{\text{obs}}]$ and a set failR $\subseteq \mathcal{R}$ of requirements. We assume that the trace does not violate initial contracts to simplify the presentation. The extension to the general case is straightforward. The execution τ completes the faulty trace with valuations of hidden variables that are consistent with the violation of the requirement interface in the last step. The set failR contains all the requirements that are violated by the execution τ. We initialize the algorithm by setting an auxiliary variable C^* to the set of all update contracts C (Line 3). In every iteration of the main loop, we encode in ϕ^*_{obs} all the executions induced by σ that violate at least one contract in C^* (Lines 6 and 7). In the next step (Line 8), we check the satisfiability of the formula ϕ^*_{obs} (**sat**(ϕ^*_{obs})), a function that returns b = true and a sequence (model) of hidden variable valuations $w^0_H, \dots, w^n_H$ if ϕ^*_{obs} is satisfiable, and (b = false, $\sigma_H = \epsilon$) otherwise. In the former case, we combine σ and σ_H into an execution τ (Line 10). We collect in failR all requirements that are violated by τ and remove the corresponding contracts from C^* (Lines $11 - 16$). The debugging pair (τ, failR) is added to debugSet (Line 16). The procedure terminates and returns debugSet when either C^* is empty or σ cannot violate any remaining contract in C^*, thus ensuring that every requirement that can be violated by σ is part of at least one debugging pair in debugSet.

Example 8.6. Consider the execution trace

$$\sigma = \begin{array}{l}(\mathsf{enq}, \mathsf{deq}, \mathsf{E}, \neg\mathsf{F})\\(\mathsf{enq}, \neg\mathsf{deq}, \neg\mathsf{E}, \neg\mathsf{F})\\(\mathsf{enq}, \neg\mathsf{deq}, \neg\mathsf{E}, \neg\mathsf{F})\end{array}$$

that was produced by an execution of the 3-buffer implementation from Algorithm 16 and results in a **fail** verdict when executing the test $T_{\sigma_I,beh}$ from Example 8.4, as we expected a 2-buffer. The tracing procedure gives as debugging information the set debugSet $= \{(\tau_1, \{r_4\}), (\tau_2, \{r_1, r_3\}), (\tau_3, \{r_1\})\}$, where τ_1, τ_2 and τ_3 correspond to the following executions that can lead to violations of requirements r_4, r_1, r_3 and r_1, respectively.

$$\tau_1 = \begin{array}{l}(\mathsf{enq}, \mathsf{deq}, k = 0, \mathsf{E}, \neg\mathsf{F})\\(\mathsf{enq}, \neg\mathsf{deq}, k = 1, \neg\mathsf{E}, \neg\mathsf{F})\\(\mathsf{enq}, \neg\mathsf{deq}, k = 2, \neg\mathsf{E}, \neg\mathsf{F})\end{array}$$

$$\tau_2 = \begin{array}{l}(\mathsf{enq}, \mathsf{deq}, k = 0, \mathsf{E}, \neg\mathsf{F})\\(\mathsf{enq}, \neg\mathsf{deq}, k = 1, \neg\mathsf{E}, \neg\mathsf{F})\\(\mathsf{enq}, \neg\mathsf{deq}, k = 0, \neg\mathsf{E}, \neg\mathsf{F})\end{array}$$

$$\tau_3 = \begin{array}{l}(\mathsf{enq}, \mathsf{deq}, k = 0, \mathsf{E}, \neg\mathsf{F})\\(\mathsf{enq}, \neg\mathsf{deq}, k = 1, \neg\mathsf{E}, \neg\mathsf{F})\\(\mathsf{enq}, \neg\mathsf{deq}, k = 1, \neg\mathsf{E}, \neg\mathsf{F})\end{array}$$

Algorithm 17 *TraceFailTC*

```
Input: $\sigma = w^0_{obs} \cdots w^n_{obs}$, $A$
Output: debugSet
debugSet $\leftarrow \emptyset$
failR $\leftarrow \emptyset$
$C^* \leftarrow C$
$b \leftarrow$ true
while $b$ do
    $\phi^*_{obs} \leftarrow \phi^0 \wedge \ldots \wedge \phi^{n-1} \wedge (\bigvee_{(\varphi,\psi)\in C^*} (\varphi \wedge \neg\psi))\ [X\backslash X^{n-1}, X'\backslash X^n]$    ▷ Negated guarantees
    $\phi^*_{obs} \leftarrow \phi^*_{obs}[X^0_{\text{obs}}\backslash w^0_{obs}, \ldots, X^n_{\text{obs}}\backslash w^n_{obs}]$    ▷ Substitute by observed values
    $((w^0_H, \ldots, w^n_H), b) \leftarrow$ sat$(\phi^*_{obs})$    ▷ Check satisfiability
    if $b$ then
        $\tau \leftarrow (w^0_{obs} \cup w^0_H) \cdots (w^n_{obs} \cup w^n_H)$    ▷ Build execution trace
        for all $c = (\varphi, \psi) \in C^*$ do
            if $(w^{n-1}_{obs} \cup w^{n-1}_H, w^n_{obs} \cup w^n_H) \models \varphi[X\backslash X^{n-1}, X'\backslash X^n] \wedge \neg\psi[X\backslash X^{n-1}, X'\backslash X^n]$ then
                failR $\leftarrow$ failR $\cup \{r \mid c \in \rho(r)\}$;    ▷ Collect violated contracts
                $C^* \leftarrow C^*\backslash\{c\}$
            end if
        end for
        debugSet $\leftarrow$ debugSet $\cup \{(\tau, \text{failR})\}$
        failR $\leftarrow \emptyset$
    end if
end while
return debugSet
```

Requirements r_0 and r_5 cannot be violated in the last step of this test execution. We note that accessing the faulty 2-buffer implementation I from Algorithm 16, the debugging pair $(\tau_1, \{r_4\})$ would enable us to exactly localize the error and trace it back to the violation of the requirement r_4. □

For requirement interfaces with hidden variables, the underlying implementation is only partially observable. The best that the tracing procedure can do when the execution of a test leads to the **fail** verdict is to complete missing hidden variables with valuations that are consistent with the partial observations of input and output variables. It follows that the debugSet consists of "hints" on possible violated requirements and the causes of their violation. We note that Algorithm 17 attempts at finding the right compromise between minimizing the amount of data presented to the designer, while still providing useful information. In particular, it focuses on implementation errors that occur at the time of the failure, for both the hidden and the output variables. We note that in some faulty implementations, errors in updating hidden variables may not immediately result in observable faults. For instance, in the execution

$$\tau_3 \quad = \quad \begin{array}{l} (\mathsf{enq}, \mathsf{deq}, k = 1, \mathsf{E}, \neg\mathsf{F}) \\ (\mathsf{enq}, \neg\mathsf{deq}, k = 1, \neg\mathsf{E}, \neg\mathsf{F}) \\ (\mathsf{enq}, \neg\mathsf{deq}, k = 1, \neg\mathsf{E}, \neg\mathsf{F}) \end{array}$$

the requirement r_0 is immediately violated in the initial step, but the implementation errors are only observed in the last step of the test execution. Algorithm 17 does not give such executions as possible causes that lead to a **fail** verdict. It is a design choice – we believe that choosing hidden variables without any restriction would result in executions that are too arbitrary and have little debugging value.

8.2.4 Implementation

We implemented a prototype that contains both our test-case generation framework and the consistency check. The prototype was added to the model-based testing tool family MoMuT and goes by the name of MoMuT::REQs. The implementation uses the programming language Scala 2.10 and Microsoft's SMT solver Z3 [73] V4.3. The tool implements both *monolithic* and *incremental* approaches to test-case generation, however it does not produce output monitors, but only sequences of inputs and outputs. Using the input sequences, one can use the requirement interface as a monitor. For deterministic systems, the output sequence can be used directly.

The tool consists of 27 classes, and about $10,000$ lines of code, though that covers some additional functionalities that are not discussed in this thesis as well. It offers a command line interface and a graphical user interface, that allows to select the contracts that should be used for the consistency check and test-case generation. Additionally, it allows to load informal requirement interfaces from Microsoft Excel files, and formalize them within the graphical user interface, storing a linking to the formalized requirements. We will demonstrate the tool on the buffer example and an industrial case study in Chapter 11. The tool was implemented by the author of this thesis and Stefan Tiran from Graz University of Technology and the Austrian Institute of Technology.

8.3 Model-Based Mutation Testing

In this section we show how to apply model-based mutation testing (see Section 1.4) to requirement interfaces. We generate a test suite covering a set of fault models, that are specified via a set of *mutation operators*. These apply specific faults to all applicable parts of the requirement interface. When injected into requirement interfaces, we mutate one contract at a time. Then we check for refinement between the original requirement interface and the mutated one. If the mutated requirement interface can produce controllable variable values that are forbidden by the original, the conformance is violated. In that case we produce a test case leading exactly to that violation. If that test case is executed on a deterministic SUT and passes, we can guarantee that the corresponding fault was not implemented. Thus, by generating all tests for all fault models, we can proof the absence of all corresponding faults in the system.

Definition 8.1
We define a *mutation operator* μ as a function $\mu : C \to \mathcal{P}(C)$, which takes a contract $c = (\varphi, \psi) \in C$ and produces a set of mutated contracts $C^{\mu} \subseteq C$, where a specific kind of fault is applied to all valid parts of ψ. We only consider mutations in the guarantee, as the fault models should simulate situations where the system produces wrong outputs, after receiving valid inputs.

We currently consider the following fault models:

1. *off-by-one:* mutate every integer constant or variable, both by adding and subtracting 1,
2. *negation:* flip every Boolean constant or variable,
3. *change comparison operators:* replace equality operators by inequality operators, and vice versa; replace every operator in $\{\leq, <, >, \geq\}$ by each of the operators in $\{\leq, <, =, >, \geq\}$.
4. *change and/or*: replace every and operator by an *or* operator, and vice versa,
5. *change implication/bi-implication*: replace every implication by a bi-implication, and vice versa;

Definition 8.2
A *mutant* $c_m = (\varphi, \psi_m) \in C^{\mu}$ is an intentionally altered (mutated) version of the contract $c = (\varphi, \psi)$. A mutant is called a *first order* mutant, if it contains only one fault. In this thesis we only consider first-order mutants.

If a mutation does not introduce new behavior to the requirement interface, or leads to an inconsistency, we consider the corresponding mutant *equivalent*. Inconsistent mutants are considered equivalent, because the mutation can never be reached if it contradicts any of the other contracts. Thus, by performing consistency checks on the mutants, these mutants can be ignored, reducing the complexity of the test-case generation procedure.

Given the contract $(\varphi, \psi) \in C$, we denote by $\bar{C}_{(\varphi,\psi)}$ the set of the other contracts in the requirement interface, i.e. $\bar{C}_{(\varphi,\psi)} = C \setminus \{(\varphi, \psi)\}$, by $C^{\mu}_{(\varphi,\psi)}$ the set of mutants obtained by applying all mutation operators to (φ, ψ) and by $c_m = (\varphi, \psi_m)$ one single mutant in $C^{\mu}_{(\varphi,\psi)}$.

c_m is a non-equivalent mutant, if there exist two valuations v, v' so that:

- v is reachable from $\hat{v}$
- $(v, v') \models \varphi$
- $\bigwedge_{(\bar{\varphi},\bar{\psi})\in\bar{C}}(v, v') \models (\bar{\varphi}, \bar{\psi})$
- $(v, v') \models (\varphi, \psi_m) \wedge \neg(\varphi, \psi)$

Thus, a mutants is non-equivalent, if there exists a reachable variable valuation v, for which there exists a following variable valuation v' that satisfies all contracts that are not mutated, satisfies the mutated contract, but does not satisfy the original contract. Contract c_m from Example 7.3 is an equivalent mutants, because it introduced inconsistency, and all valuations v' that satisfy c_m do not satisfy the other contracts. Below we will give an example of a non-equivalent mutant.

Using Definition 7.4, one can also express non-equivalent mutants via refinement. A mutant is considered non-equivalent, if the requirement interface with the mutated contract does not refine the original requirement interface. We consider a mutant *k-equivalent* to the original requirement interface, if it is equivalent up to a bound k.

The test purpose Π for detecting (φ_m, ψ_m) can be encoded by the formula

$$\Pi = \varphi \wedge \psi_m \wedge \neg\psi \wedge \bigwedge_{(\bar{\varphi},\bar{\psi})\in\bar{C}} (\bar{\varphi} \Rightarrow \bar{\psi})$$

The reachability formula for such a test purpose differs from the one presented in Section 8.2 in two details: in the last step of the test case (the step supposed to reach the test purpose), the transition relation does not hold, as we require the original contract to be violated. Additionally, a test purpose is a relation over primed and unprimed variables.

A test purpose Π can be reached in step k, if

$$\exists X^0, \ldots, X^k.\, \phi^0 \wedge \ldots \wedge \phi^{k-1} \wedge \Pi[X \backslash X^{k-1}, X' \backslash X^k],$$

where, as in Section 8.2, $\phi^0 = \hat{\phi}[X' \backslash X^0]$ and $\phi^i = \phi[X' \backslash X^i, X \backslash X^{i-1}]$ represent the transition relation of A, instantiated for the i-th step. If the test purpose is reachable, the mutation is not k-equivalent.

Remark: in our synchronous studies, we consider weak mutation testing [103]. This means that a wrong value of internal variables is already considered a conformance violation. In contrast, strong mutation testing, as applied in our part on asynchronous systems (see Chapter 3) also requires that an internal fault propagates to an observable failure. The encoding of the reachability of the mutation as a test purpose, without altering the step relation, is only possible for weak mutation testing. Strong mutation testing would require the step relation to use the mutated contract in all steps, and then detect the failure in the last step. Due to considering weak mutation testing, we also weaken the definition of a test purpose compared to the definition in Section 8.2, enabling it to use internal variables.

Contrary to the previously defined test purposes, the test purposes in model-based mutation testing lead to *negative* counter examples, that is, counter examples steering towards an incorrect state. However, as defined in Section 8.2, we only extract the input vector σ_i, which is then combined with the correct requirement interface, to form a positive test case.

Often, different mutations of a contract will generate different negative counter examples, but those tests might then combine into the same positive test case. However, if the different mutations require different inputs to be enabled, they will also produce different positive test cases.

Example 8.7. Consider the requirement interface A_{beh} for the behavioral view of the 2-bounded buffer, as presented in Example 7.1. Let $c_{2,m} : \neg\mathsf{enq}' \wedge \mathsf{deq}' \wedge k > 0 \ \vdash \ k' = k - 2$ be a mutant of c_2, where $k' = k - 1$ was mutated to $k' = k - 2$. The test purpose to detect this mutation is

$$\Pi = \varphi_{c_2} \wedge \psi_{c_{2,m}} \wedge \neg\psi_{c_2} \wedge \bigwedge_{i \in \{0,1,3,4,5\}} (\varphi_{c_i} \Rightarrow \psi_{c_i})$$

The test purpose is not valid in the initial state, as the assumption requires k to be greater than 0. Thus a corresponding test case needs to execute at least one enqueue operation, before the mutated dequeue functionality can occur. The shortest vector $\bar{\sigma}$ leading to the test purpose is $\bar{\sigma}[0] = (\mathsf{enq}, \mathsf{deq}, k = 0, E, \neg F)$, $\bar{\sigma}[1] = (\mathsf{enq}, \neg\mathsf{deq}, k = 1, \neg E, \neg F)$ and $\bar{\sigma}[2] = (\neg\mathsf{enq}, \mathsf{deq}, k = -1, \neg E, \neg F)$. We extract the input vector σ_I so that $\sigma_I[0] = (\mathsf{enq}, \mathsf{deq})$, $\sigma_I[1] = (\mathsf{enq}, \neg\mathsf{deq})$ and $\sigma_I[2] = (\neg\mathsf{enq}, \mathsf{deq})$. Now we can build the positive test case, by applying the correct step relation, thus gaining $\sigma[0] = (\mathsf{enq}, \mathsf{deq}, k = 0, E, \neg F)$, $\sigma[1] = (\mathsf{enq}, \neg\mathsf{deq}, k = 1, \neg E, \neg F)$ and $\sigma[2] = (\neg\mathsf{enq}, \mathsf{deq}, k = 0, E, \neg F)$.

As a second mutant, consider $c_{3,m}$: true $\ \vdash \ k' = 0 \Leftrightarrow \neg\mathsf{E}'$, where E' is mutated to $\neg E'$. In this case, the test purpose is not reachable, as the initial contract c_0 requires both $k' = 0$ and E', which causes an inconsistency with the mutated contract. Thus, $c_{3,m}$ could be discarded after running a consistency check.

However, consider another mutant of c_3, $c_{3,m'}$: true $\ \vdash \ k' = 0 \implies \mathsf{E}'$, that changes the bi-implication to an implication. In this case, the mutated contract can be enabled after an enqueue in the first step, when $k' = 1$ and thus the left hand side of the implication is false, allowing E' to take any value. Thus, any vector of length two that starts with an enqueue operation, e.g., $\bar{\sigma}[0] = (\mathsf{enq}, \mathsf{deq}, k = 0, E, \neg F)$, $\bar{\sigma}[1] = (\mathsf{enq}, \neg\mathsf{deq}, k = 1, E, \neg F)$ detects the mutation. This example shows, that for different mutations on the same contract, the test generation results in different outcomes. □

8.4 Restrictions

We would now like to summarize some of the restrictions of our approach:

- *State-space explosion.* While the test-case generation, which is a reachability analysis, performs rather well, the complexity of the consistency check grows exponentially, both with the size of the model and with the depth of the search. As will be shown in the next section, the consistency check can only be applied to very low depths (smaller than five), for small to medium sized examples. This can be dodged, by manually defining important variable valuations, and performing the consistency check with these valuations as starting points, but it definitely hinders the fully automatic analysis of models. As for the test-case generation, the synchronous approach gains an important advantage compared to asynchronous methods, by performing several inputs and outputs at once, and thus needing far shorter traces. Yet, it also eventually reaches a point where it becomes infeasible, if the search depth or the size of the model is increased too high.
- *Undefined behavior.* During modeling with requirement interfaces, it happens fairly easily to accidentally leave some parts of the behavior undefined. If there exists a variable valuation, where

for a specific internal or output variable x there is no contract enabled that restricts the behavior of x, x may take any value during that step. Consider the buffer example (Example 7.1), where contract c_5 specifies the intended behavior of a simultaneous enqueue and dequeue operation. If we had forgotten to specify that contract, the counter k might take any value if both operations were performed at once. While this is the intended behavior, it takes some time to get used to this behavior, and the behavior makes it likely to introduce faults into the specification.

- *Weak mutation testing.* One of the main restrictions of the current model-based mutation testing approach with requirement interfaces is the restriction to weak mutation testing. Complex errors that only trigger an observable fault several steps after the mutation takes effect are not guaranteed to be detected. That is, only if another fault generates a test case that incidentally also detects the complex fault, the fault will be found. Changing to strong mutation testing would need the mutant and the original model to be executed independently, which can not be done by expressing the mutation via a test purpose anymore.
- *Coarse modeling.* If too much of the functionality is modeled in one contract, or the assumption of one contract is very weak, model-based mutation testing may produce only one test case, for situations that can occur several times at different internal states. Strictly spoken, the mutations we apply are not first-order mutations anymore, since any mutation of a contract with assumption true may cause a bug in any internal state. Thus, to apply model-based mutation testing, we recommend very fine grained modeling with restricting assumptions, which will both improve traceability and the quality of the generated test suite.

9 Tool Integration

Parts of this chapter are based on our publication at QSIC 2014 [12].

In this chapter we will give a short overview on how the tool MoMuT::Reqs 8.2.4 was integrated into a larger tool chain, to show how such an integration benefits a typical development process. We will also discuss one run through such a process, illustrating the individual steps on one of our industrial use cases, a safing engine of an airbag chip. More details on the safing engine will be given in Chapter 11.

The development process of safety-critical systems contains various different steps, including requirement formalization, modeling, analysis, test-case creation, development and test-case execution. The appropriate integration of the methods and tools coming from these complementary fields would greatly improve the quality of the design process. However, requirements engineering, static verification and testing are studied by different communities that insufficiently exchange their ideas and experiences. In addition to cultural differences between these communities, isolated development of methods and tools prevents their effective integration into a unified framework. The European ARTEMIS project MBAT (see Section 1.7.1) addressed this problem and aimed at integrating different requirements engineering, analysis and testing tools into the so-called MBAT Reference Technology Platform (MBAT RTP) [167].

Following the MBAT vision, we develop a novel requirements-driven analysis and testing framework. The framework, illustrated in Figure 9.1, unifies and integrates methods and tools from requirements engineering, model checking, and model-based testing. In our framework, the starting point is the document containing textual customer requirements. The role of the requirements document in our framework is twofold – it is used to derive both the implementation of the system and its formal abstract model (specification). These tasks are usually separately done by two different teams. Requirement formalization tools are used to support the engineer in developing a formal model of the system from its informal requirements. The consistency checker tool, based on model checking techniques, is used to analyze whether the formal model admits at least one correct implementation. The failing result of this

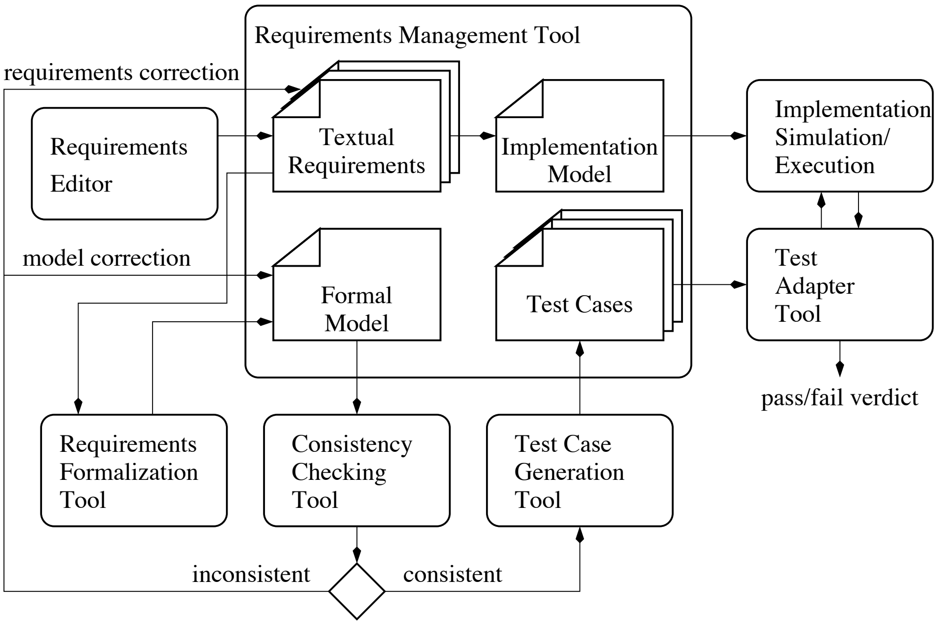


Figure 9.1: Overview of the generic framework.

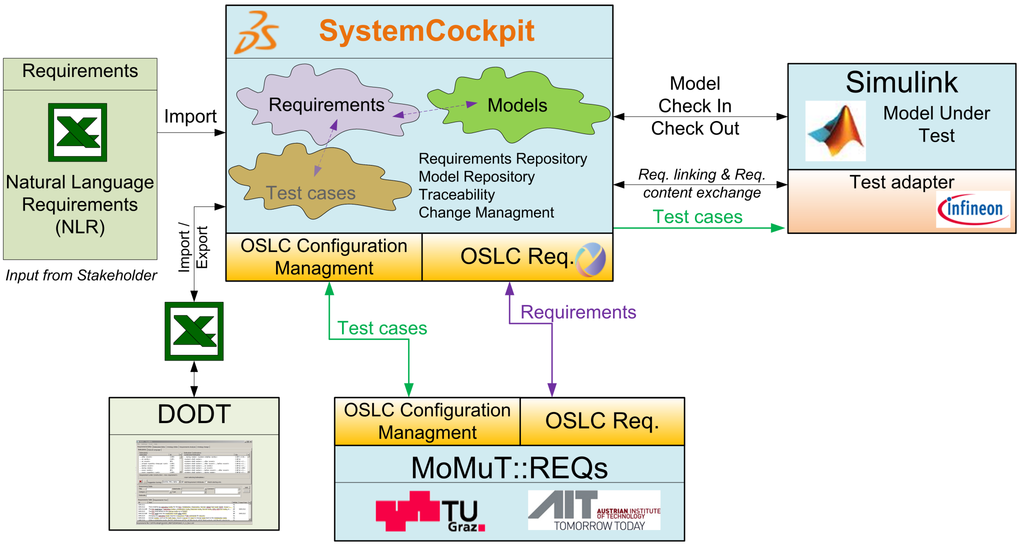


Figure 9.2: Overview of the framework instantiation.

check indicates inconsistency in the requirements from which the formal model is derived. In that case, the requirements must be corrected and the process repeated. Alternatively, the inconsistency may also result from poor requirement formalization. In that case, it is the formal model that needs correction. The test-case generation tool creates a test suite from the consistent formal model. The test cases from the suite are executed on the system implementation via the test execution tool. All the artifacts created during the workflow, that is the (informal) textual requirements, formal models, test cases and implementation models, are handled by an integrated tool providing requirements management as well as an artifact repository.

We instantiate the framework with specific tools, shown in Figure 9.2. First, we receive the requirements document, which is stored in a Microsoft Excel file and imported into System Cockpit. Then, we structure the informal requirements into boilerplates with the support of the tool DODT from Infineon Austria [84]. Boilerplates are reusable text structures, that provide a semi-formal structure for often-used sentences. They consist of fixed keywords and boilerplate attributed between them, that can e.g. be variable identifiers. DODT also compares the name of variables to indicate possible typos. By applying DODT, we gain a reduction of spelling mistakes, poor grammar and ambiguity. The resulting semi-formal requirements are transformed into formal *requirement interface* models that specify the intended behavior of the system. The MoMuT::REQs tool is used to (1) check the consistency of the formalized requirements; and (2) automatically generate test cases from requirement interfaces. We use MathWorks Simulink [138] to develop the system implementation model, which we derive directly from the textual customer requirements. The test cases generated by MoMuT::REQs are executed on the Simulink implementation model via a test adapter developed by Infineon Austria. The artifacts created in this workflow, including textual requirements, formal test and implementation models and test cases are managed in the SystemCockpit tool developed by Dassault Systèmes [70]. This commercial requirements management tool facilitates tracing formalized requirements, implementation model elements and test cases to the original textual customer requirements.

The integration of MoMuT::REQs and SystemCockpit is achieved via the MBAT Interoperability Specification (IOS) and the Open Services for Lifecycle Collaboration (OSLC) [139]. OSLC is a standard for linking systems and software engineering tools via web services, where all work products are stored as OSLC ressources, and can be stored, retrieved and changed via unified commands. The OSLC integration preserves the generic nature of our framework. In particular, it facilitates replacement of a specific tool used in the framework instantiation with another one with the compatible functionality, as long as both comply with the OSLC standards. More details on OSLC will be given at the end of this chapter. For instance, the requirements management tool SystemCockpit could be replaced by DOORS [101] without additional effort. The integration of the other tools is currently done via a file repository within SystemCockpit, where the work products are already stored and linked as OSLC resources.

We will now illustrate the complete approach on the safing engine use case presented in Section 1.8.2 and Section 11.2.2.

9.1 From Textual to Formal Requirements

This section covers the second step in our methodology: requirement formalization. The first step, storing the requirements from the Excel sheet into SystemCockpit is skipped due to simplicity. The formalization is implemented via the tools DODT and MoMuT::REQs. We now present a subset of the airbag system's textual customer requirements, and then describe how these informal requirements are structured and finally formalized. The safing engine of the airbag is responsible for double-checking, whether the airbag should be released. It consists of a state machine that consists of seven states and several input signals, that trigger different reactions and state changes, depending on the current state. Airbag deployment is only possible in some selected states. More details are given in Section 11.1.1.

9.1.1 Textual Customer Requirements

The specification of the state machine consists of 39 textual requirements, that were provided by the customer of Infineon. We illustrate our framework with three sample requirements from this specification.

***R1*:** There shall be seven operating states for the safing engine: RESET state, INITIAL state, DIAGNOSTIC state, NORMAL state, TEST mode, SAFE state and DESTRUCTION state.

***R2*:** The safing engine shall change per default from RESET state to INIT mode.

***R3*:** On a reset signal, the SE shall enter RESET state and stay while the reset signal is active.

9.1.2 Structuring Textual Requirements

We use the DODT tool to structure the informal requirements and semi-formalize them, in order to achieve a textual representation that relies on the same vocabulary and uses code words that provide additional semantics. This is done with a six step process that transforms natural language (NL) requirements into boilerplates (see Figure 9.3), that were developed within the ARTEMIS CESAR project [144]. Boilerplates facilitate the reduction of spelling mistakes, poor grammar or ambiguity. Additionally they help to capture the underlying system-related semantics.

In the first step the relevant sentences are selected. Thereafter, the typographic errors are corrected by using the domain ontology concepts as a dictionary. The tool highlights variable names that only occur once, or are very close to other used names. In the above example, it highlights *INIT mode* because it is called *INITIAL state* in the rest of the requirements document and *SE*, that should be called safing engine. Now requirements are transformed to find matching boilerplates. User-defined substitution rules are applied to replace expressions (like "will" or "must") with synonymous fixed syntax elements (like "shall"). In the next step, the text between fixed syntax elements is assigned to boilerplate attributes and several adjacent attributes are split. Through these steps *R1* to *R3* were transformed into semi-formal requirements as depicted in Figure 9.4.

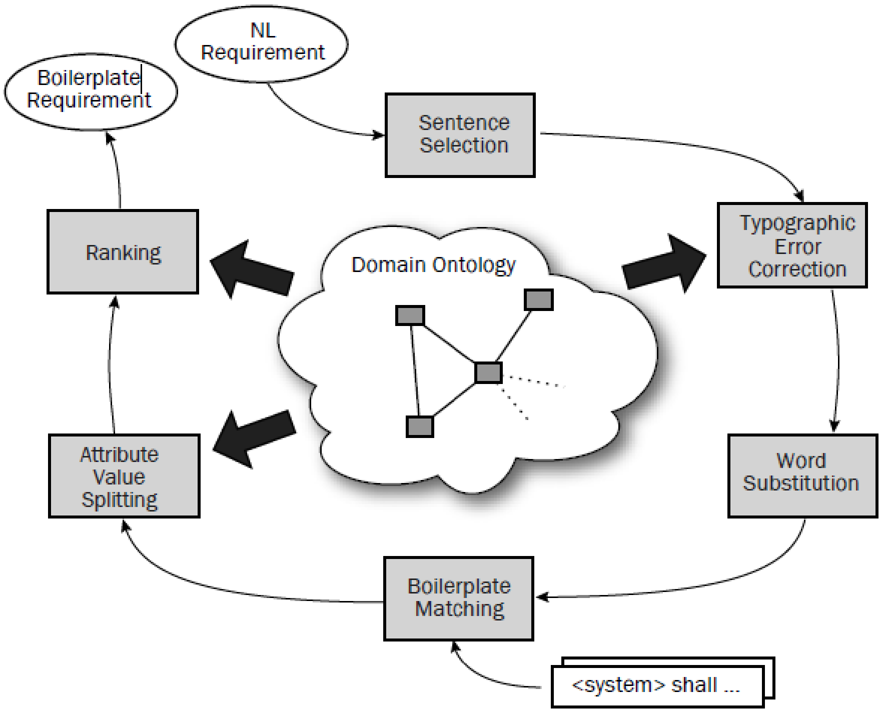


Figure 9.3: Six-step Semi-Automated Conversion Process.

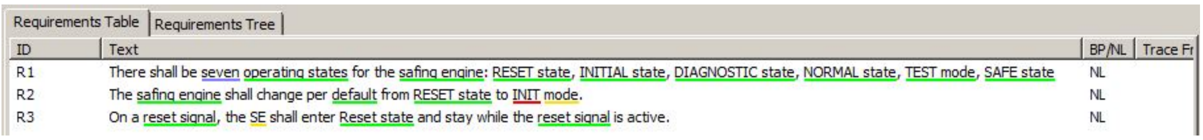
9.1.3 Formalizing Requirements

The semi-formal requirements are stored back to SystemCockpit and from there they can be accessed by MoMuT::REQs, to be formalized as *requirement interfaces*, as introduced in Chapter 7.

The formalization is done manually, and the 39 safing engine requirements from the case study were refined to 32 formal requirements. These formal requirements contain 11 input variables, 5 output variables and 11 hidden state variables. We now illustrate the formalization of textual customer requirements *R1*, *R2* and *R3* into a requirement interface. The interface has a single Boolean input variable *reset* and a single output variable *state*. It consists of three contracts *FR1*, *FR2* and *FR3*, that are associated to textual requirements *R1*, *R2* and *R3*, respectively. The resulting contracts in the requirement interface are shown in Listing 9.1. Note that in the tool, we denote the assumption by the keyword *assume* and the guarantee by the keyword *guarantee*, instead of delimiting them by ⊢.

9.2 Consistency Checking of Formal Requirements

The consistency check is also done in MoMuT::REQs, and was already discussed in Section 8.1. We applied it to the formalized requirements and found that they are inconsistent. The tool reported the



ID	Text	BP/NL	Trace Fr
R1	There shall be seven operating states for the safing engine: RESET state, INITIAL state, DIAGNOSTIC state, NORMAL state, TEST mode, SAFE state	NL	
R2	The safing engine shall change per default from RESET state to INIT mode.	NL	
R3	On a reset signal, the SE shall enter Reset state and stay while the reset signal is active.	NL	

Figure 9.4: Highlighting of variables and fixed syntax elements in DODT.

```
FR1: assume \textrm{true}
     guarantee {\sf state}'={\sf RESET} or {\sf state}'={\sf INIT}
               or {\sf state}'={\sf DIAG} or {\sf state}'={\sf NORM}
               or {\sf state}'={\sf TEST} or {\sf state}'={\sf SAFE}
               or {\sf state}'={\sf DESTR}

FR2: assume {\sf state}={\sf RESET}
     guarantee {\sf state}'={\sf INIT}

FR3: assume {\sf reset}'
     guarantee {\sf state}'={\sf RESET}
```

Listing 9.1: Requirement interface contracts formalizing the textual customer requirements *R1*-*R3*.

minimal conflict set $\{FR2, FR3\}$. The minimal conflict set provides useful debugging information to the designer that enables easier identification of the source and causes of the conflict. A conflict can arise either from a poor specification of the textual customer requirements or from their incorrect formalization. In our example, the conflict is due to the following scenario. When the reset signal is triggered while the system is in the reset state, *FR2* requires the system to be in the initial, while *FR3* requires it to be in the reset state in the next step.

```
FR2.1: assume {\sf state}={\sf RESET} and not {\sf reset}'
       guarantee {\sf state}'={\sf INIT}
FR2.2: assume {\sf state} = {\sf RESET}
       guarantee {\sf state}' = {\sf RESET} or {\sf state}' = {\sf INIT}
```

Listing 9.2: Repairing the inconsistency of *FR2*.

The conflict can be resolved in two ways, both illustrated in Listing 9.2. The first resolution consists in changing the textual customer requirement *R2* and adapting its formalization accordingly. In fact, *R2* is a potentially ambiguous requirement because it describes the correct system behavior when it is in the reset state but does not refer to the reset input signal at all. It makes sense that a reset signal is never triggered when the system is already in the reset state, hence we make this assumption explicit in *R2* and in its formalization *FR2.1*. On the other hand *FR2.2* corrects *FR2* by making a more direct translation of the original textual customer requirement *R2*. In fact, *R2* does not explicitly define the conditions required for making a transition from the reset to the initial state. We correct *FR2* by replacing it with *FR2.1*, making this observation explicit – the update of the next state is chosen non-deterministically. In this scenario, the designer needs to ensure that there exists another requirement in the specification that defines more precisely when the state change is allowed to occur.

This example shows that choosing the right conflict resolution of inconsistent requirements may not be a straight-forward task. The corrections may involve changes in either textual customer requirements, formalized requirements or both. This process requires interactions between the verification engineers and the supplier that provides the textual customer requirements in order to identify the exact causes of conflicts and to find a solution that satisfies the intended meaning of the specification. After repairing the inconsistency, the tool reports that all formalized requirements are consistent, as illustrated in Figure 9.5.

9.3 Generating Tests from Formal Requirements

Test cases are generated by MoMuT::REQs, according to the theory discussed in Section 8.2, using manually defined test purposes or generating them via model-based mutation testing. They encode a sequence of input vectors and constraints relating input, output and hidden valuations that the system must satisfy and that are defined by the requirement interface. In the case of deterministic specifications without hidden variables, a test case has a simpler representation in the form of a sequence of input and output valuations. MoMuT::REQs test cases are not adaptive – the tool fixes the input vector and a correct output vector and one can use the requirement interface specification as a *monitor* (see Section 8.2 and Algorithm 13) that observes the outputs of the implementation model to decide whether its execution satisfies or violates the specification. Such a test case guarantees that for an arbitrary input vector, the resulting **pass/fail** verdict is correct. On the other hand, non-adaptive test cases cannot guarantee that the test purpose is always reached, due to the potential implementation freedom allowed by the output constraints in the specification.

We generate test cases from the requirement interfaces that formalize textual customer requirements of the safing engine case study. The safing engine specification does not contain non-determinism in output or hidden variables. It follows that the input/output sequences produced by MoMuT::REQs can be used directly for testing the system.

Listing 9.3 shows the test case that specifies the correct transition from the reset to the initial state, as defined by the specification. We simplify the presentation of the test case by projecting away variables that do not play a role in this scenario.

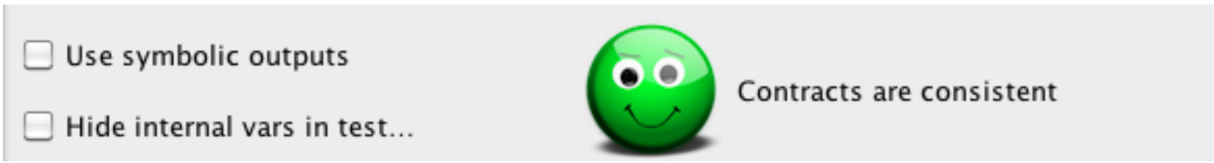
```

STEP 0
INPUT {\sf spi_command} = {\sf NONE}
INPUT {\sf reset1} = \textrm{true}
INPUT {\sf overvoltage} = \textrm{false}
OUTPUT {\sf safe1} = \textrm{false}
OUTPUT {\sf efo2} = \textrm{false}
HIDDEN {\sf operating_state} = {\sf RESET}

STEP 1
INPUT {\sf spi_command} = {\sf NONE}
INPUT {\sf reset1} = \textrm{false}
INPUT {\sf overvoltage} = \textrm{false}
OUTPUT {\sf safe1} = \textrm{false}
OUTPUT {\sf efo2} = \textrm{false}
HIDDEN {\sf operating_state} = {\sf INIT}
```

Listing 9.3: Two steps of a testcase leading to to the Init state.

9.4 Executing Tests on the Implementation Model

For the airbag system case study, the implementation, simulation and execution of our implementation model was realized by Infineon via MATLAB Simulink. The test cases generated in the previous section are executed on this model.

An extract of the Simulink model is shown in Figure 9.6. It shows the internal state machine of the safety engine. To execute the abstract test cases on the implementation model, a test adapter was implemented. It translates the abstract input labels, such as *reset* or *spi_command*, into the actual input signal implemented in the system. The outputs produced by the implementation model are then transformed



□ Use symbolic outputs

□ Hide internal vars in test...

Contracts are consistent

Figure 9.5: MoMuT::REQs reporting that the overhauled requirements are consistent.

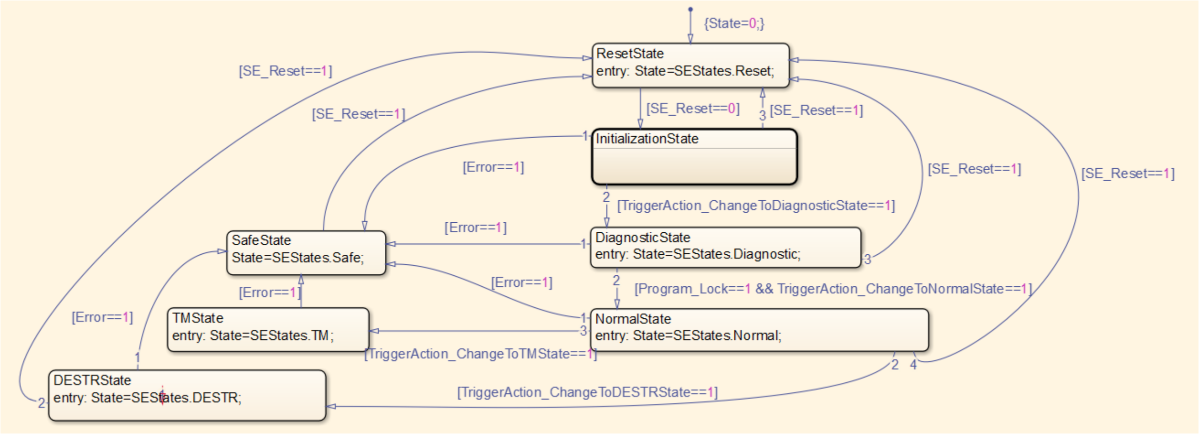

{State=0;}
ResetState
entry: State=SEStates.Reset;
[SE_Reset==1]
[SE_Reset==1]
[SE_Reset==0]
[SE_Reset==1]
InitializationState
[Error==1]
[SE_Reset==1]
[SE_Reset==1]
[TriggerAction_ChangeToDiagnosticState==1]
[Error==1]
SafeState
State=SEStates.Safe;
DiagnosticState
entry: State=SEStates.Diagnostic;
[Error==1]
[Error==1]
[Program_Lock==1 && TriggerAction_ChangeToNormalState==1]
[Error==1]
TMState
entry: State=SEStates.TM;
NormalState
entry: State=SEStates.Normal;
[TriggerAction_ChangeToTMState==1]
DESTRState
entry: State=SEStates.DESTR;
[TriggerAction_ChangeToDESTRState==1]

Figure 9.6: The Simulink model of the state machine described in Section 11.2.2.

back to the abstract test results, which can then be validated by the test oracle. The test adapter does not only translate from one syntax (e.g. test case output) to another one (e.g. target language). It also adds time steps for every test step, to ensure that the iterations take enough time for all outputs to be produced. This is done via the workflow depicted in Figure 9.7. The generated abstract test cases are refined by the test adapter, using a database that stores additional information, like e.g. the timing constraints for the different outputs. The refined inputs are then executed on the Simulink model. The produced outputs are then again processed by the test adapter, which abstracts them, to compare them with the outputs expected by the test case. Then, according to whether the tests failed or passed, a verdict is passed to the user, containing additional information, like which test cases failed.

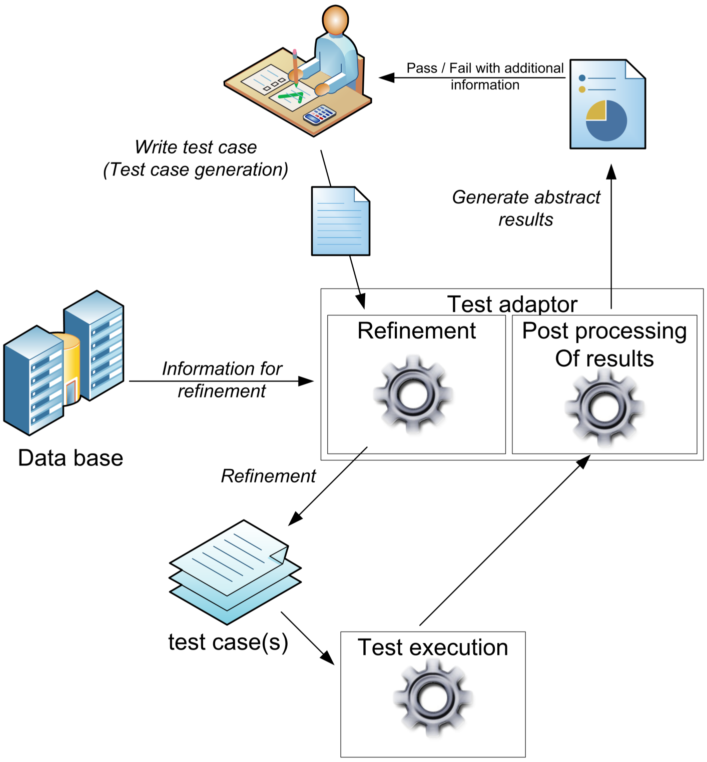


Figure 9.7: Test-case execution process.

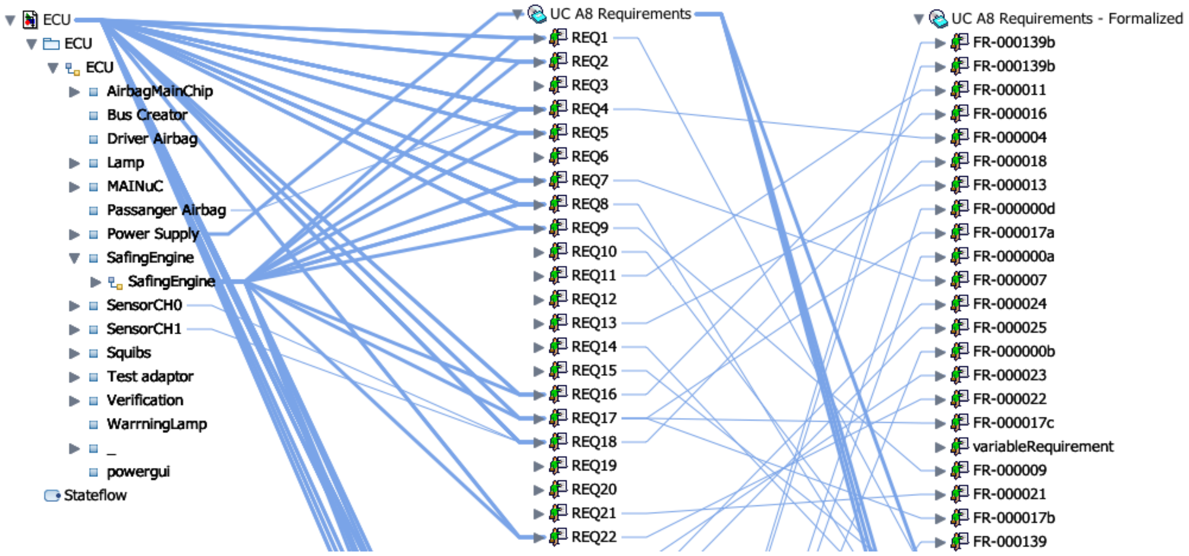


Figure 9.8: Traceability view of SystemCockpit, illustrating the links between customer requirements, formal requirements and the implementation model.

For the refinement of the abstract test cases, each signal is considered an object with assigned properties and methods. The methods assigned to the signals include the procedure to translate the signal from one to another abstraction level. During the translation, timing information is added and abstract test cases are translated into sequences. Information which is not included in the abstract test cases, but is important for the execution of the Simulink model (e.g. timing information) is retrieved from the product design specification.

The translated test cases are executed on the Simulink model by running the simulation using the newly configured signals. The simulation results cannot be directly compared to the test oracle on the higher abstraction level. Therefore the test adapter abstracts the results to match the higher abstract level.

Since the delay between inputs and outputs varies depending on the inputs, the test driver searches for the input that needs the longest delay until it is processed, and waits for this delay, before it reads the produced outputs.

9.5 Managing and Tracing Requirements, Models and Tests

The core component of our approach is a requirements management system implementing an OSLC provider. It is used as central repository not only for the natural language requirements but also for any model involved in the workflow as well as the traceability links between those elements. In our instance of the tool chain we use the tool SystemCockpit. It is an experimental platform initially developed within the ARTEMIS CESAR project and then continued in the ARTEMIS MBAT project. In the context of the MBAT project the concrete instance of OSLC RM is referred to as MBAT/IOS. This also includes extensions which enable services for navigation and creation of the traceability links using the modern web technologies HTML 5 and AJAX.

As depicted in Figure 9.1, the requirements tool is supposed to store and link the customer requirements, the semi formal requirements, the formal requirements, the test cases and the implementation model. SystemCockpit provides a way to build these IOS links via an internal web interface, as well as directly via the connected tools.

The navigation is done using tree and 2D graph representations of the models, in which the user can easily follow traceability links. The user interface can also display a traceability graph between user selected models. Figure 9.8 shows an example of such a graph: this view shows a partially expanded model, customer requirements and formalized requirements, illustrating the traceability links between the currently exposed elements. Bold links are pseudo links used to indicate that actual links exist between some elements within the sub-trees of the source or destination. As the user expands these sub-trees, these actual links are displayed. This provides a convenient visual help to perform coverage analysis.

9.6 OSLC Integration

OSLC is an open community that provides specifications for tool integration and communication. These standards facilitate easy data exchange as well as the option to replace tools with others that comply to the same OSLC sub-specification(s). In OSLC, data is linked via uniform resource identifiers (URIs), and can be accessed and modified through creating, retrieving, updating and deleting (CRUD) commands.

The OSLC standard defines two types of tools: OSLC providers and OSLC consumers. Providers are meant to store and provide data, providing the consumers easy access to browse, create and retrieve the data. One of the main advantages of SystemCockpit is its compliance to the OSLC standard, by operating as an OSLC provider. While it supports basic storage and access of work products via a file repository, it stores these work products as OSLC resources and enables the creating, retrieving, updating, deleting and linking of data via OSLC. All OSLC resources are equipped with a URI. Upon creation of a formal requirement in SystemCockpit via the tool MoMuT::REQs, it is immediately linked to its associated customer requirements by the *satisfies*-link, using the URIs of the requirements. A second link, as displayed in Figure 9.8, is created between the customer requirements and the implementation model parts.

MoMuT::REQs already uses this advanced communication as an OSLC consumer, by directly importing customer requirements, exporting the formalized requirements and linking them, all via OSLC. Excel and DODT use simple file import/export for the communication to SystemCockpit, and create the corresponding URIs manually.

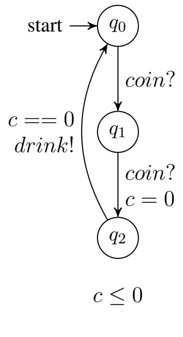
10 Real-Time Requirements

In this chapter, we will discuss the applicability of requirement interfaces to real-time systems. First, in Section 10.1, we will discuss how synchronous systems are usually designed in a way that completely neglects timing, and considers all outputs to be produced instantaneously. Then, in Section 10.2, we will show how such systems can be combined with clocks, to enable time-triggered outputs, using discrete time delays. We will first show a notion that consumes one time unit per synchronous step through the requirement interface, and then show how symbolic time delays can improve the test-case generation.

10.1 Time in Synchronous Systems

Synchronous languages, like Esterel [52] or Lustre [64], were originally designed with the goal of simplifying the specification of real-time systems. In asynchronous specifications, inputs are received and processed, and only afterwards outputs are generated. Contrary to that, synchronous systems are considered to behave according to the *synchrony hypothesis*, stating that outputs are produced simultaneously to receiving inputs [52]. This notion facilitates to completely neglect time in the specification, and is very well suited for embedded systems and circuit design. Consider the example in Figure 10.1. On the left side, we have an asynchronous timed automata specification of a beverage vending machine, that receives *coin?* as input, and triggers the release of the desired *drink!* as output, immediately after receiving the second *coin?* input. Even though the invariant enforces a zero second delay between the last input and the output, the two signals are considered consecutively. Additionally, this zero second delay needs to be specified, enlarging the complexity of the example. On the right side, we show the same example modeled via contracts of a requirement interface, where *coin* is a Boolean input, *drink* is a Boolean output and *counter* is an internal integer variable. Simultaneously to receiving the second *coin* signal, the output *drink* is immediately activated, without the need to introduce additional variables to restrict the timing, or adding any complexity.

In praxis, the synchrony hypothesis cannot really be met by any implementations, as there always is a slight delay during the calculation of the outputs. However, if the delay is small enough, so the implementation is able to react to all external events, before the next inputs arrive, the implementation is considered to satisfy the hypothesis. For a device operating according to clock cycles, this usually means that all outputs must be calculated within one clock cycle. Considering such a system, we thus only have to specify the behavior for each of the clock cycles, and can assume that everything is finished, before the next cycle starts, which significantly reduces the complexity of the specification.



$$\begin{aligned} c_0 &: \ coin' \wedge counter < 2 \ \vdash \ counter' = counter + 1 \\ c_1 &: \ coin' \wedge counter = 2 \ \vdash \ drink' \wedge counter' = 0 \end{aligned}$$

Figure 10.1: Asynchronous and synchronous specifications of a beverage vending machine.

During the experiments on the airbag safing engine, described in Section 11.2.2, we considered the synchrony hypothesis to hold. Thus, our requirement interfaces, and consequently also our test cases, did not take time into account. However, some of the output needed longer than one clock cycle to be produced. Additionally, the individual output signals needed different amounts of time, until they were generated and these times were only known to the engineers from Infineon.

We found three possible ways to apply these times during test execution:

- The easiest approach would have been to wait the maximal delay between each step of the test case, before retrieving the outputs and receiving the inputs. While this would take the minimal effort, it also would be the most coarse approach, and if any of the outputs with smaller delays would not meet their individual deadlines, it would go unnoticed.
- The most fine-grained approach would be to retrieve each output at the time of its personal deadline. This would take the most effort, as each of the outputs would be collected sequentially. Additionally, this would somehow break the synchrony, as the outputs would not be read in one step.
- For our experiments, we decided to choose a compromise between these two methods, and for each step of the test case, we searched for the maximal delay invoked by the subset of outputs that were actually produced in that specific step. At the corresponding time, we fetched all outputs at once and triggered the next inputs.

Handling these time delays in the test adapter, which has as access to a database containing all time constraints, and neglecting them in the specification, vastly reduced the complexity of our requirement interface and thus simplified test case generation. Yet we still gained meaningful test cases, and were able to extend them with the needed timing contraints.

The next section will discuss how the complexity rises if additional clock variables are added to the specification to measure the progress of time.

10.2 Time-Triggered Outputs

So far, we investigated how requirement interfaces enable the simple specification of systems that expect immediate reactions to inputs from the environment. In this section we will consider systems, where certain outputs are observed after a certain delay upon receiving the triggering inputs.

10.2.1 Explicit Ticks

We consider systems that are clocked by a single clock, and only consider immediate outputs or discrete time delays, with respect to that clock. Time progress is modeled via internal clock variables and an additional *tick*-contract, that increases the value of all clock variables in each iteration. The assumption of the tick-contract is true, and its guarantee is a conjunction of implications, stating for each clock variable, that if it is not reset in the current iteration, it is increased by one. The tick contract could be split into multiple contracts, to keep it more fine grained, if desired. Unfortunately, however, the implication cannot be split into assuming that the clock is not reset, and guaranteeing that the clock is increased by one, as that would need primed internal variables in the assumption

Example 10.1. We will illustrate this on the car alarm system, that was introduced in Section 1.8.1, analyzed several times in Part I (the part on asynchronous systems) and is illustrated as a timed automaton in Figure 5.1. It contains three timing constraints: 20 seconds after all doors were locked and closed, the alarm system shall be activated, 30 after the alarm was triggered, the sound shall stop, and 270 seconds after that, the flash shall stop as well.

Given these three delays, we use three clock variables. As the preconditions for the delays do not overlap, a single variable would also be sufficient, but this is not true in general, thus we show the more general way. We call the three clock variables $\mathsf{clk_a}$ (armed), $\mathsf{clk_f}$ (flash) and $\mathsf{clk_s}$ (sound). For each of them, we introduce a reset flag ($\mathsf{res_a}$, $\mathsf{res_f}$, $\mathsf{res_s}$), which is an internal Boolean variable, and can be used to reset the clocks.

The car alarm uses two Boolean inputs, closed and locked, six internal integer variables for handling the timing, $\mathsf{clk_a}$, $\mathsf{clk_f}$, $\mathsf{clk_s}$,$\mathsf{res_a}$, $\mathsf{res_f}$, $\mathsf{res_s}$, one internal Boolean variable silent and three boolean output variables, armed, sound and flash.

Contract c_1 shows the reset of the armed clock. When the doors are closed and locked, but were either not closed or not locked in the last iteration, the clock is reset. Contract c_2 shows the arming of the system, if the clock hits 20 while the doors are still locked and closed. Contract c_{tick} is the contract in charge of increasing all clocks, for which the reset flag is not true.

Note that the presented examples are simplified for presentational purposes. They only contain the contracts specifying the wanted behavior. For our models to be complete, we also need contracts prohibiting unwanted behavior. Consider for instance the armed variable: contract c_2 activates the arming when the clk_a variable hits 20. However, while the clock is smaller than 20, the behavior of the armed variable is undefined, thus enabling the requirement interface to arbitrarily turn it on or off. Consequently, we need to specify for every variable, that it is not supposed to change in all situations where we do not want it to change. This can be done automatically, by creating an additional contract per variable. The assumption of such a contract is the negation of the disjunction of the assumptions of all contracts that would change the corresponding variable. The guarantee simply ensures that the variable does not change. E.g., for the armed variable, the contract would look like this: $c_{armed} : \neg((\mathsf{closed}' \wedge \mathsf{locked}' \wedge \mathsf{clk_a} = 20) \vee (\mathsf{armed} \wedge \neg\mathsf{closed}') \vee (\mathsf{armed} \wedge \neg\mathsf{locked}')) \vdash \mathsf{armed}' = \mathsf{armed}$. MoMuT::REQs provides an option for building these constraints for all output and internal variables. After that, one can select which of the generated contracts shall be used for the test generation, so that if unspecified behavior of some variables is intended, it can be preserved.

The requirement interface for the car alarm system with three clocks is defined as: A^{cas1} : $\hat{C}^{cas1} = \{c_0\}$, $C^{cas1} = \{c_i \mid i \in [1,7]\} \cup \{c_{tick}\}$, $X_I^{cas1} = \{\mathsf{closed}, \mathsf{locked}\}$, X_O^{cas1} =$\{$armed, flash, sound$\}$ and $X_H^{cas1} = \{\mathsf{silent}, \mathsf{res_a}, \mathsf{res_s}, \mathsf{res_f}, \mathsf{clk_a}, \mathsf{clk_s}, \mathsf{clk_f}\}$ where

$$
\begin{array}{lcl}
c_0 & : & \neg\mathsf{res_a} \wedge \neg\mathsf{res_s} \wedge \neg\mathsf{res_f} \wedge \neg\mathsf{sound} \wedge \neg\mathsf{flash} \wedge \neg\mathsf{armed} \\
 & & \wedge\mathsf{clk_a} = 0 \wedge \mathsf{clk_s} = 0 \wedge \mathsf{clk_f} = 0 \wedge \neg\mathsf{silent} \\
c_1 & : & \mathsf{closed}' \wedge \mathsf{locked}' \wedge \neg\mathsf{flash}' \wedge \neg\mathsf{sound}' \wedge (\neg\mathsf{closed} \vee \neg\mathsf{locked}) \ \vdash \ \mathsf{res_a}' \\
c_2 & : & \mathsf{closed}' \wedge \mathsf{locked}' \wedge \mathsf{clk_a} = 20 \ \vdash \ \mathsf{armed}' \\
c_3 & : & \mathsf{armed}' \wedge \mathsf{closed} \wedge \neg\mathsf{closed}' \wedge \mathsf{locked}' \ \vdash \ \mathsf{sound}' \wedge \mathsf{flash}' \wedge \neg\mathsf{armed}' \wedge \mathsf{res_s}' \\
c_4 & : & \mathsf{armed} \wedge \neg\mathsf{locked}' \wedge \mathsf{closed}' \ \vdash \ \neg\mathsf{armed}' \\
c_5 & : & \mathsf{sound} \wedge \mathsf{flash} \wedge \mathsf{clk_s}' = 30 \ \vdash \ \neg\mathsf{sound}' \wedge \mathsf{flash}' \wedge \mathsf{res_f}' \\
c_6 & : & \neg\mathsf{sound} \wedge \mathsf{flash} \wedge \mathsf{clk_f}' = 270 \ \vdash \ \neg\mathsf{sound}' \wedge \neg\mathsf{flash}' \wedge \mathsf{silent}' \\
c_7 & : & \mathsf{sound} \wedge \neg\mathsf{locked}' \ \vdash \ \neg\mathsf{sound}' \wedge \neg\mathsf{flash}' \\
c_8 & : & \mathsf{silent} \wedge \mathsf{closed} \wedge \mathsf{locked}' \wedge \neg\mathsf{closed} \ \vdash \ \mathsf{armed}' \\
c_9 & : & \mathsf{silent} \wedge \mathsf{locked}' \ \vdash \ \neg\mathsf{silent}' \\
c_{tick} & : & \mathsf{true} \ \vdash \ \mathsf{res_a}' \implies \mathsf{clk_a}' = 0 \wedge \neg\mathsf{res_a}' \implies \mathsf{clk_a}' = \mathsf{clk_a} + 1 \wedge \\
 & & \mathsf{res_s}' \implies \mathsf{clk_s}' = 0 \wedge \neg\mathsf{res_s}' \implies \mathsf{clk_s}' = \mathsf{clk_s} + 1 \wedge \\
 & & \mathsf{res_f}' \implies \mathsf{clk_f}' = 0 \wedge \neg\mathsf{res_f}' \implies \mathsf{clk_f}' = \mathsf{clk_f} + 1
\end{array}
$$

Figure 10.2 shows the timing diagram of a trace that triggers the alarms of the car alarm system, where we set the delay for arming the system from 20 to 2, to simplify the presentation. In the inital state, the inputs are false, then, in the second time step, at the rising edge of the external clock, they are both turned on simultaneously. After they stayed activated for two time units, the system becomes armed. One step after that, the doors are opemed, and the alarm starts immediatly.

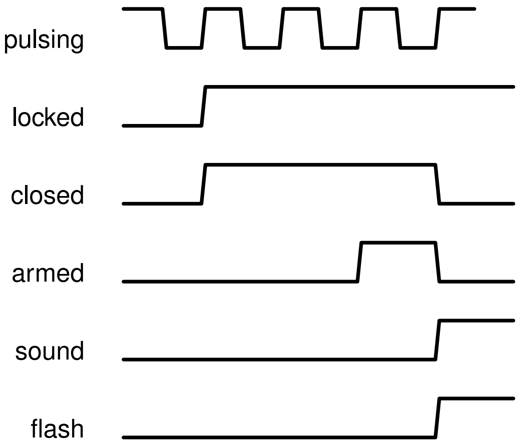


Figure 10.2: Timing diagram of a trace in the car alarm system.

To reduce the complexity of the interface, one may also assume one single clock, measuring the overall time in the system, without ever being reset. Then, for each time-triggered output, its expected time of arrival can be calculated, depending on the current system time in the iteration when the counting starts and the planned delay. We illustrate this with Example 10.2.

Example 10.2. A car alarm system modeled with only one such clock is illustrated below. Contract c_1 shows the calculation of the *time of arrival* of the armed output, twenty seconds after the current time when the doors were closed and locked. Contract c_2 shows how the armed output is set to true, once the current time hits the expected time of arrival, while the doors are still locked and closed. The tick contract only needs to update the single clock. Again, we need additional contracts to ensure that the time of arrival variables are not changed, while there is no contract enabled that restricts them.

The car alarm uses two Boolean inputs, closed and locked, and three boolean output variables, armed, sound and flash and one internal Boolean variable silent. This time, however, we only need four internal integer variables: current, toa_f, toa_s and toa_a.

The requirement interface for the car alarm system with a single clock is defined as: A^{cas2} : $\hat{C}^{cas2} = \{c_0\}$, $C^{cas2} = \{c_i \mid i \in [1,9]\} \cup \{c_{tick}\}$, $X_I^{cas2} = \{\mathsf{closed}, \mathsf{locked}\}$, $X_O^{cas2} = \{\mathsf{armed}, \mathsf{flash}, \mathsf{sound}\}$ and $X_H^{cas2} = \{\mathsf{silent}, \mathsf{toa}_\mathsf{a}, \mathsf{toa}_\mathsf{s}, \mathsf{toa}_\mathsf{f}, \mathsf{current}\}$ where

$$\begin{array}{lll}
c_0 & : & \mathsf{toa}_\mathsf{a} = 0 \land \mathsf{toa}_\mathsf{s} = 0 \land \mathsf{toa}_\mathsf{f} = 0 \land \neg\mathsf{sound} \land \neg\mathsf{flash} \land \neg\mathsf{armed} \land \mathsf{current} = 0 \land \neg\mathsf{silent} \\
c_1 & : & \mathsf{closed}' \land \mathsf{locked}' \land \neg\mathsf{flash}' \land \neg\mathsf{sound}' \land (\neg\mathsf{closed} \lor \neg\mathsf{locked})) \vdash \mathsf{toa}_\mathsf{a}{}' = \mathsf{current}' + 20 \\
c_2 & : & \mathsf{closed}' \land \mathsf{locked}' \land \mathsf{current}' = \mathsf{toa}_\mathsf{a}{}' \vdash \mathsf{armed}' \\
c_3 & : & \mathsf{armed}' \land \mathsf{closed} \land \neg\mathsf{closed}' \land \mathsf{locked}' \vdash \mathsf{sound}' \land \mathsf{flash}' \land \neg\mathsf{armed}' \land \mathsf{toa}_\mathsf{s}{}' = \mathsf{current}' + 30 \\
c_4 & : & \mathsf{armed} \land \neg\mathsf{locked}' \land \mathsf{closed}' \vdash \neg\mathsf{armed}' \\
c_5 & : & \mathsf{sound} \land \mathsf{flash} \land \mathsf{current}' = \mathsf{toa}_\mathsf{s} \vdash \neg\mathsf{sound}' \land \mathsf{flash}' \land \mathsf{toa}_\mathsf{f}{}' = 270 \\
c_6 & : & \neg\mathsf{sound} \land \mathsf{flash} \land \mathsf{toa}_\mathsf{f}{}' = \mathsf{current}' \vdash \neg\mathsf{sound}' \land \neg\mathsf{flash}' \land \mathsf{silent}' \\
c_7 & : & \mathsf{sound} \land \neg\mathsf{locked}' \vdash \neg\mathsf{sound}' \land \neg\mathsf{flash}' \\
c_8 & : & \mathsf{silent} \land \mathsf{closed} \land \mathsf{locked}' \land \neg\mathsf{closed} \vdash \mathsf{armed}' \\
c_9 & : & \mathsf{silent} \land \mathsf{locked}' \vdash \neg\mathsf{silent}' \\
c_{tick} & : & \mathsf{true} \vdash \mathsf{current}' = \mathsf{current} + 1
\end{array}$$

Note that this type of contracts can only express deterministic time delays, as a contract enforces the guarantee as soon as the assumption is enabled, thus $\mathsf{toa}'_\mathsf{a} <= \mathsf{current}' \vdash \mathsf{armed}'$ would immediately set armed' to true as soon as $\mathsf{toa}'_\mathsf{a} = \mathsf{current}'$ holds. However, if the assumption is also restricted by other variables, e.g. $\mathsf{toa}'_\mathsf{a} <= \mathsf{current}' \wedge \mathsf{closed}' \wedge \mathsf{locked}' \vdash \mathsf{armed}'$, using a time range instead of a fixed value may still make sense to only trigger the contract if the general precondition ($\mathsf{closed}' \wedge \mathsf{locked}'$) and the timing constraint ($\mathsf{toa}'_\mathsf{a} <= \mathsf{current}'$) are satisfied simultaneously.

10.2.2 Symbolic Ticks

In the previous subsection we discussed the modeling of time with explicit ticks to measure the progress of times. Naturally, a time delay of x time units needs x clock cycles invoking $tick$ to be completed. Consequently, this modeling style neglects one of the main advantages of synchronous systems in the context of test-case generation, the short length of the test cases. While we previously saw that synchronous test cases may summarize several steps of an asynchronous test case into one step, we now need several steps (up to 270 in case of the car alarm system), just to measure one time delay.

The natural way to avoid this is the parametrization of time delays. Thus, during each step, the clock is not always increased by one, but by any positive discrete value. The notion of this is, that we apply inputs, wait for the specified delay, and then read the outputs. Thus, each iteration through the requirement interface is associated to two times: the starting time, where the inputs are applied, and the ending time, when the outputs are read.

To illustrate this on the car alarm system presented in the last subsection (Example 10.2), the tick contract needs to be updated to $c_{tick} : \mathsf{true} \vdash \mathsf{current}' >= \mathsf{current}$. In the case of multiple clock variables (Example 10.1), one additionally needs to ensure that the passage of time is equal for all of them.

One problem with underspecified delays is that the expected time of arrival of a variable may be skipped by a longer time delay, thus never enabling the corresponding contract. This can be avoided by restricting the delay of time, so it cannot skip any of the deadlines. The updated tick contract for the car alarm system with restricted delays is:

$$\begin{aligned} c_{tick} \quad : \quad & \mathsf{true} \vdash \mathsf{current}' >= \mathsf{current} \wedge \\ & \mathsf{current} < \mathsf{toa_a} \implies \mathsf{current}' \leq \mathsf{toa_a} \wedge \\ & \mathsf{current} < \mathsf{toa_f} \implies \mathsf{current}' \leq \mathsf{toa_f} \wedge \\ & \mathsf{current} < \mathsf{toa_s} \implies \mathsf{current}' \leq \mathsf{toa_s} \end{aligned}$$

This notion works very well for examples like the car alarm system. However, the encoding gives time different semantics than the explicit ticks did. We now expect the system to pause during delay, while in the previous approach, the contracts that are not time-dependent were synchronously executed. Consider a contract that increases some arbitrary variable x: $c_1 : \mathsf{true} \vdash x' = x + 1$ and a contract $c_2 : toa_y = current \vdash y'$. In the previous approach, a delay of 20 ticks enforced by the second contract would have caused 20 iterations, and thus x would have been increased by 20, during the delay. With the parametrized delays, contract c_1 would only be executed once, while in parallel the parametrized tick contract would state that the iteration took 20 seconds.

This implies, that this symbolic notion of time can not be used for safety-critical embedded systems, where the constant interaction with the environment is of extremely high importance. However, for systems like the car alarm system, which contain strict timing limitations, but no variables that continuously change over time, it works very well.

A test case that activates the alarm and waits until both the flash and the sound stopped takes nine discrete steps (close, lock, armedOn, open, armedOff, flashOn, soundOn, soundOff, flashOff) in the asynchronous timed automata example, plus the time steps in between. With explicit ticks, it would take five steps where inputs are changed plus 320 steps for the ticks. With the parametrized ticks, the five steps are sufficient.

11 Case Studies and Results

Parts of this chapter are based on our publications at FMICS 2015 [11] and QSIC 2014 [12].

In this chapter we will introduce the case studies we used for evaluating our approach on synchronous systems. We will introduce a safing engine for an airbag chip and present synchronous specifications of the adjustable speed limiter and the car alarm system, that were already introduced in Part I.

Then we will present results for the test-case generation with manually designed test purposes, automatically generated test purposes for model-based mutation testing, and finally for the model-based mutation testing of systems that contain time delays. All experiments were run on a MacBook Pro with a 2.8 GHz Intel Core i7 Processor and 8 GB RAM.

11.1 Case Studies

11.1.1 Safing Engine

We now provide more details on the safing engine use case that was already introduced in Section 1.8.2 and Chapter 9. It was an automotive use case supplied by the European ARTEMIS project MBAT, that partially motivated our work on requirement interfaces. The use case was initiated by our industrial partner Infineon and evolves around building a formal model for analysis and test-case generation for the safing engine of an airbag chip.

The chip provides both the power electronics for actually deploying the airbag and the control logic interacting with the control CPU, supported by a so called safing engine. The safing engine double checks the validity of the commands sent from the CPU and can block deployment if the situation does not have the characteristics of a crash. This is obviously safety-critical, as both, deployment of the airbag without a crash and no deployment of the airbag in case of a crash, may lead to hazardous events.

To prevent any malfunction of the safing engine, the basic functionality of the airbag is controlled by an internal state machine, consisting of seven different states. The airbag can only be triggered within a certain state and while a combination of several input signals is applied. The initial state of the airbag system is the RESET state (1). Apart from the initial condition, this state can also be reached whenever the airbag is reset by an external signal. Within the RESET state all functionality of the chip is deactivated. As soon as the external reset signal is turned off, the chip traverses to the INITIAL state (2). While in this state, self-diagnostic functions are active which test the integrity of the basic hardware functions. If an error is detected, a reset is enforced leading to a transition to the RESET state. After receiving a certain sequence of commands from the controller CPU via the Serial Peripheral Interface (SPI) Bus, the controller chip moves to the DIAGNOSTIC state (3). Within the DIAGNOSTIC state, several additional tests are executed. Detecting an error in this state leads the system to the SAFE state (4), whereas a successful diagnosis leads the system to the NORMAL state (5). While the system is in the SAFE state, it may not react to any inputs or trigger any outputs, until the system has been reset by an external signal. In the NORMAL state, the system is fully functional. It is reacting to all inputs and if they match certain conditions, the airbag can be triggered. The TEST state (6) can be reached from the NORMAL state via certain SPI commands. In this state, different output signals can be triggered directly via SPI commands to test the functionality. The last state is the DESTRUCTION state (7), which enables the manual deployment of the airbags at the end of life of the airbag system.

The requirements document, developed by a customer at Infineon, is written in natural (English) language. We identified 39 requirements that represent the core of the system's functionality and iteratively formalized them in collaboration with the designers of Infineon. The resulting formal requirement interface is deterministic and consists of 32 contracts. It contains 11 input variables, 5 output variables and 11 hidden state variables, where one of each variables is an integer variable, and the rest are Boolean.

11.1.2 Adjustable Speed Limiter

The second industrial case study, the adjustable speed limiter, was already introduced in Part I, in Section 5.1.2. We will give a short description as a reminder: it contains an internal state machine with three states: OFF, LIMITING and OVERRIDDEN. Upon activation, it either takes the current speed as limit, or a predefined value. The limit can then be increased and decreased manually, and a kickdown of the gas pedal overrides the speed limiter for some time threshold. Adjusting the speed, or setting it to the predefined value, ends the overridden mode. Finally, the speed limiter can be turned off again, both from overridden and active mode.

The part of the adjustable speed limiter that was analysed within the project was documented by 17 informal requirements. These were refined to 26 formal requirements, i.e. contracts, collected in one requirements interface. The interface contains two input variables, two output variables and four internal variables. The example below shows three characteristic contracts which serve as an illustration of the functionality of the speed limiter, where set and state are output variables, in and kickdown are input variables, preset_value and timer are internal variables, and preset and plus are enum values. Contract c_1 switches the adjustable speed limiter on, assigning the preset value as current limit. Contract c_2 adjusts the current limit, increasing it by one. And c_3 activates the overridden mode, in case of a gas pedal kickdown. It also resets a clock variable for the automated timeout, that would lead back to limiting mode.

$$
\begin{array}{lll}
c_1 & : & \mathsf{state} = \mathsf{OFF} \wedge \mathsf{in}' = \mathsf{preset} \wedge \neg \mathsf{kickdown}' \ \vdash \\
 & & \mathsf{state}' = \mathsf{LIMITING} \wedge \mathsf{set}' = \mathsf{preset\_value} \\
c_2 & : & \mathsf{state} = \mathsf{LIMITING} \wedge \mathsf{in}' = \mathsf{plus} \wedge \neg \mathsf{kickdown}' \ \vdash \\
 & & \mathsf{state}' = \mathsf{LIMITING} \wedge \mathsf{set}' = \mathsf{set} + 1 \\
c_3 & : & \mathsf{state} = \mathsf{LIMITING} \wedge \mathsf{kickdown}' \ \vdash \\
 & & \mathsf{state}' = \mathsf{OVERRIDDEN} \wedge \mathsf{timer}' = 0
\end{array}
$$

11.1.3 Car Alarm System

The car alarm system was already introduced in Section 5.1.1, and its synchronous models were shown in Section 10.2. In this chapter we will present results for the models with one single clock, that is, the model from Example 10.2 with explicit ticks and Example 11.1, that is presented below, with symbolic ticks.

Both models consist of two input signals, three output signals. The explicit model needs six internal integer variables for handling time, and an internal Boolean variable. The symbolic model needs four internal integer variables for handling the time and the same internal Boolean variable.

Example 11.1. The requirement interface for the car alarm system with symbolic ticks is given as A^{cas3} : $\hat{C}^{cas3} = \{c_0\}$, $C^{cas3} = \{c_i \mid i \in [1,9]\} \cup \{c_{tick}\}$, $X_I^{cas3} = \{\mathsf{closed}, \mathsf{locked}\}$, $X_O^{cas3} = \{\mathsf{armed}, \mathsf{flash},$

$\mathsf{sound}\}$ and $X_H^{cas3} = \{\mathsf{silent}, \mathsf{toa_a}, \mathsf{toa_s}, \mathsf{toa_f}, \mathsf{current}\}$ where

$$\begin{array}{lll}
c_0 & : & \mathsf{current} = 0 \wedge \neg\mathsf{sound} \wedge \neg\mathsf{flash} \wedge \neg\mathsf{armed} \wedge \mathsf{clk_a} = 0 \wedge \mathsf{clk_s} = 0 \wedge \mathsf{clk_f} = 0 \wedge \neg\mathsf{silent} \\
c_1 & : & \mathsf{closed}' \wedge \mathsf{locked}' \wedge \neg\mathsf{flash}' \wedge \neg\mathsf{sound}' \wedge (\neg\mathsf{closed} \vee \neg\mathsf{locked})) \ \vdash \ \mathsf{toa_a}' = \mathsf{current}' + 20 \\
c_2 & : & \mathsf{closed}' \wedge \mathsf{locked}' \wedge \mathsf{current}' = \mathsf{toa_a}' \ \vdash \ \mathsf{armed}' \\
c_3 & : & \mathsf{armed}' \wedge \mathsf{closed} \wedge \neg\mathsf{closed}' \wedge \mathsf{locked}' \ \vdash \ \mathsf{sound}' \wedge \mathsf{flash}' \wedge \neg\mathsf{armed}' \wedge \mathsf{toa_s}' = \mathsf{current}' + 30 \\
c_4 & : & \mathsf{armed} \wedge \neg\mathsf{locked}' \wedge \mathsf{closed}' \ \vdash \ \neg\mathsf{armed}' \\
c_5 & : & \mathsf{sound} \wedge \mathsf{flash} \wedge \mathsf{current}' = \mathsf{toa_s} \ \vdash \ \neg\mathsf{sound}' \wedge \mathsf{flash}' \wedge \mathsf{toa_f}' = 270 \\
c_6 & : & \neg\mathsf{sound} \wedge \mathsf{flash} \wedge \mathsf{toa_f}' = \mathsf{current}' \wedge \mathsf{locked}' \ \vdash \ \neg\mathsf{sound}' \wedge \neg\mathsf{flash}' \wedge \mathsf{silent}' \\
c_7 & : & \mathsf{sound} \wedge \neg\mathsf{locked}' \ \vdash \ \neg\mathsf{sound}' \wedge \neg\mathsf{flash}' \\
c_8 & : & \mathsf{silent} \wedge \mathsf{closed} \wedge \mathsf{locked}' \wedge \neg\mathsf{closed} \ \vdash \ \mathsf{armed}' \\
c_9 & : & \mathsf{silent} \wedge \mathsf{locked}' \ \vdash \ \neg\mathsf{silent}' \\
c_{tick} & : & \mathsf{true} \ \vdash \ \mathsf{current}' >= \mathsf{current} \wedge \mathsf{current} < \mathsf{toa_a} \implies \mathsf{current}' \leq \mathsf{toa_a} \\
 & & \wedge \mathsf{current} < \mathsf{toa_f} \implies \mathsf{current}' \leq \mathsf{toa_f} \\
 & & \wedge \mathsf{current} < \mathsf{toa_s} \implies \mathsf{current}' \leq \mathsf{toa_s}
\end{array}$$

11.1.4 Overview

Table 11.1 presents the characteristics of the three case studies, in terms of the number of contracts, the number of variables, and the number of integer and Boolean variables.

11.2 Manual Definition of Test Purposes

In this section we will present our results for the general test-case generation methodology, that does not yet include model-based mutation testing. We will first discuss the buffer that was used as a demonstrating example throughout this part and then discuss the safing engine.

11.2.1 Demonstrating Example

In order to experiment with our algorithms, we model three variants of the abstract buffer behavioral interface. All three variants model buffers of size 150, with different internal structure. *Buffer 1* models a simple buffer with a single counter variable k. *Buffer 2* models a buffer that is composed of two internal counter variables of size 75 each and *Buffer 3* models a buffer that is composed of three internal counter variables of size 50 each. We also remodel a variant of the power consumption interface that creates a dependency between the power used and the state of the internal buffers (idle/used).

All versions of the behavior interfaces can be combined with the power consumption view point from Example 7.4, either using the incremental approach 8.2.1 or doing the conjunction before generating test cases from the monolithic specification.

Incremental Consistency Checking. In order to evaluate the consistency check, we introduce three

Table 11.1: Characteristics of the three case studies.

Case study	# Contracts	# Variables	# Integers	# Booleans
Safing engine	32	27	3	24
Speed limiter	26	6	5	1
CAS (explicit)	13	12	6	6
CAS (symbolic)	13	10	4	6

Table 11.2: Run-time in seconds for checking consistency of single and conjuncted interfaces of the buffer.

	Fault 1 (behavior)		Fault 2 (power)	
	single	monolithic	single	monolithic
Buffer 1	0.7	3.6	1.0	7.3
Buffer 2	5.3	13.4	1.0	26.7
Buffer 3	7.2	13.8	1.0	13

faults to the behavioral and power consumption models of the buffer: *Fault 1* causes deq to decrease k when the buffer is empty; *Fault 2* alters an assumption resulting in conflicting requirements for power consumption upon enq; and *Fault 3* causes enq to increase k when the buffer is full. Due to the fact that we have three implementations of the buffer, the fault injection results in 9 faulty variants of interfaces, each containing a single fault.

We compare monolithic consistency checking to the consistency checking of individual views. We first note that the consistency check is coupled with the algorithm for finding minimal inconsistent sets of contracts (see Section 8.2.3). We set the range of the integer values to $[-2 : 152]$ and we bound the search depth to 3. In the monolithic approach, we first conjunct all view models and then check for consistency. In the incremental approach, we first check the consistency of individual views, and then, if no inconsistency is found, conjunct them one by one, checking consistency of partial conjunctions. However, in the current example, as we knew which view was faulty, we always started with the faulty view and as the inconsistency was detectable by examining only one view, we did not need to conjunct the second view. The results thus only show the comparison of checking consistency in complete and partial models, highlighting the advantage that requirement interfaces gain by allowing to split models into separate views. Table 11.2 summarizes and compares the time it takes to find an inconsistency and compute the minimal inconsistent set of requirements in the requirement interface of a single view and in the monolithic interface which is a conjunction of both views. It gives a very nice impression on how separating the different views helps decreasing the complexity, and thus the runtime, of the consistency checks. For example, for *Fault 2* in the second buffer, it reduces the runtime of the consistency check from 26 seconds to one second. *Fault 3* is omitted in the table, as neither approach was able to find an inconsistency. This is caused by the fact that the fault lies to deep in the system, and cannot be detected with the given search depth.

The bounded consistency checking is very sensitive to the search depth. Setting the bound to 5 increases the run-time from seconds to minutes – this is not surprising, since a search of depth n involves simplifying formulas with alternating quantifiers of depth n, which is a very hard problem for SMT solvers.

Test-Case Generation. We compare the monolithic (see Section 8.2) and incremental (see Section 8.2.1) approach to test-case generation, by generating monolithic tests for the conjunction of the buffer interfaces and the power consumption interface, and incrementally, by generating tests only for the buffer interfaces, and completing them with the power consumption interface. The tests were generated according to manually defined test purposes, that required the buffer to be full. Thus, the according test cases needed to perform 150 enqueue operations, and were of length 150. Table 11.3 summarizes the results, presenting the number of contracts and variables of the requirement interfaces, the runtime of the incremental test-case generation and the runtime of the monolithic approach. For the incremental approach, the runtime includes the test-case generation using only the behavioral view and the completion of the test case, according to the power consumption. The three examples diverge in complexity, expressed in the number of contracts and variables. Our results show that the incremental approach outperforms the monolithic one, resulting in speed-ups from 33% to 68%.

Table 11.3: Run-time in seconds for incremental and monolithic test-case generation on the buffer.

	# Contracts	# Variables	t_{inc}	t_{mon}	speed-up
Buffer 1	6	6	10	16.8	68 %
Buffer 2	15	12	36.7	48.8	33 %
Buffer 3	20	15	69	115.6	68 %

11.2.2 Safing Engine

During the formalization process we already revealed several under-specifications in the informal requirements that were causing some ambiguities. These ambiguities were resolved in collaboration with the designers. The following consistency check performed with MoMuT::REQs revealed two inconsistencies between the requirements. Tracing the conflicts back to the informal requirements enabled their fixing in the customer requirements document.

We generated 21 test cases from the formalized requirements via test purposes that were designed to ensure that every Boolean internal and output variable is activated at least once and that every possible state of the underlying finite state machine is reached at least once. Thus, the test suite provides state and signal coverage. The average length of the test cases was 3.4 and the maximal length was 6, but since the test cases are synchronous, each of the steps is able to trigger several inputs and outputs at once. The test cases were used to test the Simulink model of the system, developed by Infineon as part of their design process. The Simulink model of the safing engine consists of a state machine with seven states, ten smaller blocks transforming the input signals and a Matlab function calculating the final outputs according to the current state and the input signals. In order to execute the test cases, Infineon's engineers developed a test adapter that transforms abstract input values from the test cases to actual inputs passed to the Simulink model.

We created 66 faulty Simulink models (six turned out to be equivalent), by flipping every Boolean signal (also internal ones) involved in the Matlab function calculating the final output signals. Our 21 test cases were able to detect 31 of the 60 non-equivalent faulty models, giving a mutation score of 51.6%. These numbers show that state and signal coverage is not enough to find all faults and confirm the need to incorporate a more sophisticated test-case generation methodology. Therefore, we manually added 10 test purposes generating 10 additional test cases. The combined 31 test cases finally reached a 100% mutation score. This means that all injected faults in the Simulink models were detected.

However, these additional test cases had to be designed by analyzing the existing test cases and the correct Simulink model and figuring out which combinations of states and transitions were not covered. The task took several hours, and various iterations. Thus, for the following experiments described in the next section, we only performed model-based mutation testing, without designing manual test cases before.

11.3 Model-Based Mutation Testing

We will now present the results of our model-based mutation testing approach for requirement interfaces. We will first demonstrate it on the abstract buffer specification, and then on the two industrial case studies, first on the safing engine, and then on the adjustable speed limiter.

Table 11.4: Results for model-based mutation testing on the buffer, with depth 150.

	# Mutants (Equiv.)	# Unique Tests	[min] t_{mbmt}
Buffer 1	44 (0)	29	4.7
Buffer 2	138 (20)	52	44.0
Buffer 3	240 (46)	115	128.1

11.3.1 Demonstrating Example: Buffer

We applied the model-based mutation testing technique on all three variants of the buffer from Section 11.2.1. For these experiments we did not consider the power consumption, which could be added incrementally after the generation of the tests. We used all mutation operators defined in Section 8.3. Table 11.4 shows the results of the approach, giving the number of mutants, the number of k-equivalent mutants, the number of unique test cases that were produced, and the total time for applying the complete approach to all mutants. The number of unique tests comes from the fact that the model-based mutation testing approach produces negative traces, which are then translated into positive test cases, where several traces may coincide into equivalent test cases. The bound k for the equivalence check was set to 150. The reported times include mutation, generation of according test purposes, test-case generation, conversion into positive test cases and detecting the unique test cases. Buffer 2 and Buffer 3 are more complex and create more mutants, and thus have a longer runtime. Yet, they also generate more unique tests, und thus a more thorough test suite.

11.3.2 Safing Engine

We applied two iterations of the model-based mutation testing approach to the safing engine introduced in Section 11.2.2, setting the bound k to 7. In the first iteration, we generated 362 mutants, applying all mutation operators. We generated 165 negative tests - 197 mutants were k-equivalent. From the 165 negative tests, we extracted 28 unique positive test cases.

The mutation score achieved by these 28 test cases on the 60 faulty Simulink models was surprisingly low, with only 49.2%. A closer investigation of the requirement interface shows that many of the contracts work globally, without being bound to a specific state of the state machine. For mutants from these contracts, our approach only generates one test case, even though the mutants generate multiple faults, in several different states. Due to the decomposed structure, even though we only insert one fault, our mutants are not classic first-order mutants anymore.

There are two ways to deal with this problem. The first one would be the generation of multiple test cases per mutant, that cover all possible faulty states. This technique was already applied previously, in a different context [6]. However, this approach might become very expensive, and impossible for systems with infinite state space.

The second approach is based on refactoring of the contracts, splitting global contracts into multiple more fine-grained ones. E.g., contract c_3 could be refactored into several state-bound contracts like

$$\begin{aligned} &\mathsf{c}_{3,1} : \mathsf{reset'} \wedge \mathsf{state'} = \mathsf{INIT} \vdash \mathsf{state'} = \mathsf{RESET} \\ &\mathsf{c}_{3,2} : \mathsf{reset'} \wedge \mathsf{state'} = \mathsf{DESTR} \vdash \mathsf{state'} = \mathsf{RESET} \end{aligned}$$

Applying this technique we gained 17 new contracts. The second run of our test-case generation produced 525 mutants and 293 were detected as non-equivalent. This led to 61 unique test cases. The mutation and the generation of the test cases took 139.8 seconds, and the distribution of their length is shown in Table 11.5. 75% of the time was spent on equivalent mutants, which is once more a very high value.

Table 11.5: Distribution of the length of the test cases generated for the safing engine.

	Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6	Depth 7
Number of tests	5	16	12	3	8	16	1

The test cases were able to detect 53 of the faulty mutants, resulting in a mutation score of 88%. Jia and Harman [103] state that up to 40% of all mutants are equivalent, which means that a mutation score over 60% can be considered very well. However, since we already discarded all equivalent mutations in this case, the score of 88% is still rather low, and further refining of the contracts woulds still be advantageous.

This shows, that the quality of model-based mutation testing for requirement interfaces is severely depending on the modeling style. However, while the fine grained contracts might slightly decrease the clarity of the requirement interfaces, they in turn increase the traceability, and facilitate fault detection.

11.3.3 Adjustable Speed Limiter

Next we will discuss our experiments on the adjustable speed limiter case study. Applying the mutation-based test generation to this case study setting the bound k to 4 generates 291 mutants. 57 of the mutants are equivalent, leaving a total of 234 non-equivalent mutants. 96 of these mutants can be detected within one step, 64 mutants are detected after two steps and 72 after the third step. This reflects very clearly the state-based structure that consists of three states. An analysis of the test cases shows that 60 of the tests are unique. Given that the model is deterministic, each of the unique tests enables different contracts in the individual steps. A further analysis of these 60 unique test cases shows that 12 are of length one, 18 of length two and 30 are of length three, as presented in Table 11.6 The test-case generation, including the mutation, took 18.3 seconds, 52% of the time was spent on equivalent mutants.

To evaluate the quality of the test cases, we implemented a Java version of the adjustable speed limiter, and used the *Major mutation framework* [107] to generate a set of 64 faulty implementations, using all mutation operators supported by Major. By executing our generated tests on these faulty implementations, we could perform a classic mutation analysis: our test suite was able to detect 48 of the faults. Further investigation of the undetected faults revealed that 13 of the remaining Java mutants were equivalent, and could thus not be detected by any test case. Another two of the faults were introduced in the conditions of *if*-statements. The conditions correspond to the assumptions of our interfaces, which we did not mutate during the test-case generation.

The last remaining fault was introduced in the timing behavior of the Java implementation, which was simulated via a $tick$ method, indicating the passage of one second. In the requirement interface, it was modeled via a non-deterministically increasing variable. The fault caused the implementation to trigger the state change already after 9 seconds instead of 10. The test driver was not sensitive enough to detect that, as the test case only specified the behavior after 10 seconds, and did not specify what the correct behavior after 9 seconds would be. If, however, the faulty implementation had been delayed instead of producing the output early, we would have detected that, as the output would not have arrived after the expected 10 seconds.

Table 11.6: Distribution of the length of the test cases generated for the adjustable speed limiter.

	Depth 1	Depth 2	Depth 3
Number of tests	12	18	30

11.4 Explicit Ticks vs. Symbolic Ticks: Car Alarm System

We used the car alarm system to evaluate the explicit and symbolic tick modeling styles for real-time systems, both with the single clock notation using times of arrival. The two presented models could both be processed by MoMuT::REQs, without needing any changes to the implementation.

We evaluated the two models by first generating tests cases via model-based mutation testing, and then execution of the test cases on the faulty Java implementations that were already introduced in Chapter 5. Note that these Java implementations work asynchronously, and that thus our test cases had to be executed asynchronously as well. Hence, if lock and close were performed synchronously in the test case, we performed them in random order without time delay between them.

Explicit ticks. The model with explicit ticks, after completing it to disable unintended variable changes, consisted of 28 contracts. From these contracts we generated 126 mutants. We then applied the test-case generation with a maximal search depth of 40. This search depth was sufficient, as the delays in the faulty implementations were divided by ten, and thus reduced to 2,3 and 27 and the requirement interface was adapted accordingly. 24 of the mutants turned out to be equivalent with respect to the search depth. The remaining 102 mutants were non-equivalent and thus produced test cases, 17 of which were unique. The test-case generation took 80.4 seconds. Table 11.7 shows the distribution of the length of the generated test cases. Note that a test case of length 1 can already include both closing and locking the doors. The equivalent mutants took 87% of the complete runtime.

We executed the 17 unique test cases on the 38 faulty implementations and were able to detect the faults in 23 of them. This gives a mutation-score of 60.5%. While this value is comparatively low, this is not very unexpected, given that we applied weak mutation testing, while in the previous experiments using timed automata, we applied strong mutation testing. Several of the faults only propagate to a visible fault after several steps. Additionally, when several actions were modeled synchronously, we only chose one of the interleavings for the test-case execution, thus not testing the other branch.

A further inspection of tests and faulty implementations revealed, that again some of the contracts were too coarse. Consider e.g. contract $R7$ that specifies that unlocking the door deactivates the alarms:

$$c_7 : \mathsf{flash} \wedge \neg \mathsf{locked}' \ \vdash \ \neg \mathsf{sound}' \wedge \neg \mathsf{flash}'$$

In its current form, it is enabled as long as the flash signal is activated. This covers both the cases when additionally the sound is active, and when the sound is already deactivated. As our test cases are designed to be minimal, and the first case can be reached using one step less, this only creates a test case for the first case, and leaves the second one (which actually occurs in one of the faulty implementations) untested.

Symbolic ticks. The model with symbolic time delays consisted of 20 contracts, after being completed. From these, we derived 198 mutants. The test-case generation was executed with a search depth of 10, as the car alarm system can be fully explored with that depth due to the symbolic time delays. With respect to that depth, 43 mutants were found to be equivalent. We generated 155 test cases in 18.3 seconds, 18 of which were unique. The distribution of the lengths of the test cases can be found in Table 11.8.

Table 11.7: Distribution of the length of the test cases generated for the car alarm system with explicit ticks.

	Depth 1	Depth 3	Depth 4	Depth 5	Depth 7	Depth 34	Depth 35
Number of tests	54	1	21	5	11	8	2

Table 11.8: Distribution of the length of the test cases generated for the car alarm system with symbolic ticks.

	Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6
Number of tests	55	42	27	21	3	7

The equivalent mutants needed 66% of the runtime, which is far fewer than for the explicit ticks. This reduction is probably based on the shorter search depth, where the exploration of an equivalent mutant is not so expensive.

By executing the tests on the faulty implementations, we were able to detect 26 of the faults, leaving 12 faults undetected. The resulting mutation score of 68.4% is about 8 percentage points higher than with the other modeling style. For both styles, we manually checked whether the timing faults we manually added to the faulty Java implementations were killed, and can report that they all were killed by the test suite.

11.5 Overview

In this section, we will present a compact overview of the results gained on the industrial case studies described in this chapter. The results are presented in Table 11.9. The different search depths were adjusted to the individual examples, and thus vary strongly. Accordingly, the times for the test-case generation cannot be compared directly. However, the difference in runtime between the explicit and the symbolic ticks of the car alarm system are noteworthy, especially given that the symbolic modeling style lead to better mutation score. Another interesting fact is given by the number of equivalent mutants for the safing engine. This might be caused by the fact that some of the contracts restrict the same variables. Thus, if one of them is mutated, the mutation causes an inconsistency and cannot be detected.

Table 11.9: Model-based mutation testing results of the three case studies.

Case study	# Mutants	# Equiv. Mutants	Search depth	Unique Tests	% Mutation Score	t_{mbmt}
Safing engine	525	232	7	61	88	139.8
Speed limiter	291	57	4	60	94	18.3
CAS (explicit)	126	24	40	17	60.5	80.4
CAS (symbolic)	198	43	10	18	68.4	18.3

Part III

Discussion

Overview

In the third part of this thesis, we will discuss the presented work, focusing on several aspects. First, in Chapter 12, we will compare our approaches for asynchronous and synchronous models and summarize the most important differences we found. Then we will give a short overview how the two approaches may be combined to form asynchronous specifications with synchronous components. Next, in Chapter 13, we will discuss related work in the contexts of model-based testing of real-time systems, model-based mutation testing, timed automata and synchronous specifications. Finally, in Chapter 14, we will provide a summary of the thesis, focus on the achieved contributions and conclude our work with an overview of possible future work.

12 Asynchrony and Synchrony

To bring the two parts of this thesis together, this chapter will contain both a comparison and a discussion of the possible combination of asynchronous and synchronous systems. The comparison of the two approaches will be given in Section 12.1, discussing the strengths and weaknesses of the two approaches. Then, in Section 12.2 we show how the two approaches can be combined and how such a combination can benefit from the strengths of both approaches.

12.1 Comparison of Asynchrony and Synchrony

A direct experimental comparison between the asynchronous and the synchronous modeling approaches is hardly possible. The models of the two approaches differ in terms of general attributes, like model-size, possible model-elements or the numer of variables, but also in several means in the context of testing: the length of the produced test cases, the complexity of the test-case execution procedure or the level of abstraction. However, we want to point out some interesting details we learned during our experiments:

- **Length of the Test Cases.** The first and most substantial observation was the length of the test cases. Being able to process several inputs simultaneously, drastically decreases the needed search depths for most systems, and thus substantially decreases the complexity of the conformance checks. While this seems like a definite advantage of the synchronous approach, is creates problems as well: the test-case generation procedure, trying to create as short test cases as possible, will always perform as many inputs simultaneously as possible. If, however, an interleaved version would also be possible in the SUT, the different interleavings will never be tested.
- **Possible Levels of Abstractions.** During our experiments on the adjustable speed limiter (Section 11.3.3), we tried several different modeling approaches on three levels of abstraction, that were performed and analyzed by Grischa Liebel from Chalmers University. The first approach was a network of timed automata, that was performed on a very detailled level, including C-like functions in guards and consisting of 13 automata. As this model could not be processed by MoMuT::TA due to the unsupported C-like functions, and since it was not very comprehensible either, this model was soon dropped, and replaced by a more compact model.

  This second model was on a medium level of abstraction. But while it got rid of the C-like functions, it still contained a network of timed automata and guards and updates with data variables, and was thus not processable by MoMuT::TA, that only supports either of these two modeling elements. However, we did a manual transformation to MoMuT::REQs, and could create first test cases on that level of abstraction. MoMuT::REQs seemed very well suited for that abstraction level, as it could cope with the data variables very well, and most of the automata in the network worked independently, and their translation could simply be conjuncted to the requirement interface. However, the model was soon refactored once more, and a third high-level model was designed and used for the rest of the project.

  This third model is the one presented in Section 11.3.3, and was on the right level of abstraction for both MoMuT::REQs and MoMuT::TA. While both tools performed well for that specific model, MoMuT::REQs was already able to process the more low-level specification as well, which indicates that it might be better applicable for low-level abstraction models than MoMuT::TA.
- **Mutation Score.** In terms of quality of the created test suites, the asynchronous models perform better. Our experiments on the car alarm system achieved a 100% mutation score on the asynchronous car alarm system, and only $60/68\%$ on the synchronous models. However, the mutation

operators are slightly different, the synchronous approach currently only performs weak mutation testing and the car alarm system implementation is an asynchronous system, thus the tests had to be transformed to asynchronous execution traces, missing some interleavings. Choosing a synchronous implementation instead of the asynchronous Java implementation of the car alarm system might change that comparison, but implementing strong mutation testing for MoMuT::REQs is still an important future task for upgrading the quality of the generated test suites.

- **Suitability for Real-Time Systems.** For embedded systems that conform to the synchrony hypothesis, the synchronous approach that implicitly determines that all outputs are produced instantaneously is very well suited. It facilitates the specification of such systems in a way that is both comprehensibly and easily to process. However, for delayed outputs that rely on hard timing constraints, timed automata still have a clear advantage. The implicit progress of time in the locations clearly outgoes the explicit modeling of time progress with ticks, both with respect to comprehensibility of the model, and with respect to reduced complexity of the test-case generation.
- **Non-determinism.** Non-determinism is supported by both tools. However, MoMuT::TA needs to determinize the models before the test-case generation, which adds additional complexity and increases the state-space. Thus, MoMuT::Reqs, even though right now the monitoring using the non-deterministic specification needs to be done manually, is probably better suited for non-deterministic models. However, only MoMuT::TA supports non-deterministic timing behavior, as the contracts of MoMuT::Reqs always have to trigger as soon as the assumption is satisfied. Thus, a contract of the form $clk_x \geq 1 \ \vdash \ y$ will always fire as soon as clk_x is equal to one, and even though it would also hold later.

12.2 Combination of Asynchrony and Synchrony

Combinations of synchronous and asynchronous systems are well known in the literature, where the most important combination is called *GALS* systems, which stands for *globally asynchronous, locally synchronous* systems [158]. Such systems consist of several synchronous components, that are connected via a global asynchronous interconnection. The different synchronous components may run at different frequencies, and the asynchronous communication between them is handled by e.g. FIFO buffers. Teehan et al.[158] provided a survey on GALS systems, giving a classification of different GALS approaches, and also stating the advantages of the GALS design, like fast block-reuse and power savings.

In our context, we can have several requirement interfaces that model individual components, and are connected via a global time automata model. The advantage of this modeling style is that the individual components as well as the timed automata system can be kept very small, enabling efficient test-case generation for them. After generating test cases from the timed automata model, the individual signals in the trace can be replaced by more concrete sub-traces gained from the synchronous components.

Example 12.1. Consider e.g. the car alarm system, that was already mentioned several times during this thesis. In all previous timed automata models, we had the four inputs lock?, unlock?, close? and open?. However, these signals reflect the actual inputs to a car rather poorly, given that a car usually consists of 6 doors, counting the luggage compartment and the bonnet. And actually, the original requirements state that very well, talking about ”all doors, the bonnet and the luggage compartment”. However, modeling inputs for every of these doors leads to a huge state-space explosion in the model, due to all the possible interleavings. Thus, previously the inputs were abstracted and a single close? input was supposed to simulate the closing of the last open door, while an open? was considered opening the first door. Since our Java implementation of the car alarm system was on the same level of abstraction, this did not pose a problem. However, for hardware testing on a real car, the test cases would need to be translated to a less abstract form, closing and opening as many doors as needed in each step.

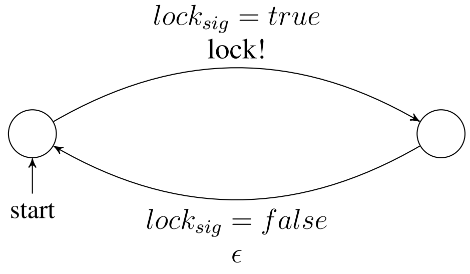
Now consider the requirement interface depicted below. It only contains one single component of the car alarm system, that takes care of the individual doors. It contains eight different inputs: $\mathsf{closed}_{\mathsf{d1}} \ldots \mathsf{closed}_{\mathsf{d4}}$ denote whether the individual doors are closed or opened, and $\mathsf{locked}_{\mathsf{d1}} \ldots \mathsf{locked}_{\mathsf{d4}}$ denote whether the doors are currently locked or unlocked[7]. The four outputs, $\mathsf{lock}_{\mathsf{sig}}$, $\mathsf{close}_{\mathsf{sig}}$, $\mathsf{unlock}_{\mathsf{sig}}$ and $\mathsf{open}_{\mathsf{sig}}$ are triggered if all doors are locked or closed, and if one door is unlocked or opened. Contracts c_1 to c_4 are responsible for triggering the outputs in the right situations, while contracts c_5 to c_8 deactivate the outputs, whenever c_1 to c_4 are not active.

We formally define A^{cas_doors} as $\hat{C}^{cas\_doors} = \{c_0\}$, $C^{cas\_doors} = \{c_i \mid i \in [1,8]\}$, $X_I^{cas\_doors} = \{\mathsf{closed}_{\mathsf{d1}}, \mathsf{closed}_{\mathsf{d2}}, \mathsf{closed}_{\mathsf{d3}}, \mathsf{closed}_{\mathsf{d4}}, \mathsf{locked}_{\mathsf{d1}}, \mathsf{locked}_{\mathsf{d2}}, \mathsf{locked}_{\mathsf{d3}}, \mathsf{locked}_{\mathsf{d4}}\}$, $X_O^{cas\_doors} = \{\mathsf{lock}_{\mathsf{sig}}, \mathsf{close}_{\mathsf{sig}}, \mathsf{unlock}_{\mathsf{sig}}, \mathsf{open}_{\mathsf{sig}}\}$ and $X_H^{cas\_doors} = \{\}$ where

$$\begin{array}{lll}
c_0 & : & \neg\mathsf{lock}_{\mathsf{sig}} \wedge \neg\mathsf{unlock}_{\mathsf{sig}} \wedge \neg\mathsf{close}_{\mathsf{sig}} \wedge \neg\mathsf{open}_{\mathsf{sig}} \\
c_1 & : & \mathsf{closed}_{\mathsf{d1}}{}' \wedge \mathsf{closed}_{\mathsf{d2}}{}' \wedge \mathsf{closed}_{\mathsf{d3}}{}' \wedge \mathsf{closed}_{\mathsf{d4}}{}' \\
 & & \wedge\neg(\mathsf{closed}_{\mathsf{d1}} \wedge \mathsf{closed}_{\mathsf{d2}} \wedge \mathsf{closed}_{\mathsf{d3}} \wedge \mathsf{closed}_{\mathsf{d4}}) \ \vdash \ \mathsf{close}'_{\mathsf{sig}} \\
c_2 & : & \neg\mathsf{closed}_{\mathsf{d1}}{}' \vee \neg\mathsf{closed}_{\mathsf{d2}}{}' \vee \neg\mathsf{closed}_{\mathsf{d3}}{}' \vee \neg\mathsf{closed}_{\mathsf{d4}}{}' \\
 & & \wedge(\mathsf{closed}_{\mathsf{d1}} \wedge \mathsf{closed}_{\mathsf{d2}} \wedge \mathsf{closed}_{\mathsf{d3}} \wedge \mathsf{closed}_{\mathsf{d4}}) \ \vdash \ \mathsf{open}_{\mathsf{sig}}{}' \\
c_3 & : & \mathsf{locked}_{\mathsf{d1}}{}' \wedge \mathsf{locked}_{\mathsf{d2}}{}' \wedge \mathsf{locked}_{\mathsf{d3}}{}' \wedge \mathsf{locked}_{\mathsf{d4}}{}' \\
 & & \wedge\neg(\mathsf{locked}_{\mathsf{d1}} \wedge \mathsf{locked}_{\mathsf{d2}} \wedge \mathsf{locked}_{\mathsf{d3}} \wedge \mathsf{locked}_{\mathsf{d4}}) \ \vdash \ \mathsf{lock}_{\mathsf{sig}}{}' \\
c_4 & : & \neg\mathsf{locked}_{\mathsf{d1}}{}' \vee \neg\mathsf{locked}_{\mathsf{d2}}{}' \vee \neg\mathsf{locked}_{\mathsf{d3}}{}' \vee \neg\mathsf{locked}_{\mathsf{d4}}{}' \\
 & & \wedge(\mathsf{locked}_{\mathsf{d1}} \wedge \mathsf{locked}_{\mathsf{d2}} \wedge \mathsf{locked}_{\mathsf{d3}} \wedge \mathsf{locked}_{\mathsf{d4}}) \ \vdash \ \mathsf{unlock}'_{\mathsf{sig}} \\
c_5 & : & \neg(\mathsf{closed}_{\mathsf{d1}}{}' \wedge \mathsf{closed}_{\mathsf{d2}}{}' \wedge \mathsf{closed}_{\mathsf{d3}}{}' \wedge \mathsf{closed}_{\mathsf{d4}}{}' \\
 & & \wedge\neg(\mathsf{closed}_{\mathsf{d1}} \wedge \mathsf{closed}_{\mathsf{d2}} \wedge \mathsf{closed}_{\mathsf{d3}} \wedge \mathsf{closed}_{\mathsf{d4}})) \ \vdash \ \neg\mathsf{close}_{\mathsf{sig}}{}' \\
c_6 & : & \neg(\neg\mathsf{closed}_{\mathsf{d1}}{}' \vee \neg\mathsf{closed}_{\mathsf{d2}}{}' \vee \neg\mathsf{closed}_{\mathsf{d3}}{}' \vee \neg\mathsf{closed}_{\mathsf{d4}}{}' \\
 & & \wedge(\mathsf{closed}_{\mathsf{d1}} \wedge \mathsf{closed}_{\mathsf{d2}} \wedge \mathsf{closed}_{\mathsf{d3}} \wedge \mathsf{closed}_{\mathsf{d4}})) \ \vdash \ \neg\mathsf{open}_{\mathsf{sig}}{}' \\
c_7 & : & \neg(\mathsf{locked}_{\mathsf{d1}}{}' \wedge \mathsf{locked}_{\mathsf{d2}}{}' \wedge \mathsf{locked}_{\mathsf{d3}}{}' \wedge \mathsf{locked}_{\mathsf{d4}}{}' \\
 & & \wedge\neg(\mathsf{locked}_{\mathsf{d1}} \wedge \mathsf{locked}_{\mathsf{d2}} \wedge \mathsf{locked}_{\mathsf{d3}} \wedge \mathsf{locked}_{\mathsf{d4}})) \ \vdash \ \neg\mathsf{lock}_{\mathsf{sig}}{}' \\
c_8 & : & \neg(\neg\mathsf{locked}_{\mathsf{d1}}{}' \vee \neg\mathsf{locked}_{\mathsf{d2}}{}' \vee \neg\mathsf{locked}_{\mathsf{d3}}{}' \vee \neg\mathsf{locked}_{\mathsf{d4}}{}' \\
 & & \wedge(\mathsf{locked}_{\mathsf{d1}} \wedge \mathsf{locked}_{\mathsf{d2}} \wedge \mathsf{locked}_{\mathsf{d3}} \wedge \mathsf{locked}_{\mathsf{d4}})) \ \vdash \ \neg\mathsf{unlock}_{\mathsf{sig}}{}'
\end{array}$$

The output signals can then be processed by an intermediate timed automaton, that passes them forward to the car alarm system as input events. An example of such an automaton for the $\mathsf{lock}_{\mathsf{sig}}$ output signal of the requirement interface is given in Figure 12.1. It transforms the signal $\mathsf{lock}_{\mathsf{sig}}$ into the even $\mathsf{lock!}$ as soon as it is activated. It goes back into the initial state, when the $\mathsf{lock}_{\mathsf{sig}}$ signal is deactivated.

[7] Note that we still neglect the luggage compartment, as this is only an illustrative example and should be kept simple. Additionally we consider the car not to contain a central locking system.



□

Figure 12.1: An intermediate timed automaton, reading the $\mathsf{lock}_{\mathsf{sig}}$ signal of the synchronous system and passing it on to the car alarm system automaton via the event $\mathsf{lock!}$, once it is activated.

Given a test suite that was generated from the timed automata car alarm system, one can now replace each of the input signals by refined traces consisting of the input signals of the individual doors. However, adding all possible interleavings of these signals to the test suite would lead to an exponential blowup in the size of the test suite. Thus, we propose to extract from each test case the sequence consisting of only the signals that are used in the requirement interface (e.g. in the current case, this are only the inputs. Thus from the test case $(\mathsf{close}, \mathsf{lock}, \mathsf{armedOn}, \mathsf{open})$ we extract $(\mathsf{close}, \mathsf{lock}, \mathsf{open})$) and perform a reachability analysis through the requirement interfaces, to gain a random trace producing exactly these signals in that order. The functionality needed to gain such a refined trace in one single test-case execution is currently not implemented in MoMuT::REQs, as the current test purposes only specify the values in one step, while these new test purposes would specify multiple signals within the trace. Thus, currently one would need to create a test case leading to the first signal ($\mathsf{close}_{\mathsf{sig}}$), and then use the variable valuation at the end of the created test case as the initial variable valuation when creating a test case leading to the second signal ($\mathsf{lock}_{\mathsf{sig}}$), and so on. Then, each of the signals in the timed automata can be replaced by the partial refined trace that leads from the last signal to the current one. This creates a refined test case, that contains all the signals needed for an execution on a real system, and still leads to the fault it is intended to catch.

Example 12.2. Another component in the car alarm system that could be expressed as a synchronous system would be the alarm switching unit itself. A possible requirement interface for the alarm is illustrated below. In this case, we do not refine signals in the timed automata model, but actually replace the corresponding parts of it by the synchronous component. The inputs to the requirement interface are $\mathsf{armed\_on}_{\mathsf{sig}}$, $\mathsf{armed\_off}_{\mathsf{sig}}$, $\mathsf{sound\_off}_{\mathsf{sig}}$, $\mathsf{flash\_off}_{\mathsf{sig}}$ and $\mathsf{open}_{\mathsf{sig}}$. It stores the internal variable armed, and emits the outputs $\mathsf{flash}_{\mathsf{sig}}$ and $\mathsf{sound}_{\mathsf{sig}}$, where the sound is an alternation of high and low tuned pitches. These outputs are not passed to the timed automaton, but directly to the environment.

We formally define A^{cas_alarm} as $\hat{C}^{cas\_alarm} = \{c_0\}$, $C^{cas\_alarm} = \{c_i \mid i \in [1,5]\}$, $X_I^{cas\_alarm} = \{\mathsf{armed\_on}_{\mathsf{sig}}, \mathsf{armed\_off}_{\mathsf{sig}}, \mathsf{sound\_off}_{\mathsf{sig}}, \mathsf{flash\_off}_{\mathsf{sig}}, \mathsf{open}_{\mathsf{sig}}\}$, $X_O^{cas\_alarm} = \{\mathsf{high\_tune}_{\mathsf{sig}}, \mathsf{sound}_{\mathsf{sig}}, \mathsf{flash}_{\mathsf{sig}}\}$ and $X_H^{cas\_alarm} = \{\mathsf{armed}\}$ where

$$\begin{aligned}
c_0 &: \neg\mathsf{armed} \wedge \neg\mathsf{flash}_{\mathsf{sig}} \wedge \neg\mathsf{sound}_{\mathsf{sig}} \\
c_1 &: \mathsf{armed\_on}_{\mathsf{sig}}{}' \vdash \mathsf{armed}' \\
c_2 &: \mathsf{armed\_off}_{\mathsf{sig}}{}' \vdash \neg\mathsf{armed}' \\
c_3 &: \mathsf{armed}' \wedge \mathsf{open}_{\mathsf{sig}}{}' \vdash \neg\mathsf{armed}' \wedge \mathsf{sound}_{\mathsf{sig}}{}' \wedge \mathsf{flash}_{\mathsf{sig}}{}' \\
c_3 &: \mathsf{sound}_{\mathsf{sig}}{}' \vdash \mathsf{high\_tune}_{\mathsf{sig}} \Leftrightarrow \neg\mathsf{high\_tune}_{\mathsf{sig}}{}' \\
c_4 &: \mathsf{sound\_off}_{\mathsf{sig}}{}' \vdash \neg\mathsf{sound}_{\mathsf{sig}}{}' \\
c_5 &: \mathsf{flash\_off}_{\mathsf{sig}}{}' \vdash \neg\mathsf{flash}_{\mathsf{sig}}{}'
\end{aligned}$$

Figure 12.2 shows the updated timed automaton of the car alarm system, where the turning on of the alarm is removed and replaced by the requirement interface. The communication between the requirement interface and the timed automaton is based on the signals $\mathsf{open}_{\mathsf{sig}}$, $\mathsf{armend\_on}_{\mathsf{sig}}$, $\mathsf{armend\_off}_{\mathsf{sig}}$, $\mathsf{sound\_off}_{\mathsf{sig}}$ and $\mathsf{flashoff\_off}_{\mathsf{sig}}$, which are all sent from the timed automaton to the requirement interface. It is a one-side communication, as the outputs of requirement interface, $\mathsf{sound}_{\mathsf{sig}}$ and $\mathsf{flash}_{\mathsf{sig}}$, $\mathsf{high\_tune}_{\mathsf{sig}}$ and $\mathsf{low\_tune}_{\mathsf{sig}}$, are all sent directly to the environment. Communication from the timed automaton to the requirement interface could again be done by intermediate timed automata, similar to the one presented in Figure 12.1, that set the corresponding signal upon receiving an event from the car alarm system automaton.

The part of the requirement interface is comparatively small, but the chosen borders bring several advantages: The handling of the time constraints is still left in the timed automaton, that will trigger the $\mathsf{soundOff}$ and $\mathsf{flashOff}$ event, once the timers run out. The alarms can be modeled in more detail, like e.g. with switching between high and low tunes, as seen in Contract c_3. And the orders of the flash

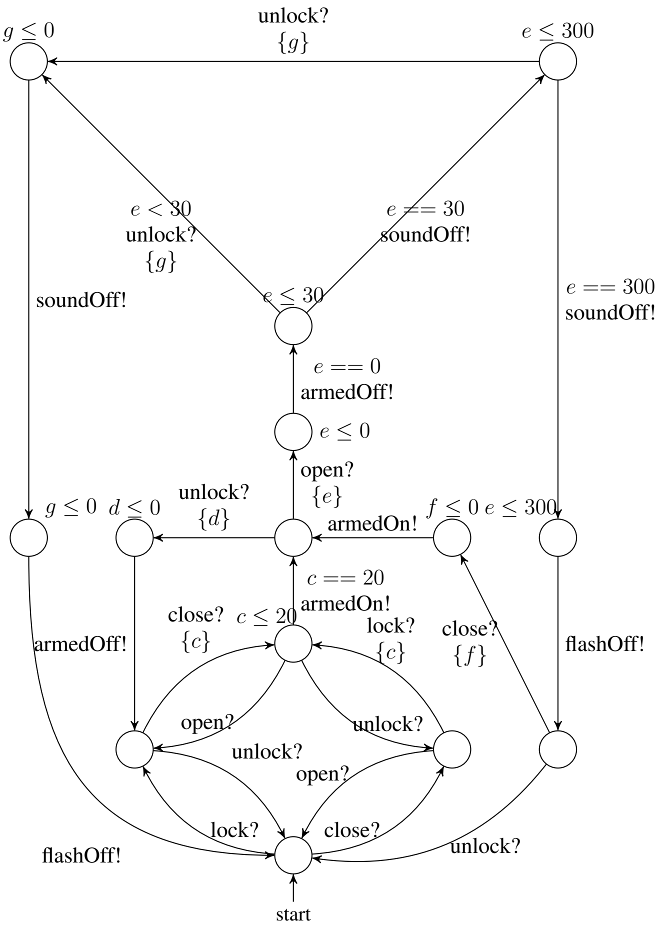


Figure 12.2: A car alarm system, where the alarm switching unit is shifted to a synchronous component.

and sound alarms, which was previously either modeled non-deterministically or fixed to one of the two possible interleavings, does not need to be modeled anymore, as they are now turned on synchronously, as intended.

Even though we did not apply the GALS style in our experiments, it shows a lot of potential. Given that the individual models stay very small and compact, this facilitates a very efficient test-case generation from the individual models, and the test cases can then either be combined, or the individual components can be tested separately.

13 Related Work

13.1 Model-Based Mutation Testing

Model-based mutation testing was initially used for predicate-calculus specifications in a publication by Budd and Gopal [62]. This approach was a combination of classic mutation testing and model-based mutation testing. They created model-mutants, and used them to assess the quality of an existing test suite. Then they produced new test cases, in order to kill the remaining mutants, but this was not done automatically as in the approach presented in this thesis.

Later, model-based mutation testing was applied to formal Z specifications by Stocks [155]. While he showed how to generate test cases able to kill a mutant, this workflow was not automated. Amman et al. [34] used temporal formula to check equivalence between models and mutants, and converted counter-examples to test cases, in case of non-equivalence. Using equivalence as conformance relation, this approach was restricted to deterministic models, and did not support partial models.

Model-based mutation testing has since been applied to various different specification languages: Belli et al. [43, 42] applied model-based mutation testing to event sequence graphs and pushdown automata.

Hierons and Merayo [98] applied mutation-based test case generation to probabilistic finite state machines. The work presents mutation operators and describes how to create input sequences to kill a given mutated state machine.

Fraser and Wotawa [88] presented a model-checker based approach to model-based mutation testing, focusing on the requirement specification. They evaluate the approach by a property coverage criteria, to measure whether the resulting test cases cover all aspects of the specification. The evaluation is done on five examples from the automotive domain.

Weiglhofer and Wotawa [174] performed model-based mutation testing using the input output conformance relation ioco, working with LOTOS specifications.

El Fakih et al. [81, 80] proposed a model-based mutation testing approach based on finite state machines. They use deterministic mutation machines, that contain all possible mutants according to a user-defined fault model and generate test-cases covering these mutants. Petrenko et al. [142] define mutation machines for finite state machines as well, using them for mutation analysis of a given test suite. Koufareva et al. [111] extended the fault-based test-casse generation for finite state machines, allowing the implementations to consist of more states than its specification, and possibly reducing the length of the created test cases. Contrary to that work, Simão and Petrenko [68] extended the approach, by allowing the implementations to have a smaller state space than the specification, which reduces the state space for test-case generation.

Henard et al. [92] applied mutation-based testing to the testing of product line configurations, where they use mutations of the feature models to select interesting product line configurations for testing.

Sampaio et al. [148] propose a compositional input output conformance for the CSP process algebra, where they specially mention that this could be used for mutation-based test-case generation. However, no such experiments were performed yet, as they concentrated on checking the conformance between the specification and the actual implementation.

Recently, model-based mutation testing was also applied to the specification language Circus by Alberto et al. [26]. Circus is a textual specification for processed, based on Z and CSP, with semantics based on UTP. In the paper they provide a complete theory for model-based mutation testing, based on refinement. They provide automation via a Java prototype tool and present experimental results on a cash machine.

Jia and Harman [103] provide a survey on mutation-testing, which also covers model-based mutation testing, and gives a very good overview on the technique.

Bernhard Aichernig, my supervisor and leader of our research group, contributed to model-based mutation testing in several different areas:

In his first publication on that topic, Aichernig [1] introduced mutation testing to pre/post condition contracts. He used the refinement calculus for expressing the theory of mutation testing, and discussed the benefits of contract mutation, compared to program mutation, including the possibility of testing under refinement, if the specification is refined during development. Then, Aichernig and He [9] explored mutation testing for designs in UTP, where they viewed test cases as specification predicates, which allows to define refinement between tests and the specification. This work was the first testing theory for UTP, and also the first work on mutation of non-deterministic contracts. The first implementation on test-case generation via mutated contracts was presented by Aichernig et al. [22] for UML contracts in the OCL language. Later, Krenn and Aichernig [112] also applied the technique to Spec# contracts, using a combination of the verification generator Boogie and Z3 for solving the conformance check. Contrary to the definition in our requirement interfaces, the contracts in the above mentioned works were not synchronous.

The work on UTP designs was also extended to Reo connectors [130]. Reo is a channel-based modeling style, where models consist of simple connectors, which are composed together to build more complex ones. By modeling the connectors as UTP designs, the authors could design a fault-based test-case generation approach for connectors, which they support with a prototype implementation.

Aichernig and Corrales Delgado [8] performed model-based mutation testing, by generating test purposes for each model mutant. While this is very closely related to our approach for requirement interfaces, is was performed on asynchronous models, and did not concentrate on real-time properties. Using the test purpose technique, Aichernig et al. [24] also developed a slicing method to prune away the parts of the search tree that do not contain the mutation, which is a very similar approach than the partial models we suggested for timed automata in Chapter 6.

Model-based mutation testing has also been applied in our research group to UML models [3, 6] and action systems [5]. In the UML approach, both the specification and the mutants were translated to action systems. The conformance check, which was either refinement or ioco, was then performed on the action systems. The used tool is called MoMuT::UML, and supports both an explicit and a symbolic backend [115].

Model-based mutation testing was also applied to hybrid systems by Aichernig et al. [4], leading to the development of qualitative action systems [7]. Test cases produced by this technique additionally contain expected values of the environment after their execution.

Most of this work is summarized in the habilitation thesis by Bernhard Aichernig [2]. Additionally, the work led to the PhD thesis by Harald Brandl [59] on hybrid systems and the PhD thesis by Elisabeth Jöbstl [105] on model-based mutation testing with constraint and SMT solvers.

13.2 Theory of Timed Automata

Timed Automata were introduced by Alur and Dill in 1994 [29] and in the years since there has been ongoing work on both the theoretical and the practical aspects of timed automata. A recent survey by Waez et al. [171] lists forty different tools that use timed automata in the context of code development and verification. The survey identifies eleven classes of timed automata and almost eighty concrete variants belonging to those classes.

Already Alur and Dill [29] showed that non-deterministic timed automata are more expressive than deterministic ones, a result that was later also confirmed by Tripakis [162] and Finkel [86]. Bérard et al. [50] showed that timed automata with silent transitions are even more expressive, but also showed that silent transitions without clock resets can be removed without changing the language of a timed automaton. One class of determinizable timed automata, namely *Event-Recording Timed Automata*, was already found by Alur et al. [30]. Some other classes of determinizable timed automata were identified by Baier et al. [39]. Suman and Pandya [156] showed that *Integer Reset Timed Automata with Silent Transitons* can be determinized. These are timed automata where transitions that reset clocks can only occur at integer valued time points. This restricts the timed language of the automata, but may be very efficient in applications where the exact timing of the transition can be neglected.

Alur et al. [32] also gave a survey of various decision problems of reachability, language inclusion and language equivalence for timed automata and its variants.

The main inspiration to our work with regards to silent transition removal and determinization comes from Bérard et al. [50] and Baier et al. [39]. Bérard et al. [50] show that silent transitions extend the expressive power of timed automata and identify a sub-class of timed automata with silent transitions for which silent transitions can be removed. By restricting ourselves to the bounded setting, we can remove silent transition of all strongly-responsive TA_ϵ. In addition, our approach for removing silent transitions preserves diagonal constraints in the resulting automaton, thus avoiding a potential exponential blow-up in the size of its representation. Bouyer et al [58] discuss the practical advantages of preserving diagonal constraints in timed automata. Baier et al. [39] propose a procedure for translating non-deterministic timed automata to *infinite* deterministic trees, and then identify several classes of non-deterministic timed automata that can be efficiently determinized into finite deterministic timed automata. In contrast to our work, their procedure works on the region graph, which makes it impractical for implementation. In addition, we also allow in our determinization procedure disjunctive constraints which results in a more succinct representation that can be directly handled by the bounded model-checking tools. Both Bérard et al. [50] and Baier et al. [39] tackle non-determinism and observability in TA from a general theoretical perspective. We adopted the ideas from these papers and developed an effective procedure for the bounded determinization of NON-DET(TA_ϵ).

Bertrand et al. [54] develop a game-based method for determinization of NON-DET(TA_ϵ) which generates either a language equivalent DET(TA) when possible, or its approximation otherwise. A similar approach is proposed by Bertrand et al. [53] in the context of model-based testing, where it is shown that their approximate determinization procedure preserves the tioco relation. In contrast to our approach, which is language preserving up to a bound k, and thus appropriate for bounded model checking algorithms, determinization in the above-mentioned papers introduces a different kind of approximation than ours.

Wang et al. [173] perform language inclusion on timed automata. Their procedure involves building a tree, renaming the clocks and determinization of the tree. Contrary to our work, they do not restrict themselves to the bounded setting, thus taking the risk that their algorithm does not terminate for some classes of timed automata. They also use the ”standard” determinization method that involves splitting non-deterministic transitions into a possibly far larger set of deterministic transitions, whereas we join them into one transition.

De Wulf et al. [74] proposed an antichain-based algorithm for checking several problems on finite automata, including a trace-inclusion check that does not need to explicitly determinize the automata first. Iosif et al. [102] extended the trace inclusion algorithm via antichains to infinite state systems. While timed automata can be expressed as such systems, one would also need to change their language, so that the values of all clocks is made observable after each step. Additionally, the algorithm does not consider silent transitions and termination is not guaranteed.

13.3 Test-Case Generation for Timed Automata

Model-based test-case generation from timed automata has been done in several approaches:

Nielsen and Skou [134] proposed a test case generation framework for non-deterministic (but determinizable) timed automata, implemented in the prototype tool RTCAT. Contrary to our approach, they do not use mutants to steer test-case generation and they can only process determinizable timed automata. Their coverage criteria are based on equivalence classes of the state space, where for each equivalence class, certain observations have to be seen at least once.

The UPPAAL tool family contains a series of tools working with timed automata. There are three UPPAAL tools used in the context of testing: UPPAAL Cover [97] generates tests offline and provides means for specifying observers to generate tests satisfying pre-defined coverage criteria. Cover requires the specification to be deterministic. UPPAAL Tron [131] is used for online testing, where inputs and delays are chosen non-deterministically and executed on the SUT and the specification simultaneously and all outputs that are received from the system are checked for conformance on the model. UPPAAL Yggdrasil [109] is the newest testing tool in the UPPAAL family, facilitating offline test-case generation, with the advantage of adding test scripts to transitions, that are added to the tests during generation. The resulting tests can thus be executable scripts or function calls in any desired language. None of the tools in the UPPAAL family performs mutation-based test-case generation. Note that while we model our automata in UPPAAL, the automata produced by our determinization and silent transition removal procedure can not be analysed with UPPAAL anymore, as they contain disjunctions. They can, however, still be opened and viewed.

Springintveld [154] proposed a test-case generation procedure for timed I/O automata, where the automata are reduced to *grid automata* with discrete time information. They claim that their test suite grows too big for practical purposes, but is complete with respect to the grid automata. Contrary to our approach, they do not create test cases with dense time, and do not apply mutation-based testing.

Cardell-Oliver [63] introduced a method for conformance testing between implementations and timed automata specifications. He translates timed automata into *testable timed transition systems* and creates test views to separate the parts that shall be tested, while hiding the transitions that are not part of the test view. Contrary to our work, he only considers four types of faults, that are not especially designed for catching real-time errors and tests for trace equivalence, instead of tioco-conformance.

Krichen and Tripakis [117] produce deterministic testers for non-deterministic timed automata in the context of model-based testing. They restrain the testers to using only one clock, which is reset upon receiving an input. The testers are sound, but not in general complete and might accept behavior of the SUT that should be rejected.

Marinescu et al. [125] developed a framework for testing automotive embedded systems, where the main model is an EAST-ADL specification, and individual components are modelled as timed automata with UPPAAL Port. They can verify the models according to timed computation tree logic formulas and generate test cases according to this formula via UPPAAL Port. The tests are then automatically translated into executable python scripts.

Vulgarakis et al. [169] provided formal semantics for the ProCom real-time component model, where the semantics are given by an extended finite state machine language. These finite state machines can be translated to timed automata with urgency and priorities. Systems with several components are naturally expressed by networks of timed automata. Tools like UPPAAL or TDSpin can then be used for the verification of ProCom models. Another closely related formalism that supports automated translation into (priced) timed automata is called REMES [152], which is a ressource model for embedded systems. REMES allows the annotation of ressources, to express and solve resource analysis problems, like e.g. optimal/worst-case ressource analysis.

Model-based test-case generation for real-time systems has been performed with various models other than timed automata. Some have been summarized by Nilsson [135], where he separates the methods into methods based on Process Algebra, Finite State Machines, Temporal Logic, Petri Nets and Informal Methods.

Several different approaches to model mutation have already been published, using Finite State Machines [83, 143], Kripke structures [56] or Event Sequence Graphs [43]. Nilsson et al. [136] introduced mutation operators for *timed automata with tasks*, yet the mutation operators there concentrate on tasks and timeliness and not the core essence of TA. Our mutation operators 6 and 8 from Section 3.1 are specific to TA, while the other ones are similar or closely related to the operators described by Ferraz Fabbri et al. [143].

Krenn et al. [113] proposed an incremental language inclusion check for networks of timed automata. Instead of checking the language inclusion for the complete networks, they check it for two single automata of both networks. If they find a witness to the language inclusion, they expand it (if possible) to a counter example for the complete networks. The approach can be used for incremental tioco-conformance checking of networks of timed automata, and thus for efficient model-based mutation testing.

13.4 Other Variants of Timed Automata

In the following, we would like to shortly discuss different variants of timed automata, even though in this thesis we only used the variants discussed in Section 2.1. The definitions are adapted from the survey of timed automata by Waez et al. [171].

- **Timed Automata with other Clock Constraints**

  Timed automata with other clock constraints denote *timed automata with weakened guard definitions*. Waez. et al. give three subcategories of these class: periodic clock constraints, additiv/multiplicative clock constraints and parametric timed automata.

  Periodic clock constraints denote constraints that are enabled at certain intervalls, like when the value of a clock is odd. *Automata with periodic clock constraints* [67] (and periodic clock updates) can express timed automata with silent transitions [77].

  Timed automata with additive/multiplicative clock constraints [49] are closely related to timed automata with diagonal constraints, but allow addition and multiplication of clocks. Theoretic results on these automata vary strongly depending on the number of clocks and used constraints. The results for reachability are presented in Table 5 of the survey.

  Parametric timed automata [31] allow the parametrization of clock constraints. Instead of concrete constraints like $x > 4$, the numeric values can be replaced by parameters (e.g. $x > p$), changing the reachability problem to either ”does there exist a parameter valuation, so that a certain location can be reached” or ”for a certain parameter valuation, can a certain location be reached?”.

- **Timed Automata with Clock Updates**

  Timed automata with clock updates can reset clocks to values other than zero. These may be updatable timed automata, suspension automata or integer reset timed automata.

  In *updatable timed automata* [57], clocks may either be set to a fixed value, or, with a constraint of the form $x :> 5$, may be non-deterministically set to any value satisfying the constraint.

  Suspension automata [129] either suspend the passage of time in certain locations, or use stopwatch-like clocks, that enable the subtraction of integer values from clocks, to subtract the time that was spent in a certain location. McManis and Varaiya [129] showed that the language inclusion problem for suspension automata is decidable. They use hybrid automata with rate 1 to model the suspension automata.

Integer reset timed automata [156] only reset clocks when they have integer values. It is noteworthy, that it is decidable to check whether the language of an integer reset timed automata is included in the language of a classical one.

- **Timed Automata with other Clock Rates**

  Timed Automata with other clock rates let different clocks progress at different rates. If the clock rates are defined and can not change, the corresponding automata can also be expressed by a classic timed automata [94]. If the clock rates may vary during execution, each clocks needs to be reset every time its rate changes. In that case, it can still be expressed by a classic timed automaton, and reachability is decidable, otherwise it is not.

- **Timed Automata with Resources**

  Timed automata with resources can be used to model resource consumption additionally to the usual timing constraints. The most well-known timed automata with resources are *priced / weighted automata* [33, 41].

  The consumption of ressources is annotated both for the transitions and for the locations, where in location the consumption usually rises with passing time, while for each transition a fixed amount is added. Consequently, each run through the automaton is assigned a price / weight. Using these prices, one can create reachability problems like how to reach a certain location with minimal cost.

  Another timed automata variant with resources are *timed automata with real-time tasks* [137], where transitions may be assigned tasks, containing a best-case execution time and a worst-case execution time. Thus, transitions are no longer instantaneous. As the worst-case execution time might be set to zero, these automata are at least as expressive as classic timed automata.

- **Timed Automata with Probabilities**

  In *timed automata with probabilities* [28], transitions may be assigned a probability, stating its likelihood of occurring. They can be used to estimate performance parameters like throughput or mean service time. They can also be used to specify soft real-time properties, like ”at most 5% of the runs through the system take longer than 10 seconds”.

- **Timed Automata with Communication**

  Timed automata with communication [120] are timed automata extended by channels that enable communication, e.g. in a network of timed automata. The timed automata with inputs and outputs that are used in the most parts of this thesis are an example for communicating timed automata.

- **Timed Automata with Determinizability**

  This subclass of timed automata contains those classes of timed automata that are known to be determinizable. These are for instance *event-clock automata* [30]. Each action is associated with an event-recording and an event-predicting clock, recording the time since the action last occurred and predicting the time until its next occurrence. By definition, event clock automata do not contain any silent transitions.

  Timed automata with integer resets were proven to be determinizable as well [157]. Another variant of determinizable timed automata,*strongly non-Zeno timed automata*, was proposed by Baier et al. [39].

- **Timed Automata with Self-Embedded Recursion** ...

  Timed automata with self-embedded recursion, e.g. *pushdown timed automata* or *recursive timed automata* [163], let each state of the automata to consist of another automaton, that might even be itself. Each such state contains a marked entry and a marked exit state. Clock values may be passed between the different automata, enabling the ”inner” automata to reason about the global time.

- **Timed Automata with Succinctness**

  These classes of timed automata focus on easier modeling, instead of easier analysis. Thus, they offer additional modeling elements which are equally hard or harder to analyze, but offer a more comprehensive modeling style. One example are *timed automata with urgency* [40]. According to different definitions, they may either contain urgent locations, which must be left as soon as possible, or urgent transitions, which have to be taken within a specified time intervall (deadline) after being enabled, and have priority over non-urgent transitions.

- **Timed Automata with Games**

  Timed game automata [124] have been designed to model open real-time systems. The game reachability problem for these automata focuses on whether the system has a strategy to reach a target state, regardless of the decisions of the environment.

We decided to model with timed automata with inputs and outputs for two reasons. Most importantly, the separation into inputs and outputs is needed for testing, to separate which events are triggered by the environment, and which are part of the system under test. The second reason was the tool support by UPPAAL, which was already used in MBAT and allows us to analyze the specifications before starting the test-case generation.

13.5 Symbolic Semantics of Timed Automata

There exist two symbolic representations for timed automata, that convert the infinite state-space of timed automata into a finite one, while preserving most properties of the automata. The two representations are called *regions* and *zones* [27]. While regions are mainly used for calculating and proofing decidability results, zones are used in many of the available tools for timed automata. Using our bounded model-checking approach, we can work directly on timed automata, as the infinite state-space does not pose a problem. However, we will now shortly discuss the two representations, as they enable interesting alternatives to bounded model-checking.

13.5.1 Regions

A *region*, which is a state in a *region graph*, is defined as a tuple $\langle l, r\rangle$, s.t. l is a location and r is a set of clock valuations, called a *clock region*. r is defined in a way, so that for every clock x and any two clock valuations v, v', the integral value of $v(x)$ equals the integral value of $v'(x)$, and that for all clocks the order of the fractional parts is preserved. As all constraints only use integral values, the integral values suffice for checking whether a clock constraint is fulfilled, while the order on the fractional part is needed to know, which clock will change its integral value first.

Two regions $\langle l, r\rangle$ and $\langle l, r'\rangle$ may be merged, if for all constrains C in the invariants of l and the guards of the transitions leaving l, r satisfies C iff r' satisfies C.

The number of regions grows exponentially with the number of clocks and the value of the highest constant in the timed automaton.

13.5.2 Zones

Zones are also defined by tuples $\langle l, Z \rangle$, where Z is a conjunction of general clock constraints, and the zone is defined to contain all clock valuations satisfying Z.

Zone can be used for forward and backward analysis of timed automata systems. The forward analysis starts in the initial location, with all clocks initialized to zero, and its constraint being a conjunction of the invariant of the initial location, and the constraint $x_1 = x_2 = \cdots x_n$ for all clocks $x_k \in \mathcal{C}$. From there, we compute all reachable computations, until all final locations are found, or the algorithm terminates, even though termination is not guaranteed. During the exploration, the constraints on the zones are updated according to clock resets, guards and invariants.

For a backward analysis, one starts with the final states, and all clocks unrestrained, and explores the predecessor zones, trying to reach the initial location.

Zones are usually represented by DBMs (*Difference Bound Matrices*), which facilitate performing tests like emptiness checking or the comparison of zones syntactically on the DBMs. While this improves computation efficiency a lot, these representations can not be used for timed automata containing diagonal constraints or disjunction. However, Bouyer et al [58] introduced an extension to DBMs, that allows diagonal constraints.

Most major timed automata tools, like e.g. UPPAAL [120] and Kronos [72], use zones. As already mentioned, our intention was to use bounded model-checking for the test-case generation, which enables us to perform analysis directly on the timed automata, and enables the usage of additional modeling elements, like disjunction.

13.6 Synchronous Systems

Synchronous languages were introduced in the nineteen-eighties, and mostly driven in France, where the most well-known three synchronous languages were developed: Lustre [64], Signal [89] and Esterel [52]. In 1991, IEEE devoted a special issue to synchronous systems, featuring e.g. a paper by Benveniste and Berry [44], discussing the major issues and approaches of synchronous specifications of real-time systems. A decade later, Benveniste et al. [47] gave an overview on the development of synchronous languages during that decade, especially mentioning the rising tool support and the industrial acceptance. They also mention globally asynchronous, locally synchronous systems [158] as an upcoming trend.

The main inspiration for our requirement interfaces was the introduction of the conjunction operation and the investigation of its properties [78] in the context of synchronous interface theories [66]. While the mathematical properties of the conjunction in different interface theories were further studied by Benveniste et al. [45], Reineke and Tripakis [146] and Henzinger and Ničković [95], we are not aware of any similar work related to model-based testing. Requirement interfaces differ from the *Moore interfaces* and *bidirectional interfaces* introduced by Chakrabarti [66] in two main ways: they support internal variables, and they are designed to build a strong linkage to the original informal requirements.

Synchronous data-flow modeling [48] has been an active area of research in the past. The most important synchronous data-flow programming languages are Lustre [64] and SIGNAL [89]. These languages are implementation languages, while requirement interfaces enable specifying high-level properties of such programs. Testing of Lustre-like programs was studied by Raymond et al. [145] and Papailiopoulou [141], using SCADE. The specification language SCADE [51] supports graphical representation of synchronous systems. Internally, SCADE models are stored in a textual representation very similar to Lustre. Experimental results for test-case generation were presented by Wakankar et al. [172] for experiments where the manually translated SCADE models to SAL-ATG models, and used the SAL-ATG for the test-case generation.

Compositional properties of specifications in the context of testing were studied before [166, 162, 148, 25, 69]. None of these works consider synchronous data-flow specifications, and the compositional properties are investigated with respect to the parallel composition and hiding operations, but not conjunction. A different notion of conjunction is introduced for the test-case generation with SAL [90]. In that work, the authors encode test purposes as trap properties, and conjunct them in order to drive the test-case generation process towards reaching all the test purposes with a single test case. Consistency checking of contracts has been studied by Ellen et al. [82], yet for a weaker notion of consistency.

Our specifications using constraints share similarities with the Z specification language [153], that also follows a multiple-viewpoint approach to structuring a specification into pieces called schemas. However, a Z schema defines the dynamics of a system in terms of operations. In contrast, our requirement interfaces follow the style of synchronous languages.

Seceleanu and Seceleanu [151] proposed a synchronization mechanism for action systems. They achieve the synchronization by introducing a new parallel composition operator, which performs rounds of internal actions, covering all possible interleavings of the individual action systems, before synchronously producing all global variables and reaching a new global state of the action system. They also extend the approach to continuous and timed action systems.

Brillout et al. [60] performed mutation-based test-case generation on Simulink models. They implemented the approach in the tool COVER, based on the model-checker CBMC. He et al. [91] exploit similarity measures on mutants of Simulink models, to decrease the cost of mutation-based test-case generation. They provide experiments to show the advantages of model-based mutation testing compared to random testing, and compared to simpler mutation-based testing approaches.

There exist several tools for test-case generation for synchronous systems. The tool Lutess [79] is based on Lustre. It takes the specification of the environment (specified in Lustre), a test sequence generator, and an oracle and performed online testing on the system under test according to the environment and traces selected by the generator according to several different modes. Another tool based on Lustre is called Lurette [145]. Lurette only performs random testing, but is able to validate systems with numerical inputs and outputs. A third testing tool based on Lustre is called GATeL [126]. It generates tests according to test purposes, using constraint logic programming to search for suitable traces.

The tool Autofocus [100] facilitates test-case generation from time-synchronous communicating extended finite state machines that build a distributed system. It is based on constraint logic programming, and supports functional, structural and stochastic test specifications.

14 Summary and Conclusions

Within this thesis we have shown how model-based mutation testing can be applied to real-time systems, discussing both synchronous and asynchronous models. This chapter is ment to summarize the presented work, illustrate the contributions, give conclusions while answering the research questions and provide an outlook to possible future extensions.

14.1 Summary

In the first part of this thesis, we started with an introductory chapter, that presented our motivation for testing real-time systems, which is mainly based on the vast amount of time-sensitive and safety-critical components in the automotive domain. Then we explained the concepts of model-based testing and model-based mutation testing, giving clear priority to the latter, as it was the main topic of this thesis. The main workflow of model-based mutation testing, independently on the used formalisms, consists of three steps: (1) mutating the specification, to receive a set of faulty models (mutants), (2) checking for conformance between the specification and the mutants, and (3) in case of non-conformance, build a test case that can guarantee to detect the corresponding fault in deterministic systems. At the end of the introductory chapter, we established our research questions and gave an overview of the publications that led to this thesis.

We started with Part I, which is the part about introducing model-based mutation testing to asynchronous real-time systems, by giving basic definitions of timed automa and timed conformance relations. We picked timed automata as the exemplary formalism for asynchronous models, as they are among the most well-known formalisms for specifying real-time systems, and are supported by a well defined and explored theory.

To apply model-based mutation testing to timed automata we had to take care of several steps: first, we defined mutation operators for timed automata, focusing on introducing time-dependent faults. Then, we showed how the tioco-conformance check can be expressed as a language inclusion problem, which can then be expressed as an SMT-formula and solved by bounded model-checking. The produced SMT-formula was designed to detect counter-examples to the language inclusion, i.e., it tried to find a reachable location, in which the mutant can perform an output that is not possible in the specification. If such a location is found, we produce a test case with symbolic time delays that leads there. The complete workflow was implemented in a tool we call MoMuT::TA.

The main restriction of this approach was that it was only defined for fully-observable timed automata, and that it may produce spurious counter-examples for non-deterministic timed automata. Thus, in the next chapter we investigated how to remove both silent transitions and non-determinism from our models. It is well known that silent transitions and non-determinism add to the expressiveness of timed automata, and cannot be removed in general. We thus introduced a bounded approach that cuts all traces after a specified number of observable steps. The resulting timed automaton, which is in the form of a tree, resets exactly one clock per transition and all transitions of same depth reset the same clock. Due to the bounded traces, the silent transitions can be removed and the tree can be determinized effectively. The silent transition removal traverses through the state space of the tree searching for every silent transition. For each of them, it creates a bypass transition, that is, a transition combining the last observable transitions before the silent transition, and the silent transition itself. Additionally, for the complete subtree below the silent transitions, all future guards that refer to the clock that was reset on the silent transition need to be updated and synchronized. Once all silent transitions are removed, we apply a determinization step, that combines non-deterministic transitions via disjunction. While disjunctions in guards do not conform to the standard definition of timed automata, we exploit both diagonal constraints and disjunction, in order to reduce our state-space. Additionally, we implemented an on-the-fly algorithm that only needs to explore the state-space once to perform the unfolding, the silent transition removal and the determinization in one step. This on-the-fly algorithm was then expanded, to also include building the product of networks of timed automata during the unfolding.

In Chapter 5 we presented the experiments we performed for evaluating the presented approaches. As a continuous example we presented several versions of a car alarm system, including a deterministic version, a non-deterministic one with silent transitions, and a version with data variables. The test-case generation from the deterministic version produced a test suite strong enough to reach a 100% mutation score on a Java implementation that was mutated via the tool μ-java. The non-deterministic model grew so big during the determinization, that test-case generation became infeasible. We thus had to split the model into two partial models, to perform the test-case generation. While test-case generation of the complete test suite from the deterministic model needed 30 minutes, the approach on the two unfolded partial models took 43 minutes, which illustrates the complexity introduced by state-space increase caused by the unfolding.

As a second industrial example we presented an automated speed limiter, where we both had a deterministic model and one that contained both non-deterministic and silent transitions. We used the first model to assess the quality of an existing test suite, and generated additional test cases only for the mutants that were not killed by the previous test suite. We expanded the existing test suite by additional eleven test cases. For the non-deterministic model, we performed our complete workflow, including silent transition removal, determinization and test-case generation. The complete approach took 5.4 hours and produced 128 test cases. Unfortunately, the quality of the test-cases could not be evaluated, as we did not have access to the real system under test. However, we also created test cases for a more abstract deterministic version of the speed limiter, and the resulting test cases were analyzed by Chalmers and Volvo. Despite detecting some previously untested functionality of the system, the system engineers that manually inspected the tests and mutants showed high interest in the mutants, using them as validation why the corresponding test cases make sense.

We concluded Part I by presenting several smaller approaches we performed with timed automata: first, we investigated how the model mutants we create during the test-case generation can be used to debug a system under test. The approach is supposed to both hint on the location of the bug, and select a subset of the model mutants that reflect the possible faults that might be contained in the implementation. Then, we showed how to build tioco-conform partial models during the determinization approach. This approach enables the efficient test-case generation from determinized models, that might otherwise exceed the capabilities of MoMuT::TA. Finally we showed how timed automata can be encoded as timed action systems, and compared MoMuT::TA to a tool based on symbolic execution, developed by Martin Tappler.

In Part II we presented our work on synchronous systems, illustrated on requirement interfaces, which are a synchronous formalism we proposed within the MBAT project. Requirement interfaces provide the possibility to separately specify different views on a system, e.g. the functional view can be modeled separately from the power consumption. This enables an easier test-case generation as well, where the test cases of one view can then be extended using a second view.

First, we defined requirement interfaces, providing syntax and semantics, as well as discussing general attributes like consistency, refinement between requirement interfaces and their conjunction.

Then, we demonstrated how to perform a bounded consistency check on requirement interfaces, using SMT-solvers. However, due to the quantifiers in the formula, the state-space of the consistency check grows exponentially, and does not support very high bounds. Thus, we focused more on the test-case generation, where we introduced a methodology driven by test purposes (the variable valuations we try to reach). We also defined how test cases produced according to one view of the system can be expanded using a second view and demonstrated it on our running example, the abstract specification of a buffer. Additionally, we showed how requirement interfaces facilitate an easy way to be linked to the textual requirements they were built from, which supports traceability. At the end of the section, we developed a model-based mutation testing methodology for requirement interfaces. This included the definition of mutation operators for requirement interfaces, and the theory on how to automatically create test purposes that lead to the mutations, if these introduce faulty behavior. As we wanted to express the conformance check via test purposes, we only introduced weak mutation testing. Consequently, the generated test suite is of slightly lower fault detection capability than the test suites produced by our asynchronous approach, as the tests only lead to the mutation, but not necessarily to the observable failures caused by the mutation. All introduced methodologies, the consistency check, the test-case generation and the model-based mutation testing, were implemented in a tool called MoMuT::REQs.

In the following section, we showed one iteration through a typical development process, from the textual requirements up to the test-case execution on a (simulated) system under test. The process is built up around a central data management system that stores and links all work products that are created along the development. Communication between MoMuT::REQs and this system is implemented via the OSLC standard, providing high interoperability. The presentation of this workflow is used to give details on one of our industrial case studies, the safing engine of an airbag, and to show the benefits of traceability.

Next we investigated the suitability of requirement interfaces for real-time systems. We differentiated between two types of real-time systems: those conforming to the synchrony hypothesis, which states that all outputs can be produced before the next inputs arrive, and those not conforming, where outputs arrive delayed and other inputs may be arriving in between. For the former, requirement interfaces, just like all synchronous languages, are very well suited. Time can be completely neglected from the specification, as all outputs are considered to arrive instantaneously. This reduces the complexity of the specification, both with regards to comprehensibility and with regards to the automated test-case generation. For the latter, we investigated two modeling styles that can take care of delayed outputs, one with explicit ticks for measuring time, and one with symbolic time delays. Explicit ticks are very well suited for modeling systems depending on clock cycles that continuously interact with the environment. However, for testing long timed delays, the test cases become huge, simply since every time unit that needs to be waited is another step in the test case. This poses a problem for the test-case generation, as we need higher search depth. However, afterwards these delays can be expressed as integer values, to decrease the size of the test case. Symbolic timed delays take care of that problem already during the test-case generation, but change the notion of time in a way that completely freezes the system while waiting. While this cannot be used to model embedded systems that are ment to continuously react to environmental changes, it is a promising approach for modeling systems where we can regulate all inputs to the system under test, like the car alarm system, where we can decide when to open and close the doors.

We evaluated our approaches on three industrial studies: The car alarm system, the safing engine and the speed limiter.

First, we generated test cases from manual test purposes for the safing engine, generating a test suite that covers every possible value of every signal at least once. We evaluated the test suite with a set of faulty Simulink models, receiving a mutation score of only 51%. We then inspected the test suite, and manually added ten more test purposes, to receive a 100% mutation score. However, the manual inspection was time consuming, and needed several iterations.

Thus, in the next experiment we applied model-based mutation testing to the safing engine and executed the produced test suite on the faulty Simulink models. We gained a mutation score of 88%, where simply trying to reach all possible variable values once only lead to a 51% coverage. For the automated speed limiter, where we did not generate manual test purposes and only performed the model-based mutation testing, we even reached 98% mutation score on a Java implementation mutated via the Major mutation tool.

We applied the model-based mutation technique to two versions of the car alarm system, one with explicit ticks and one with symbolic ticks. The mutation scores of the corresponding test suites on the previously mentioned faulty Java implementations was comparatively low, with 60% and 68%. This was caused by three facts: A rather coarse modeling style, the weak mutation testing and the execution of synchronous test suites on an asynchronous system.

We concluded Part II with an overview on related work in the context of model-based testing of synchronous systems.

In the last part of the thesis we brought the two different directions of Part I and Part II together, by first summarizing the different advantages of the two modeling styles, and then discussing how the two styles can be combined to form globally asynchronous and locally synchronous systems. We showed in detail how different components of the car alarm system can be modelled via requirement interfaces, and connected via a global timed automata model.

Then, we gave a summary of the work presented in this thesis, and now we will concentrate on presenting the contributions, drawing some conclusions and pointing towards future work.

14.2 Contributions

We would like to state once more the contributions of this thesis:

- We developed a model-based mutation testing approach (see Chapter 3) for deterministic timed automata. We encoded the tioco-check via language inclusion, and developed a set of mutation operators, some of which were developed to cover real-time aspects. We developed the theory behind the approach, and implemented it in the tool MoMuT::TA using bounded model-checking.
- We introduced and implemented a method for bounded determinization and silent transition removal of timed automata (see Chapter 4), to enable the processing of non-deterministic systems with our test-case generation approach. Even though in general timed automata cannot be determinized or made fully observable, both approaches work for general timed automata, since we bounded the traces. This technique can be used for analysis of the tools as well, but is especially useful for test-case generation, where we only consider finite traces anyway.
- To improve the efficiency of the determinization, we developed an on-the-fly algorithm (see Section 4.5), which supports networks of timed automata and produces a deterministic bounded unfolding of the product of all automata in the network, while only going through the state-space once. This significantly reduces the runtime of the approach.
- We showed how the model-mutants we create for model-based mutation testing can also be used for debugging (see Section 6.1), where we select a subset of mutants that correspond to a faulty system under test and analyze which kind of implementation error may correspond to the selected mutations.
- We introduced requirement interfaces (see Chapter 7), which are a synchronous contract-based specification language for synchronous data-flow systems and showed how to build their conjunction, how to check their consistency and how to generate tests from them, using either manually designed or automatically generated test purposes.
- We extended the test-case generation for requirement interfaces, to support model-based mutation testing (see Section 8.3), by automatically generating test purposes for mutants, enabling weak mutation testing.
- We discussed how time is treated in synchronous languages and how discrete delays can be integrated into requirement interfaces (see Chapter 10). We proposed both an explicit and a symbolic approach for delayed outputs.

- We implemented both approaches based on bounded model-checking and evaluated both the synchronous and the asynchronous test-case generation on industrial use cases (see Chapter 5 and Chapter 11) and reported and discussed the empirical results.

14.3 Conclusions

In the introduction we defined four main research questions and 12 subquestions for this thesis:

- **Q1: Can real-time systems be tested with model-based mutation testing?**
  - **Q1.1**: Can we support non-determinism for model-based mutation testing?
  - **Q1.2**: Can we support internal transitions?
  - **Q1.3**: What kind of mutation operators reflect violated timing properties?
- **Q2: Can bounded-model checking be applied to test real-time systems?**
  - **Q2.1**: How can we encode the conformance check via bounded-model checking?
  - **Q2.2**: How should we implement the bounded model-checking?
  - **Q2.3**: Is bounded model-checking an efficient approach?
- **Q3: Can the approach be applied to both synchronous and asynchronous systems?**
  - **Q3.1**: What modeling languages should we use as representations for the individual modeling styles?
  - **Q3.2**: How well do timing properties fit in the individual styles?
  - **Q3.3**: How can the two approaches be combined?
- **Q4: Can the approach be effectively performed on industrial use cases?**
  - **Q4.1**: How big does the state-space of the models become?
  - **Q4.2**: How can we reduce the state-space explosion?
  - **Q4.3**: Are the reachable search depths of our bounded model-checking approach sufficient for industrial case studies?

We will give our answers to these three questions in a backward order, starting with question $Q4$.

Q4.1: How bad does the state-space explosion become? As for all model-checking problems, the state-space increases exponentially with the search depth. While this problem is decreased for requirement interfaces by the reduced search depth due to simultaneous inputs, it still poses a problem for applications where this is not possible. For timed automata, the problem especially arises for non-deterministic models, that need to be explicitly determinized.

Q4.2: How can we reduce the state-space explosion? However, the search space can be reduced in several ways: we investigated both manually and automatically generated partial models for timed automata (see Section 6.2 and Section 6.3), which enables us to process models that were infeasible otherwise. For requirement interfaces, we showed that the complexity may be reduced by separating different views of the system, and dividing them into separate requirement interfaces (see Example 7.1 and Example 7.4), which can then be used to extend test cases from the main interface later on. Additionally, the synchronous nature of requirement interfaces reduces the state-space, as several signals can be processed simultaneously.

Q4.3: Are the reachable search depths sufficient for industrial case studies? We applied our approach successfully to three industrial case studies, that is, a car alarm system from a previous project, a safing engine of an airbag, which was a use case in the MBAT project, and an adjustable speed limiter that was a use case in the CRYSTAL project. We had to split the non-deterministic timed automata model of the car alarm system into two models, to be able to process it, and we had to pick a suitable level of abstraction for the speed limiter, but in the end, the test suites could be generated in reasonable time, could provide a high mutation score when executed on faulty implementations, and covered the complete functionality we intended to test. The experiments were presented in Chapter 5 and Chapter 11.

We thus conclude that the technique works efficiently enough to be applied to industrial case studies.

Q3.1: What modeling languages should we use as representations for the individual modeling styles? Picking timed automata as representative for asynchronous timed systems was an easy choice. Not only are they probably the most well-know formalism for real-time system specification, but also Aalborg University, one of the two universities developing UPPAAL, was a partner in the MBAT project. Thus, timed automata were used in MBAT already and using them let us build a deeper cooperation with other partners in the project. The choice of requirement interfaces also originated in MBAT, where we developed the requirement interface language in cooperation with our industrial partners at a face-to-face meeting, so we could adapt it for their personal needs.

Q3.2: How well do timing properties fit in the individual styles? Both formalisms are well suited to represent their class of systems, i.e. asynchronous and synchronous systems. While the car alarm system was easier to model as a timed automata model, the safing engine which is an embedded chip, was definitely better suited to being specified by a synchronous language. To summarize the experiences we gained from our experiments, timed automata are more well suited for specifying complex timing behavior, while requirement interfaces are only well suited for instantaneous outputs, our outputs are also triggered by discrete delays.

Q3.3: How can the two approaches be combined? We evaluated the approaches on several examples, presented a summary of their individual advantages, and showed how the two modeling styles may be combined to form an architecture that benefits from their individual advantages (see Section 12.2. While we saw a clear advantage in their combination, we did not perform any experiments, as the need did not arise in our projects.

We answer research question $Q3$ with the statement that both, synchronous and asynchronous real-time systems, can be tested vial model-based mutation testing, where asynchronous systems are better suited for modeling time delays, and synchronous systems improve the specification of systems with instantaneous outputs.

Q2.1: How can we encode the conformance check via bounded-model checking? In the part on asynchronous systems, we encoded a tioco-conformance check for timed automata via language inclusion (see Section 3.3.1). This works based on the fact that we made the specifications input enabled. We encoded the language inclusion as a bounded model-checking problem and expressed it as an SMT-formula. In the part on synchronous systems, we used refinement for our conformance checks. We implemented a way to automatically generate test purposes for mutated contracts, that can be used to detect non-conformance (see Section 8.3). These test purposes, together with the step relation of the requirement interface, can be used for a reachability analysis via bounded model-checking.

Q2.2: How should we implement the bounded model-checking? In both of the approaches, we used SMT-solving for solving our bounded model-checking problems. More specifically, we used Microsofts SMT-solver Z3. The representation as SMT-LIB formulas would also allow an easy integration of other solvers, but since Z3 worked very well for both approaches, this was never done.

Q2.3: Is bounded model-checking an efficient approach? We compared bounded model-checking to symbolic execution in Section 6.3.6, where we found out that symbolic execution can handle non-determinism better than bounded model-checking. On the other hand, higher numbers of clocks affects the runtime of symbolic execution far stronger than the runtime of the bounded model-checking. We also encoded language inclusion in UPPAAL, by adding a trap property to the product of the specification and a mutant. First experiments on the car alarm system models presented in Section 6.3.6 showed that UPPAAL is very fast in detecting non-conformance in the deterministic case. In the non-deterministic case, the encoding we used suffered from the same problem as the bounded-model checking: it lead to spurious counter examples. Adding a PIN code with a range of $0-500$ to the deterministic model already slowed the conformance check down and expanding it to 5000 made the whole approach infeasible. Thus, for pure timed automata, where difference bound matrices (DBMs) can be used for the checks, UPPAAL outperforms our approach, but with different modeling elements that are not supported by the DBMs, bounded model-checking performs a lot faster.

Q1.1: Can we support non-determinism for model-based mutation testing? We do support non-determinism for both types of models, where we ran into different challenges for each of them: timed automata need to be pre-processed (see Section 4), to remove the non-determinism before starting the test-case generation. This leads to increased complexity of the workflow, and increases the state-space for the test-case generation. However, underspecification, in the sense that we allow different outputs to leave a location, is supported without pre-processing. Requirement interfaces facilitate non-determinism without much additional complexity, but they do not support non-deterministic time delays.

Q1.2: Can we support internal transitions? Both formalisms support internal transitions (or in the case of requirement interfaces, the update of internal state variables). For timed automata this needs a pre-processing step (see Section 4), and thus once more increases the complexity. However, we could perform the test-case generation on all industrial examples we had in our projects.

Q1.3: What kind of mutation operators reflect violated timing properties? We mainly targeted off-by-one faults, where outputs arrive either a little too soon, or a little too late. Additionally, we targeted additional or missing clock resets, leading to timed outputs which arrive a lot sooner than expected (if the clock already was close before reaching the timing constraints, when it should have been reset) or not at all, if the timing constraint was already exceeded when the clock should have been reset, but was not. For the asynchronous deterministic adjustable speed limiter, we created tests that covered the timed internal state change that were not covered by the test suite in use (see Section 5.2.2). For the asynchronous car alarm system (see Section 5.2.1), we were able to detect all time related faults we manually inserted into the car alarm system. For the synchronous car alarm system (see Section 11.4), an investigation of the killed mutants showed that we covered all timing faults. Only for the synchronous model of the speed limiter (see Section 11.3.3), one timing fault could not be caught, due to being too coarse during test-case execution.

To sum up the conclusions, we were fairly satisfied with the empirical evaluations of both approaches. They are both well suited for their domains, provide adequate test suites in reasonable time, and can be applied to case studies of industrial size.

14.4 Future Work

As there is always room for further improvements, we will limit this summary of possible future work to only four main topics:

- **All-In-One Algorithm.** The processing of non-deterministic timed automata still suffers from scalability issues. Building an on-the-fly algorithm for the determinization and the silent transition removal definitely was a step in the right direction. However, the step can still be expanded. By adding the model mutator to this algorithm, one could achieve several goals: first, it might be possible to only perform the determinization once, instead of determinizing all mutants separately, or at least use the tree of the specification for those parts not mutated. Secondly, the position of the mutation in the determinized tree could be marked, enabling us to perform a reachability analysis to that location, and to perform the *tioco*-check from there.

To expand the algorithm even further, the conformance check could / should be integrated as well. This would benefit the *tioco*-check starting from the mutation and reduce the amount of disk accesses needed during an execution. The improvements of such an algorithm to the complete model-based mutation approach for timed automata would probably be comparable to the improvement achieved by the on-the-fly algorithm compared to the sequential approach.

- **Strong Mutation Testing.** Applying weak mutation testing to the requirement interfaces was a reasonable approach, with regards to the existing test-case generation based on test purposes. It provided a strong coherence between the model-based mutation testing and the general approach. However, the implementation of strong mutation testing should still be one of the top goals for future work, considering the improvement it would mean for the generated test suites. Stopping a test case at the point where it reaches the mutation, without expanding it until the mutation eventually leads to an observable failure is like running in a race, and stopping right in front of the goal. While it might be sufficient to detect most errors of low complexity, it will most probably skip most of the interesting, hard to detect, faults.
- **Comparison to Other Coverage Criteria.** Both for the asynchronous and the synchronous approach, we would be highly interested in performing a study that compares the results of model-based mutation testing with the results achieved by other model-based techniques. We already presented the mutation score we gained for the safing engine by applying signal coverage (covering every value of each signal at least once). However this coverage criterion is rather simple as it does not take into account the possible combinations of several signals. Thus, an analysis of more complex coverage criteria and their relation to model-based mutation testing would be a perfect topic for a future study.
- **Reducing the Number of Mutants.** For both approaches, the equivalent mutants took a high percentage of the processing time. The same observation was already made by Elisabeth Jöbstl in her PhD thesis on model-based mutation testing with constraint and SMT-solvers [105]. This calls for an investigation of the mutation operators, to see whether all of them are needed, or whether some are subsumed by others. Various methods for reducing the number of mutants are summarized in the survey by Jia and Harman [103].

This thesis provided several different contributions to the field of model-based testing of real-time systems. Hopefully some of the ideas will be picked up by other researchers and be built up on, to overcome the current limitations.

Bibliography

[1] Bernhard K. Aichernig. Mutation Testing in the Refinement Calculus. *Formal Aspects of Computing Journal*, 15(2):280–295, 2003. (Cited on page 152.)

[2] Bernhard K. Aichernig. Model-Based Mutation Testing: Theory and Application. Habilitation thesis, Graz University of Technology, Austria, January 2012. (Cited on page 152.)

[3] Bernhard K. Aichernig, Halard Brandl, Elisabeth Jöbstl, and Willibald Krenn. Efficient mutation killers in action. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 120–129, March 2011. (Cited on pages 67 and 152.)

[4] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. Model-based mutation testing of hybrid systems. In Frank S. de Boer, Marcello M. Bonsangue, Stefan Hallerstede, and Michael Leuschel, editors, *Formal Methods for Components and Objects - 8th International Symposium, FMCO 2009, Eindhoven, The Netherlands, November 4-6, 2009. Revised Selected Papers*, volume 6286 of *Lecture Notes in Computer Science*, pages 228–249. Springer, 2009. (Cited on pages 85, 88 and 152.)

[5] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. UML in action: A two-layered interpretation for testing. *SIGSOFT Softw. Eng. Notes*, 36(1):1–8, January 2011. (Cited on pages 9, 63 and 152.)

[6] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, Willibald Krenn, Rupert Schlick, and Stefan Tiran. Killing strategies for model-based mutation testing. *Software Testing, Verification and Reliability*, 25(8):716–748, 2015. (Cited on pages 5, 88, 91, 136 and 152.)

[7] Bernhard K. Aichernig, Harald Brandl, and Willibald Krenn. Qualitative action systems. In *Proceedings of the 11th International Conference on Formal Engineering Methods (ICFEM 2009)*, volume 5885 of *Lecture Notes in Computer Science*, pages 206–225. Springer, 2009. (Cited on page 152.)

[8] Bernhard K. Aichernig and Carlo Corrales Delgado. From faults via test purposes to test cases: on the fault-based testing of concurrent systems. In *Proceedings of FASE'06, Fundamental Approaches to Software Engineering, Vienna, Austria, March 27–29, 2006*, volume 3922 of *Lecture Notes in Computer Science*, pages 324–338. Springer, 2006. (Cited on page 152.)

[9] Bernhard K. Aichernig and Jifeng He. Mutation testing in UTP. *Formal Aspects of Computing*, 21(1-2):33–64, 2009. (Cited on page 152.)

[10] Bernhard K. Aichernig, Klaus Hörmaier, and Florian Lorber. Debugging with timed automata mutations. In Andrea Bondavalli and Felicita Di Giandomenico, editors, *Computer Safety, Reliability, and Security - 33rd International Conference, SAFECOMP 2014, Florence, Italy, September 10-12, 2014. Proceedings*, volume 8666 of *Lecture Notes in Computer Science*, pages 49–64. Springer, 2014. (Cited on pages 10 and 75.)

[11] Bernhard K. Aichernig, Klaus Hörmaier, Florian Lorber, Dejan Nickovic, and Stefan Tiran. Require, test and trace IT. In *Formal Methods for Industrial Critical Systems - 20th International Workshop, FMICS 2015, Oslo, Norway, June 22-23, 2015 Proceedings*, pages 113–127, 2015. (Cited on pages 11, 97, 103 and 131.)

[12] Bernhard K. Aichernig, Klaus Hörmaier, Florian Lorber, Dejan Ničković, Rupert Schlick, Didier Simoneau, and Stefan Tiran. Integration of requirements engineering and test-case generation via OSLC. In *2014 14th International Conference on Quality Software, Allen, TX, USA, October 2-3, 2014*, pages 117–126. IEEE, 2014. (Cited on pages 11, 115 and 131.)

[13] Bernhard K. Aichernig, Elisabeth Jöbstl, and Matthias Kegele. Incremental refinement checking for test case generation. In Margus Veanes and Luca Viganò, editors, *Tests and Proofs: 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings*, pages 1–19, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. (Cited on page 9.)

[14] Bernhard K. Aichernig, Elisabeth Jöbstl, and Stefan Tiran. Model-based mutation testing via symbolic refinement checking. *Sci. Comput. Program.*, 97:383–404, 2015. (Cited on page 85.)

[15] Bernhard K. Aichernig and Florian Lorber. Towards generation of adaptive test cases from partial models of determinized timed automata. In *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*, pages 1–6. IEEE Computer Society, 2015. (Cited on pages 11, 63 and 75.)

[16] Bernhard K. Aichernig and Florian Lorber. On-the-fly determinization of bounded networks of timed automata. In *Tenth International Symposium on Theoretical Aspects of Software Engineering, TASE 2016, 17-19 July 2016, Shanghai, China*. IEEE Computer Society, 2016. In press. (Cited on pages 11, 35 and 63.)

[17] Bernhard K. Aichernig, Florian Lorber, and Dejan Ničković. Time for mutants - model-based mutation testing with timed automata. In Margus Veanes and Luca Viganò, editors, *Tests and Proofs - 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings*, volume 7942 of *Lecture Notes in Computer Science*, pages 20–38. Springer, 2013. (Cited on pages 10 and 25.)

[18] Bernhard K. Aichernig, Florian Lorber, and Martin Tappler. Conformance checking of real-time models - symbolic execution vs. bounded model checking. In Erika Ábrahám, Marcello M. Bonsangue, and Einar Broch Johnsen, editors, *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, volume 9660 of *Lecture Notes in Computer Science*, pages 15–32. Springer, 2016. (Cited on pages 11, 75, 85 and 87.)

[19] Bernhard K. Aichernig, Florian Lorber, and Stefan Tiran. Integrating model-based testing and analysis tools via test case exchange. In Tiziana Margaria, Zongyan Qiu, and Hongli Yang, editors, *Sixth International Symposium on Theoretical Aspects of Software Engineering, TASE 2012, 4-6 July 2012, Beijing, China*, pages 119–126. IEEE Computer Society, 2012. (Cited on page 12.)

[20] Bernhard K. Aichernig, Florian Lorber, and Stefan Tiran. Formal test-driven development with verified test cases. In Luís Ferreira Pires, Slimane Hammoudi, Joaquim Filipe, and Rui César das Neves, editors, *MODELSWARD 2014 - Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development, Lisbon, Portugal, 7 - 9 January, 2014*, pages 626–635. SciTePress, 2014. (Cited on page 12.)

[21] Bernhard K. Aichernig, Dejan Ničković, and Stefan Tiran. Scalable incremental test-case generation from large behavior models. In Jasmin Christian Blanchette and Nikolai Kosmatov, editors, *Tests and Proofs - 9th International Conference, TAP 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 22-24, 2015. Proceedings*, volume 9154 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2015. (Cited on page 106.)

[22] Bernhard K. Aichernig and Percy Antonio Pari Salas. Test case generation by OCL mutation and constraint solving. In *Proceedings of the 5th International Conference on Quality Software (QSIC 2005)*, pages 64–71. IEEE, 2005. (Cited on page 152.)

[23] Bernhard K. Aichernig and Martin Tappler. Symbolic input-output conformance checking for model-based mutation testing. *Electr. Notes Theor. Comput. Sci.*, 320:3–19, 2016. (Cited on pages 88 and 89.)

[24] Bernhard K. Aichernig, Martin Weiglhofer, Bernhard Peischl, and Franz Wotawa. Test purpose generation in an industrial application. In *Proceedings of the 3rd International Workshop on Advances in Model-based Testing*, A-MOST '07, pages 115–125, New York, NY, USA, 2007. ACM. (Cited on page 152.)

[25] Marc Aiguier, Frédéric Boulanger, and Bilal Kanso. A formal abstract framework for modelling and testing complex software systems. *Theor. Comput. Sci.*, 455:66–97, 2012. (Cited on page 159.)

[26] Alex Alberto, Ana Cavalcanti, Marie-Claude Gaudel, and Adenilso Simão. Formal mutation testing for circus. *Information and Software Technology*, pages –, 2016. (Cited on page 151.)

[27] Rajeev Alur. Timed automata. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification: 11th International Conference, CAV'99 Trento, Italy, July 6–10, 1999 Proceedings*, pages 8–22, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. (Cited on page 157.)

[28] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for probabilistic real-time systems. In Javier Leach Albert, Burkhard Monien, and Mario Rodríguez Artalejo, editors, *Automata, Languages and Programming: 18th International Colloquium Madrid, Spain, July 8–12, 1991 Proceedings*, pages 115–126, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg. (Cited on page 156.)

[29] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994. (Cited on pages 6, 19, 22, 29, 35 and 152.)

[30] Rajeev Alur, Limor Fix, and Thomas A Henzinger. A determinizable class of timed automata. In *Computer Aided Verification*, pages 1–13. Springer, 1994. (Cited on pages 153 and 156.)

[31] Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. Parametric real-time reasoning. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 592–601, New York, NY, USA, 1993. ACM. (Cited on page 155.)

[32] Rajeev Alur and Parthasarathy Madhusudan. Decision problems for timed automata: A survey. In *Formal Methods for the Design of Real-Time Systems*, pages 1–24. Springer, 2004. (Cited on page 153.)

[33] Rajeev Alur, Salvatore La Torre, and George J. Pappas. Optimal paths in weighted timed automata. *Theoretical Computer Science*, 318(3):297 – 322, 2004. (Cited on page 156.)

[34] Paul Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In *Proceedings of the 2nd International Conference on Formal Engineering Methods (ICFEM 1998)*, pages 46–54. IEEE, 1998. (Cited on page 151.)

[35] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008. (Cited on page 2.)

[36] Gilles Audemard, Alessandro Cimatti, Artur Kornilowicz, and Roberto Sebastiani. Bounded model checking for timed systems. In *Proceedings of the 22Nd IFIP WG 6.1 International Conference Houston on Formal Techniques for Networked and Distributed Systems*, FORTE '02, pages 243–259, London, UK, UK, 2002. Springer-Verlag. (Cited on page 30.)

[37] Ralph-Johan Back and Reino Kurki-Suonio. Decentralization of process nets with centralized control. In Robert L. Probert, Nancy A. Lynch, and Nicola Santoro, editors, *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983*, pages 131–142. ACM, 1983. (Cited on page 85.)

[38] Bahareh Badban and Martin Lange. Exact incremental analysis of timed automata with an SMT-solver. In Uli Fahrenberg and Stavros Tripakis, editors, *Formal Modeling and Analysis of Timed Systems: 9th International Conference, FORMATS 2011, Aalborg, Denmark, September 21-23, 2011. Proceedings*, pages 177–192, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. (Cited on page 30.)

[39] Christel Baier, Nathalie Bertrand, Patricia Bouyer, and Thomas Brihaye. When are timed automata determinizable? In Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris Nikoletseas, and Wolfgang Thomas, editors, *Automata, Languages and Programming: 36th International Collogquium, ICALP 2009, Rhodes, greece, July 5-12, 2009, Proceedings, Part II*, pages 43–54, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. (Cited on pages 38, 71, 153 and 156.)

[40] Roberto Barbuti and Luca Tesei. Timed automata with urgent transitions. *Acta Informatica*, 40(5):317–347, 2004. (Cited on page 157.)

[41] Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim Larsen, Paul Pettersson, Judi Romijn, and Frits Vaandrager. Minimum-cost reachability for priced time automata. In Maria Domenica Di Benedetto and Alberto Sangiovanni-Vincentelli, editors, *Hybrid Systems: Computation and Control: 4th International Workshop, HSCC 2001 Rome, Italy, March 28–30, 2001 Proceedings*, pages 147–161, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. (Cited on page 156.)

[42] Fevzi Belli, Mutlu Beyazit, Tomohiko Takagi, and Zengo Furukawa. Model-based mutation testing using pushdown automata. *IEICE Transactions*, 95-D(9):2211–2218, 2012. (Cited on page 151.)

[43] Fevzi Belli, Christof J. Budnik, and W. Eric Wong. Basic operations for generating behavioral mutants. *Mutation Analysis, Workshop on*, 0:9, 2006. (Cited on pages 151 and 155.)

[44] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. In Giovanni De Micheli, Rolf Ernst, and Wayne Wolf, editors, *Readings in Hardware/Software Co-design*, pages 147–159. Kluwer Academic Publishers, Norwell, MA, USA, 2002. (Cited on pages 6 and 158.)

[45] Albert Benveniste, Benoît Caillaud, Alberto Ferrari, Leonardo Mangeruca, Roberto Passerone, and Christos Sofronis. Multiple viewpoint contract-based specification and design. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures*, volume 5382 of *Lecture Notes in Computer Science*, pages 200–225. Springer, 2007. (Cited on page 158.)

[46] Albert Benveniste, Benoit Caillaud, Dejan Ničković, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Thomas Henzinger, and

Kim G. Larsen. Contracts for System Design. Rapport de recherche RR-8147, INRIA, November 2012. (Cited on page 95.)

[47] Albert Benveniste, Paul Caspi, Stephen A Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, Jan 2003. (Cited on page 158.)

[48] Albert Benveniste, Paul Caspi, Paul Le Guernic, and Nicolas Halbwachs. Data-flow synchronous languages. In *REX School/Symposium*, volume 803 of *Lecture Notes in Computer Science*, pages 1–45. Springer, 1993. (Cited on page 158.)

[49] Béatrice Bérard and Catherine Dufourd. Timed automata and additive clock constraints. *Information Processing Letters*, 75(1):1 – 7, 2000. (Cited on page 155.)

[50] Béatrice Bérard, Antoine Petit, Volker Diekert, and Paul Gastin. Characterization of the expressive power of silent transitions in timed automata. *Fundam. Inform.*, 36(2-3):145–182, 1998. (Cited on pages 22, 23, 35 and 153.)

[51] Gérard Berry. Scade: Synchronous design and validation of embedded control software. In S. Ramesh and Prahladavaradan Sampath, editors, *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems: Proceedings of the GM R&D Workshop, Bangalore, India, January 2007*, pages 19–33, Dordrecht, 2007. Springer Netherlands. (Cited on page 158.)

[52] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming*, 19(2):87–152, 1992. (Cited on pages 125 and 158.)

[53] Nathalie Bertrand, Thierry Jéron, Amélie Stainer, and Moez Krichen. Off-line test selection with test purposes for non-deterministic timed automata. In ParoshAziz Abdulla and K.RustanM. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, pages 96–111. Springer Berlin Heidelberg, 2011. (Cited on page 153.)

[54] Nathalie Bertrand, Amélie Stainer, Thierry Jéron, and Moez Krichen. A game approach to determinize timed automata. In *Foundations of Software Science and Computational Structures: 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings*, pages 245–259, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. (Cited on page 153.)

[55] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003. (Cited on page 7.)

[56] Sergiy Boroday, Alexandre Petrenko, and Roland Groz. Can a model checker generate tests for non-deterministic systems? *Electronic Notes in Theoretical Computer Science*, 190(2):3–19, 2007. (Cited on page 155.)

[57] Patricia Bouyer and Fabrice Chevalier. On conciseness of extensions of timed automata. *J. Autom. Lang. Comb.*, 10(4):393–405, April 2005. (Cited on page 155.)

[58] Patricia Bouyer, François Laroussinie, and Pierre-Alain Reynier. Diagonal constraints in timed automata: Forward analysis of timed systems. In Paul Pettersson and Wang Yi, editors, *Formal*

Modeling and Analysis of Timed Systems: Third International Conference, FORMATS 2005, Uppsala, Sweden, September 26-28, 2005. Proceedings, pages 112–126, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. (Cited on pages 22, 36, 153 and 158.)

[59] Harald Brandl. *Testing of Hybrid Systems using Qualitative Models.* PhD thesis, Graz University of Technology, 2012. (Cited on page 152.)

[60] Angelo Brillout, Nannan He, Michele Mazzucchi, Daniel Kröning, Mitra Purandare, Philipp Rümmer, and Georg Weissenbacher. Mutation-based test case generation for Simulink models. In *Revised Selected Papers of the 8th International Symposium on Formal Methods for Components and Objects (FMCO 2009)*, volume 6286 of *Lecture Notes in Computer Science*, pages 208–227. Springer, 2010. (Cited on pages 5 and 159.)

[61] Laura Brandán Briones and Ed Brinksma. A test generation framework for quiescent real-time systems. In Jens Grabowski and Brian Nielsen, editors, *Formal Approaches to Software Testing: 4th International Workshop, FATES 2004, Linz, Austria, September 21, 2004, Revised Selected Papers*, pages 64–78, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. (Cited on page 24.)

[62] Timothy A. Budd and Ajei S. Gopal. Program testing by specification mutation. *Comput. Lang.*, 10(1):63–73, January 1985. (Cited on page 151.)

[63] Rachel Cardell-Oliver. Conformance tests for real-time systems with timed automata specifications. *Formal Aspects of Computing*, 12(5):350–371, 2000. (Cited on page 154.)

[64] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 178–188. ACM Press, 1987. (Cited on pages 125 and 158.)

[65] Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. Refinement: An overview. In Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock, editors, *Refinement Techniques in Software Engineering: First Pernambuco Summer School on Software Engineering, PSSE 2004, Recife, Brazil, November 23-December 5, 2004 Revised Lectures*, pages 1–17, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. (Cited on page 100.)

[66] Arindam Chakrabarti, Luca de Alfaro, Thomas A. Henzinger, and Freddy Y. C. Mang. Synchronous and bidirectional component interfaces. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, pages 414–427. Springer, 2002. (Cited on pages 95 and 158.)

[67] Christian Choffrut and Massimiliano Goldwurm. Timed automata with periodic clock constraints. *J. Autom. Lang. Comb.*, 5(4):371–403, October 2000. (Cited on page 155.)

[68] Adenilso da Silva Simão and Alexandre Petrenko. Fault coverage-driven incremental test generation. *Comput. J.*, 53(9):1508–1522, 2010. (Cited on page 151.)

[69] Przemyslaw Daca, Thomas A Henzinger, Willibald Krenn, and Dejan Ničković. Compositional specifications for ioco testing: Technical report. Technical report, IST Austria, 2014. `http://repository.ist.ac.at/152/`. (Cited on page 159.)

[70] Dassault Systèmes. Systemcockpit. Online. `http://www.3ds.com/about-3ds/3dexperience-platform`, Visited: 2014-03-06. (Cited on page 116.)

[71] Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Timed I/O automata. In *Proceedings of the 13th ACM international conference on Hybrid systems: computation and control - HSCC '10*, page 91, New York, New York, USA, April 2010. ACM Press. (Cited on pages 47 and 48.)

[72] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool kronos. In Rajeev Alur, ThomasA. Henzinger, and EduardoD. Sontag, editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 208–219. Springer Berlin Heidelberg, 1996. (Cited on page 158.)

[73] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. (Cited on pages 33, 58, 105 and 111.)

[74] M. De Wulf, L. Doyen, T. A. Henzinger, and J. F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings*, pages 17–30, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. (Cited on page 153.)

[75] Volker Diekert, Paul Gastin, and Antoine Petit. Removing epsilon-transitions in timed automata. In *Proceedings of the 14th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '97, pages 583–594, London, UK, UK, 1997. Springer-Verlag. (Cited on page 71.)

[76] Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972. (Cited on page 1.)

[77] Cătălin Dima and Ruggero Lanotte. Removing all silent transitions from timed automata. In Joël Ouaknine and Frits W. Vaandrager, editors, *Formal Modeling and Analysis of Timed Systems: 7th International Conference, FORMATS 2009, Budapest, Hungary, September 14-16, 2009. Proceedings*, pages 118–132, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. (Cited on page 155.)

[78] Laurent Doyen, Thomas A. Henzinger, Barbara Jobstmann, and Tatjana Petrov. Interface theories with component reuse. In Luca de Alfaro and Jens Palsberg, editors, *Proceedings of the 8th ACM & IEEE International conference on Embedded software, EMSOFT 2008, Atlanta, GA, USA, October 19-24, 2008*, pages 79–88. ACM, 2008. (Cited on pages 95 and 158.)

[79] Lydie. du Bousquet, Farid Ouabdesselam, Jean Luc Richier, and Nicolas Zuanon. Lutess: A specification-driven testing environment for synchronous software. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 267–276, New York, NY, USA, 1999. ACM. (Cited on page 159.)

[80] Khaled El-Fakih, Rita Dorofeeva, Nina Yevtushenko, and Gregor von Bochmann. FSM-based testing from user defined faults adapted to incremental and mutation testing. *Programming and Computer Software*, 38(4):201–209, 2012. (Cited on page 151.)

[81] Khaled El-Fakih, Nina Yevtushenko, and Hacène Fouchal. Testing timed finite state machines with guaranteed fault coverage. In Manuel Núñez, Paul Baker, and Mercedes G. Merayo, editors, *Testing of Software and Communication Systems, 21st IFIP WG 6.1 International Conference, TESTCOM 2009 and 9th International Workshop, FATES 2009, Eindhoven, The Netherlands, November 2-4, 2009. Proceedings*, volume 5826 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2009. (Cited on page 151.)

[82] Christian Ellen, Sven Sieverding, and Hardi Hungar. Detecting consistencies and inconsistencies of pattern-based functional requirements. In Frédéric Lang and Francesco Flammini, editors, *Formal Methods for Industrial Critical Systems - 19th International Conference, FMICS 2014, Florence, Italy, September 11-12, 2014. Proceedings*, volume 8718 of *Lecture Notes in Computer Science*, pages 155–169. Springer, 2014. (Cited on page 159.)

[83] Sandra C. Fabbri, Márcio Eduardo Delamaro, José Carlos Maldonado, and Paulo Cesar Masiero. Mutation analysis testing for finite state machines. In *Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on*, pages 220–229, 1994. (Cited on page 155.)

[84] Stefan Farfeleder, Thomas Moser, Andreas Krall, Tor Stålhane, Herbert Zojer, and Christian Panis. DODT: Increasing requirements formalism using domain ontologies for improved embedded systems development. In *Proceedings of the 14th IEEE International Symposium on Design and Diagnostics of Electronic Circuits & Systems, DDECS 2011, Cottbus, Germany, April 13-15, 2011*, pages 271–274, April 2011. (Cited on page 116.)

[85] Colin J. Fidge and Andy J. Wellings. An action-based formal model for concurrent real-time systems. *Formal Aspects of Computing*, 9(2):175–207, 1997. (Cited on page 86.)

[86] Olivier Finkel. Undecidable problems about timed automata. In *Proceedings of the 4th International Conference on Formal Modeling and Analysis of Timed Systems*, FORMATS'06, pages 187–199, Berlin, Heidelberg, 2006. Springer-Verlag. (Cited on pages 35 and 152.)

[87] Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. A symbolic framework for model-based testing. In *Proceedings of the First Combined International Conference on Formal Approaches to Software Testing and Runtime Verification*, FATES'06/RV'06, pages 40–54, Berlin, Heidelberg, 2006. Springer-Verlag. (Cited on page 87.)

[88] Gordon Fraser and Franz Wotawa. Using model-checkers for mutation-based test-case generation, coverage analysis and specification analysis. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA 2006)*, pages 16–22. IEEE, 2006. (Cited on page 151.)

[89] Thierry Gautier and Paul Le Guernic. SIGNAL: A declarative language for synchronous programming of real-time systems. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture, Portland, Oregon, USA, September 14-16, 1987, Proceedings*, volume 274 of *Lecture Notes in Computer Science*, pages 257–277. Springer, 1987. (Cited on page 158.)

[90] Grégoire Hamon, Leonardo De Moura, and John Rushby. Automated test generation with SAL. *CSL Technical Note*, 2005. (Cited on page 159.)

[91] Nannan He, Philipp Rümmer, and Daniel Kröning. Test-case generation for embedded simulink via formal concept analysis. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 224–229, New York, NY, USA, 2011. ACM. (Cited on page 159.)

[92] Christopher Henard, Mike Papadakis, and Yves Le Traon. Mutation-based generation of software product line test configurations. In Claire Le Goues and Shin Yoo, editors, *Search-Based Software Engineering - 6th International Symposium, SSBSE 2014, Fortaleza, Brazil, August 26-29, 2014. Proceedings*, volume 8636 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2014. (Cited on page 151.)

[93] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, LICS '96, pages 278–, Washington, DC, USA, 1996. IEEE Computer Society. (Cited on page 3.)

[94] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? In *Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing*, STOC '95, pages 373–382, New York, NY, USA, 1995. ACM. (Cited on page 156.)

[95] Thomas A. Henzinger and Dejan Ničković. Independent implementability of viewpoints. In *Monterey Workshop*, volume 7539 of *Lecture Notes in Computer Science*, pages 380–395. Springer, 2012. (Cited on page 158.)

[96] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. You assume, we guarantee: Methodology and case studies. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification: 10th International Conference, CAV'98 Vancouver, BC, Canada, June 28 – July 2, 1998 Proceedings*, pages 440–451, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. (Cited on page 100.)

[97] Anders Hessel and Paul Pettersson. Cover-a test-case generation tool for timed systems. *Testing of Software and Communicating Systems*, pages 31–34, 2007. (Cited on page 154.)

[98] Rob .M. Hierons and Mercedes G. Merayo. Mutation testing from probabilistic finite state machines. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*, pages 141–150, Sept 2007. (Cited on page 151.)

[99] Robert M. Hierons. Applying adaptive test cases to nondeterministic implementations. *Inf. Process. Lett.*, 98(2):56–60, 2006. (Cited on page 84.)

[100] Franz Huber, Bernhard Schätz, Alexander Schmidt, and Katharina Spies. AutoFocus: A tool for distributed systems specification. In *Proceedings of the 4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT 1996)*, volume 1135 of *Lecture Notes in Computer Science*, pages 467–470. Springer, 1996. (Cited on page 159.)

[101] IBM. Rational doors. Online. `http://www-03.ibm.com/software/products/en/ratidoorfami`, Visited: 2014-03-06. (Cited on page 117.)

[102] Radu Iosif, Adam Rogalewicz, and Tomás Vojnar. Abstraction refinement for trace inclusion of data automata. *CoRR*, abs/1410.5056, 2014. (Cited on page 153.)

[103] Yue Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649 –678, 2011. (Cited on pages 5, 34, 112, 137, 152 and 168.)

[104] Elisabeth Jöbstl. Automating test case generation from transition systems via symbolic execution and SAT solving. Master's thesis, Graz University of Technology, 2009. (Cited on page 9.)

[105] Elisabeth Jöbstl. *Model-Based Mutation Testing with Constraint and SMT Solvers*. PhD thesis, Graz University of Technology, 2014. (Cited on pages xv, 3, 4, 5, 9, 152 and 168.)

[106] Ulrich Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In *Proceedings of the 19th National Conference on Artificial Intelligence*, AAAI'04, pages 167–172. AAAI Press, 2004. (Cited on page 108.)

[107] René Just, Franz Schweiggert, and Gregory M. Kapfhammer. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 612–615, November 9–11 2011. (Cited on page 137.)

[108] Ahmed Khoumsi, Thierry Jéron, and Hervé Marchand. Test cases generation for nondeterministic real-time systems. In Alexandre Petrenko and Andreas Ulrich, editors, *Formal Approaches to Software Testing: Third International Workshop on Formal Approaches to Testing of Software, FATES 2003, Montreal, Quebec, Canada, October 6th, 2003. Revised Papers*, pages 131–146, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. (Cited on page 24.)

[109] JinHyun Kim, KimG. Larsen, Brian Nielsen, Marius Mikucionis, and Petur Olsen. Formal analysis and testing of real-time automotive systems using uppaal tools. In Manuel Núñez and Matthias Güdemann, editors, *Formal Methods for Industrial Critical Systems*, volume 9128 of *Lecture Notes in Computer Science*, pages 47–61. Springer International Publishing, 2015. (Cited on page 154.)

[110] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Scala to the power of Z3: Integrating SMT and programming. In *Proceedings of the 23rd International Conference on Automated Deduction*, CADE'11, pages 400–406, Berlin, Heidelberg, 2011. Springer-Verlag. (Cited on page 33.)

[111] I. Koufareva, Alexandre Petrenko, and Nina Yevtushenko. Test generation driven by user-defined fault models. In Gyula Csopaki, Sarolta Dibuz, and Katalin Tarnay, editors, *Testing of Communicating Systems: Method and Applications, IFIP TC6 12^{th} International Workshop on Testing Communicating Systems, September 1-3, 1999, Budapest, Hungary*, volume 147 of *IFIP Conference Proceedings*, pages 215–236. Kluwer, 1999. (Cited on page 151.)

[112] Willibald Krenn and Bernhard K. Aichernig. Test case generation by contract mutation in Spec#. In *Proceedings of the 5th Workshop on Model-Based Testing (MBT 2009)*, volume 253(2) of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 71–86. Elsevier, 2009. (Cited on page 152.)

[113] Willibald Krenn, Dejan Ničković, and Loredana Tec. Incremental language inclusion checking for networks of timed automata. In Víctor A. Braberman and Laurent Fribourg, editors, *Formal Modeling and Analysis of Timed Systems - 11th International Conference, FORMATS 2013, Buenos Aires, Argentina, August 29-31, 2013. Proceedings*, volume 8053 of *Lecture Notes in Computer Science*, pages 152–167. Springer, 2013. (Cited on pages 33, 58 and 155.)

[114] Willibald Krenn, Rupert Schlick, and Bernhard K. Aichernig. Mapping UML to labeled transition systems for test-case generation - a translation via object-oriented action systems. In *Revised Selected Papers of the 8th International Symposium on Formal Methods for Components and Objects (FMCO 2009)*, volume 6286 of *Lecture Notes in Computer Science*, pages 186–207. Springer, 2010. (Cited on page 9.)

[115] Willibald Krenn, Rupert Schlick, Stefan Tiran, Bernhard K. Aichernig, Elisabeth Jöbstl, and Harald Brandl. Momut::UML model-based mutation testing for UML. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–8, April 2015. (Cited on page 152.)

[116] Moez Krichen and Stavros Tripakis. Interesting properties of the real-time conformance relation. In *Proceedings of the 3rd International Colloquium on Theoretical Aspects of Computing (ICTAC 2006)*, volume 4281 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2006. (Cited on page 88.)

[117] Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, 2009. (Cited on pages 23, 24, 27, 28, 29, 48 and 154.)

[118] Reino Kurki-Suonio. Action systems in incremental and aspect-oriented modeling. *Distributed Computing*, 16(2-3):201–217, 2003. (Cited on page 86.)

[119] Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. Online testing of real-time systems using uppaal. In Jens Grabowski and Brian Nielsen, editors, *Formal Approaches to Software Testing: 4th International Workshop, FATES 2004, Linz, Austria, September 21, 2004, Revised Selected Papers*, pages 79–94, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. (Cited on page 24.)

[120] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *STTT*, 1(1-2):134–152, 1997. (Cited on pages 156 and 158.)

[121] Florian Lorber. Model-based mutation testing of synchronous and asynchronous real-time systems. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, pages 1–2. IEEE, 2015. (Cited on page 12.)

[122] Florian Lorber. Test-case generation via language inclusion for non-deterministic networks of timed automata. In Bernhard K. Aichernig and Alessandro Rossini, editors, *Proceedings of the Doctoral Symposium of Formal Methods 2015*, pages 15–18, 2015. (Cited on page 12.)

[123] Florian Lorber, Amnon Rosenmann, Dejan Ničković, and Bernhard K. Aichernig. Bounded determinization of timed automata with silent transitions. In Sriram Sankaranarayanan and Enrico Vicario, editors, *Formal Modeling and Analysis of Timed Systems - 13th International Conference, FORMATS 2015, Madrid, Spain, September 2-4, 2015, Proceedings*, volume 9268 of *Lecture Notes in Computer Science*, pages 288–304. Springer, 2015. (Cited on pages 11, 35 and 63.)

[124] Oded Maler, Amir Pnueli, and Joseph Sifakis. On the synthesis of discrete controllers for timed systems. In Ernst W. Mayr and Claude Puech, editors, *STACS 95: 12th Annual Symposium on Theoretical Aspects of Computer Science Munich, Germany, March 2–4, 1995 Proceedings*, pages 229–242, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. (Cited on page 157.)

[125] Raluca Marinescu, Mehrdad Saadatmand, Alessio Bucaioni, Cristina Seceleanu, and Paul Pettersson. A model-based testing framework for automotive embedded systems. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 38–47, Aug 2014. (Cited on page 154.)

[126] B. Marre and A. Arnould. Test sequences generation from LUSTRE descriptions: GATEL. In *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*, pages 229–237, 2000. (Cited on page 159.)

[127] W. Mayer and M. Stumptner. Evaluating models for model-based debugging. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 128–137, Washington, DC, USA, 2008. IEEE Computer Society. (Cited on page 75.)

[128] Wolfgang Mayer and Markus Stumptner. Model-based debugging – state of the art and future challenges. *Electron. Notes Theor. Comput. Sci.*, 174(4):61–82, May 2007. (Cited on page 75.)

[129] Jennifer McManis and Pravin Varaiya. Suspension automata: A decidable class of hybrid automata. In *Proceedings of the 6th International Conference on Computer Aided Verification*, CAV '94, pages 105–117, London, UK, UK, 1994. Springer-Verlag. (Cited on pages 155 and 156.)

[130] Sun Meng, Farhad Arbab, Bernhard K Aichernig, Lăcrămioara Aştefănoaei, Frank S de Boer, and Jan Rutten. Connectors as designs: Modeling, refinement and test case generation. *Science of Computer Programming*, 77(7):799–822, 2012. (Cited on page 152.)

[131] Marius Mikucionis, Brian Nielsen, and Kim G. Larsen. Real-time system testing on-the-fly. In Kaisa Sere and Marina Waldén, editors, *the 15th Nordic Workshop on Programming Theory*, number 34 in B, pages 36–38, Turku, Finland, October 29–31 2003. Abo Akademi, Department of Computer Science, Finland. Abstracts. (Cited on page 154.)

[132] Mihai Nica, Simona Nica, and Franz Wotawa. Does testing help to reduce the number of potentially faulty statements in debugging? In *Proceedings of the 5th International Academic and Industrial Conference on Testing - Practice and Research Techniques*, TAIC PART'10, pages 88–103, Berlin, Heidelberg, 2010. Springer-Verlag. (Cited on page 76.)

[133] Peter Niebert, Moez Mahfoudh, Eugene Asarin, Marius Bozga, Oded Maler, and Navendu Jain. Verification of timed automata via satisfiability checking. In Werner Damm and Ernst Rüdiger Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems: 7th International Symposium, FTRTFT 2002 Co-sponsored by IFIP WG 2.2 Oldenburg, Germany, September 9–12, 2002 Proceedings*, pages 225–243, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. (Cited on page 30.)

[134] Brian Nielsen and Arne Skou. Automated test generation from timed automata. *STTT*, 5(1):59–77, 2003. (Cited on page 154.)

[135] Robert Nilsson. Automated selective test case generation methods for real-time systems. Master's thesis, University of Skövde, Department of Computer Science, 2000. (Cited on page 155.)

[136] Robert Nilsson, Jeff Offutt, and Sten F. Andler. Mutation-based testing criteria for timeliness. In *28th International Computer Software and Applications Conference (COMPSAC 2004), Design and Assessment of Trustworthy Software-Based Systems, 27-30 September 2004, Hong Kong, China, Proceedings*, pages 306–311. IEEE Computer Society, 2004. (Cited on pages 5 and 155.)

[137] Christer Norstrom, Anders Wall, and Wang Yi. Timed automata as task models for event-driven systems. In *Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference on*, pages 182–189, 1999. (Cited on page 156.)

[138] Chee-Mun Ong. *Dynamic simulation of electric machinery: using MATLAB/SIMULINK*, volume 5. Prentice Hall PTR Upper Saddle River, NJ, 1998. (Cited on page 116.)

[139] Open Services for Lifecycle Collaboration. What is OSLC? - OSLC primer. Online. `http://open-services.net/resources/tutorials/oslc-primer/what-is-oslc/`, Visited on: 2014-03-06. (Cited on page 116.)

[140] Mike Papadakis and Yves Le Traon. Using mutants to locate "unknown" faults. In Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche, editors, *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, pages 691–700. IEEE Computer Society, 2012. (Cited on page 75.)

[141] Virginia Papailiopoulou. Automatic test generation for LUSTRE/SCADE programs. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 517–520, Washington, DC, USA, 2008. IEEE Computer Society. (Cited on page 158.)

[142] Alexandre Petrenko, Omer Nguena-Timo, and S. Ramesh. Multiple mutation testing from FSM. In Elvira Albert and Ivan Lanese, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9688 of *Lecture Notes in Computer Science*, pages 222–238. Springer, 2016. (Cited on page 151.)

[143] Sandra C. Pinto Ferraz Fabbri, José Carlos Maldonado, Tatiana Sugeta, and Paulo Cesar Masiero. Mutation testing applied to validate specifications based on statecharts. In *Software Reliability Engineering*, pages 210–219, 1999. (Cited on page 155.)

[144] Ajitha Rajan and Thomas Wahl. *CESAR - Cost-efficient Methods and Processes for Safety-relevant Embedded Systems*. Springer Vienna, 2013. (Cited on page 117.)

[145] Pascal Raymond, Xavier Nicollin, Nicolas Halbwachs, and Daniel Weber. Automatic testing of reactive systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS '98, pages 200–, Washington, DC, USA, 1998. IEEE Computer Society. (Cited on pages 158 and 159.)

[146] Jan Reineke and Stavros Tripakis. Basic problems in multi-view modeling. Technical Report UCB/EECS-2014-4, EECS Department, University of California, Berkeley, Jan 2014. (Cited on page 158.)

[147] Raymond Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, April 1987. (Cited on page 75.)

[148] Augusto Sampaio, Sidney Nogueira, and Alexandre Mota. Compositional verification of input-output conformance via CSP refinement checking. In Karin Breitman and Ana Cavalcanti, editors, *Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings*, Lecture Notes in Computer Science, pages 20–48. Springer, 2009. (Cited on pages 151 and 159.)

[149] Rupert Schlick, Wolfgang Herzner, and Elisabeth Jöbstl. Fault-based generation of test cases from UML-models – approach and some experiences. In Francesco Flammini, Sandro Bologna, and Valeria Vittorini, editors, *Computer Safety, Reliability, and Security: 30th International Conference,SAFECOMP 2011, Naples, Italy, September 19-22, 2011. Proceedings*, pages 270–283, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. (Cited on page 63.)

[150] Julien Schmaltz and Jan Tretmans. On conformance testing for timed systems. In *Proceedings of the 6th International Conference on Formal Modeling and Analysis of Timed Systems*, FORMATS '08, pages 250–264, Berlin, Heidelberg, 2008. Springer-Verlag. (Cited on page 23.)

[151] Cristina Cerschi Seceleanu and Tiberiu Seceleanu. Synchronization can improve reactive systems control and modularity. *J. UCS*, 10(10):1429–1468, 2004. (Cited on pages 86 and 159.)

[152] Cristina Cerschi Seceleanu, Aneta Vulgarakis, and Paul Pettersson. REMES: A resource model for embedded systems. In *14th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2009, Potsdam, Germany, 2-4 June 2009*, pages 84–94. IEEE Computer Society, 2009. (Cited on page 154.)

[153] J. Michael Spivey. *Z Notation - a reference manual (2. ed.)*. Prentice Hall International Series in Computer Science. Prentice Hall, 1992. (Cited on page 159.)

[154] Jan Springintveld, Frits Vaandrager, and Pedro R D'Argenio. Testing timed automata. *Theoretical computer science*, 254(1):225–257, 2001. (Cited on pages 24 and 154.)

[155] Philip Alan Stocks. *Applying formal methods to software testing*. PhD thesis, Department of computer science, University of Queensland, 1993. (Cited on page 151.)

[156] P. Vijay Suman and Paritosh K. Pandya. Determinization and expressiveness of integer reset timed automata with silent transitions. In Adrian Horia Dediu, Armand Mihai Ionescu, and Carlos Martín-Vide, editors, *Language and Automata Theory and Applications: Third International Conference, LATA 2009, Tarragona, Spain, April 2-8, 2009. Proceedings*, pages 728–739, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. (Cited on pages 153 and 156.)

[157] P. Vijay Suman, Paritosh K. Pandya, Shankara Narayanan Krishna, and Lakshmi Manasa. Timed automata with integer resets: Language inclusion and expressiveness. In *Proceedings of the 6th International Conference on Formal Modeling and Analysis of Timed Systems*, FORMATS '08, pages 78–92, Berlin, Heidelberg, 2008. Springer-Verlag. (Cited on page 156.)

[158] P. Teehan, M. Greenstreet, and G. Lemieux. A survey and taxonomy of GALS design styles. *IEEE Design Test of Computers*, 24(5):418–428, Sept 2007. (Cited on pages 146 and 158.)

[159] Stefan Tiran. On the Effects of UML Modeling Styles in Model-based Mutation Testing. Master thesis, Graz, University of Technology, 2013. (Cited on page 33.)

[160] Jan Tretmans. Test generation with inputs, outputs, and quiescence. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems: Second International Workshop, TACAS '96 Passau, Germany, March 27–29, 1996 Proceedings*, pages 127–146, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. (Cited on page 23.)

[161] Jan Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing*, pages 1–38, 2008. (Cited on pages 27 and 28.)

[162] Stavros Tripakis. Folk theorems on the determinization and minimization of timed automata. *Inf. Process. Lett.*, 99(6):222–226, 2006. (Cited on pages 35, 152 and 159.)

[163] Ashutosh Trivedi and Dominik Wojtczak. Recursive timed automata. In *Proceedings of the 8th International Conference on Automated Technology for Verification and Analysis*, ATVA'10, pages 306–324, Berlin, Heidelberg, 2010. Springer-Verlag. (Cited on page 156.)

[164] Mark Utting and Bruno Legeard. *Practical Model-Based Testing – A Tools Approach*. Morgan Kaufmann Publishers, 2007. (Cited on page 2.)

[165] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability (STVR)*, 22(5):297–312, 2012. (Cited on pages xv, 2, 3 and 4.)

[166] Machiel van der Bijl, Arend Rensink, and Jan Tretmans. Compositional testing with ioco. In Alexandre Petrenko and Andreas Ulrich, editors, *Formal Approaches to Software Testing: Third International Workshop on Formal Approaches to Testing of Software, FATES 2003, Montreal, Quebec, Canada, October 6th, 2003. Revised Papers*, pages 86–100, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. (Cited on page 159.)

[167] Parham Vasaiely, Andreas Keis, and Rainer Ersch. The CESAR and MBAT interoperability specification. Technical report, Technical report, EADS UK Ltd, 2012. (Cited on page 115.)

[168] Sabrina von Styp, Henrik Bohnenkamp, and Julien Schmaltz. A conformance testing relation for symbolic timed automata. In Krishnendu Chatterjee and Thomas A. Henzinger, editors, *Formal Modeling and Analysis of Timed Systems: 8th International Conference, FORMATS 2010, Klosterneuburg, Austria, September 8-10, 2010. Proceedings*, pages 243–255, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. (Cited on pages 87 and 88.)

[169] Aneta Vulgarakis, Jagadish Suryadevara, Jan Carlson, Cristina Cerschi Seceleanu, and Paul Pettersson. Formal semantics of the procom real-time component model. In *35th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2009, Patras, Greece, August 27-29, 2009, Proceedings*, pages 478–485. IEEE Computer Society, 2009. (Cited on page 154.)

[170] Axel Wabenhorst. A model of real-time distributed systems. In *Proceedings of the IFIP TC2/WG2.2,2.3 International Conference on Programming Concepts and Methods*, PROCOMET '98, pages 462–482, London, UK, UK, 1998. Chapman & Hall, Ltd. (Cited on page 86.)

[171] Md Tawhid Bin Waez, Juergen Dingel, and Karen Rudie. A survey of timed automata for the development of real-time systems. *Computer Science Review*, 9:1–26, August 2013. (Cited on pages 19, 152 and 155.)

[172] Asmita Wakankar, Anup K. Bhattacharjee, S. D. Dhodapkar, Paritosh K. Pandya, and Kavi Arya. Automatic test case generation in model based software design to achieve higher reliability. In *Reliability, Safety and Hazard (ICRESH), 2010 2nd International Conference on*, pages 493–499, Dec 2010. (Cited on page 158.)

[173] Ting Wang, Jun Sun, Yang Liu, Xinyu Wang, and Shanping Li. Are timed automata bad for a specification language? language inclusion checking for timed automata. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, pages 310–325, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. (Cited on pages 44, 71 and 153.)

[174] Martin Weiglhofer and Franz Wotawa. ”On the fly” input output conformance verification. In *Proceedings of the IASTED International Conference on Software Engineering*, pages 286–291. ACTA Press, 2008. (Cited on page 151.)

[175] Tomi Westerlund and Juha Plosila. Formal timing model for hardware components. In *Norchip Conference, 2004. Proceedings*, pages 293–296, Nov 2004. (Cited on page 86.)

[176] Franz Wotawa. On the relationship between model-based debugging and program slicing. *Artif. Intell.*, 135(1-2):125–143, February 2002. (Cited on page 76.)