Complete Model-Based Testing Applied to the Railway Domain

Dissertation

zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften
– Dr.-Ing. –

vorgelegt von

Felix Hübner

im Oktober 2017

Fachbereich 3 (Mathematik / Informatik)

Complete Model-Based Testing Applied to the Railway Domain

Dissertation

zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften
– Dr.-Ing. –

vorgelegt von

Felix Hübner

im Fachbereich 3 (Mathematik / Informatik)
der Universität Bremen
im Oktober 2017

Datum des Kolloquiums: 10.Januar 2018

Gutachter: Prof. Dr. Jan Peleska (Universität Bremen)
Gutachter: Prof. Mohammad Reza Mousavi, Ph.D. (University of Leicester)

Summary

Testing is the most important verification technique to assert the correctness of an embedded system. Model-based testing (MBT) is a popular approach that generates test cases from models automatically. For the verification of safety-critical systems, complete MBT strategies are most promising. Complete testing strategies can guarantee that all errors of a certain kind are revealed by the generated test suite, given that the system-under-test fulfils several hypotheses. This work presents a complete testing strategy which is based on equivalence class abstraction. Using this approach, reactive systems, with a potentially infinite input domain but finitely many internal states, can be abstracted to finite-state machines. This allows for the generation of finite test suites providing completeness. However, for a system-under-test, it is hard to prove the validity of the hypotheses which justify the completeness of the applied testing strategy. Therefore, we experimentally evaluate the fault-detection capabilities of our equivalence class testing strategy in this work. We use a novel mutation-analysis strategy which introduces artificial errors to a SystemC model to mimic typical HW/SW integration errors. We provide experimental results that show the adequacy of our approach considering case studies from the railway domain (i.e., a speed-monitoring function and an interlocking-system controller) and from the automotive domain (i.e., an airbag controller). Furthermore, we present extensions to the equivalence class testing strategy. We show that a combination with randomisation and boundary-value selection is able to significantly increase the probability to detect HW/SW integration errors.

Zusammenfassung (auf Deutsch)

Testen ist die wichtigste Verifikationstechnik, um die Korrektheit eines eingebetteten Systems zu überprüfen. Modellbasiertes Testen (MBT) kommt seit mehreren Jahren eine große Bedeutung zu. Hierbei werden Testfälle automatisch aus Modellen abgeleitet. Besonders vielversprechend für den Test von sicherheitskritischen Systemen sind komplette Testverfahren. Komplette Testverfahren garantieren alle Fehler einer bestimmten Fehlerklasse aufzudecken, sofern das zu testende System einige Eigenschaften erfüllt. In dieser Arbeit setzen wir ein solches Verfahren um. Dieser Ansatz basiert auf der Äquivalenzklassenpartitionierung von Reaktiven Systemen mit einem potentiell unendlich großen Eingabebereich und einer endlichen Menge von internen Zuständen. Der daraus resultierende endliche Automat ermöglicht das Erzeugen einer endlichen Menge an Testfällen, welche Komplettheitseigenschaften aufweist. Da sich die Hypothesen von kompletten Testverfahren für das zu testende System in der Regel nicht beweisen lassen, ist es Ziel dieser Arbeit, experimentell nachzuweisen, dass das entwickelte Testverfahren eine hohe Fehleraufdeckungsrate besitzt. Hierbei verwenden wir ein neuartiges Mutationsanalyse Verfahren, welches künstliche Fehler in ein SystemC-Modell einfügt und somit typische HW/SW Integrationsfehler nachempfindet. Eine akzeptable Fehleraufdeckungsrate konnte für Fallbeispiele aus dem Bahnbereich (ein System zur Zuggeschwindigkeitsüberwachung und eine Stellwerkssteuerung) sowie ein weiteres Fallbeispiel aus dem Automobilbereich (ein Airbag-Steuergerät) nachgewiesen werden. Außerdem zeigen wir, dass sich die Fehleraufdeckungsrate durch die Kombination des Äquivalenzklassenverfahrens mit Randomisierung und Grenzwerttests noch deutlich steigern lässt.

Acknowledgements

At the moment of this writing, I am missing words to describe all the emotions that I am currently experiencing. Though this text appears at the very beginning of this work, I am writing these words at the end of a very long journey. Looking back makes me feel grateful, proud, relieved, happy, and—at the very same time—sad that a journey has now come to an end. It has been three years and five months since I started this work. Right now, I do not remember the first doubts, the first days of stress facing the large challenge of handling all of the time-consuming duties related to teaching on the one hand and a supervisor pushing me to my first publication on the other hand. Instead, I remember the great excitement of the first submission, the first acceptance, the great excitement and gladness after my first teaching experience—especially my first lecture in front of hundreds of students. I remember the great days at my office that I owe to my nice colleagues and my supervisor. I remember the invaluable pleasure of working on a topic that is complex, promising, challenging and offers significant technological advance. It has been exciting to work at the leading edge of technology in the domain of model-based testing. Starting as a doctoral student, I always feared the painful path that lay in front of me. That this path was never painful but instead always pleasurable is mainly due to many people who accompanied me. Therefore, I would like to take the opportunity to say thank you.

First and foremost I want to express my great gratitude to Jan Peleska: Thank you for your invaluable support as my supervisor. At the beginning of my doctoral studies, you pushed, motivated and always praised me at the right moments. You gave me guidance in the right directions and you provided me with the possibility to work on an overly interesting topic. Especially at the end, your precious remarks on this dissertation have greatly influenced this work. Even before I started my doctoral position at your department, you had greatly influenced my academic career. From my time as a young bachelor student on, I was impressed by your ability to make desperately complicated topics enjoyable. This made me attend almost every one of your lectures. I learned a lot, mainly because of your great manner of teaching, which I always admired. Thank you as well for all the motivating words of praise that you generously gave to me. This among other things has made our joint work on publications and teaching a pleasure for me.

Furthermore, I want to thank Mohammad Reza Mousavi for being the second reviewer of this work. I hope that you will enjoy reading this work as I enjoyed writing it. I have known you as a prominent expert of model-based testing since my stay in Halmstad for the Summer School of Testing in 2013. I hope that some of the passion for formal testing methods by which I was inspired in Halmstad can be sensed upon reading this work.

Another thank-you goes to Siemens for the funding of my dissertation. In particular, I want to thank Ralf Pinger and Jens Braband. I very much enjoyed my stays in Braunschweig, where you prepared exciting and well-organised meetings of the Siemens International Rail Automation Graduate School (iRAGS).

As mentioned, every-day life at university was a pleasure. This is due to all the nice colleagues I had the opportunity to work with. Thank you Christoph, Uwe, Wen-ling, Florian, Cécile, Blagoy

and Birgit for the friendly atmosphere at the working group AGBS, for good discussions during lunches and coffee breaks. Christoph, Uwe and Florian were always on the spot for nice talks for distraction, and they also helped med with technical problems whenever I needed assistance. The same is true for Frank, Fabian, Laurens and Niklas who, although not officially part of the working group AGBS, were always on the spot. Special thanks to Niklas for proofreading and helpful remarks.

I am very grateful to the University of Bremen. I started my Bachelor's studies in 2008. Since then, the University of Bremen has become the centre of my everyday life. I enjoyed my time and love the atmosphere of this charming University, which I will always promote with pride as an excellent place to learn and to do research in a great environment. Of course, the University of Bremen as an institution is a union of many people. I would like to personally thank some of them. Thank you to Görschwin Fey, Robert Wille, Martin Gogolla, Daniel Grosse and Ulrich Kühne for offering support as supervisors in the context of the Graduate school of System Design (SyDe). Thank you to all fellow students of SyDe for the SyDe meetings with interesting talks and discussions. I also wish to thank Karsten Hölscher. With your excellence in teaching and your kind, cooperative and humorous nature, you made the otherwise demanding struggles of teaching much less painful. In the end, teaching became an experience that I will remember gladly.

Endless gratitude is devoted to my family. Probably the decision to start doctoral studies often requires some bravery, self-confidence and optimism. The fact that I am in a position to have these prerequisites at hand is possible only due to your unquestionable support. The fact that you always stand by my side without my need to ask or thank you for is invaluable. I know that I can always rely on you, and this made it possible for me to never feel serious doubt. You support me on the hardest journeys and would evidently run the last half of a marathon with me to get me to the finish line.

Contents

1 Introduction

1.1 Model-Based Testing Applied to the Railway Domain

Nowadays, embedded systems are becoming more and more complex. At the same time, these systems are used in more contexts and more domains and take over more and more functionality—including safety-related functionality. Though safety once relied on human supervision, it is now replaced by supervision and intervention by embedded systems. Autonomous driving, autonomous flight and train-protection systems are examples of safety-critical functionality that allows for a higher degree of safety. However, the safety functionality provided by an embedded system assumes correct functionality and imposes, in most cases, high reliability and availability constraints on the system. The higher the level of reliability and availability constraints (i.e., the higher the safety criticality), the higher the need for formal methods for the verification and validation of such systems. Among the verification methods to be considered are formal proofs and model checking and testing. While formal proofs and model checking are in most cases applied to formal models, testing can be applied to the final system. In the case of embedded systems, testing can be applied to the final integrated HW/SW system, but also to subsystems running on the final target hardware, or in simulators, HW emulators or combinations of these possibilities. The flexibility of testing—and the fact that testing in most cases is the only way to assess the functional correctness of a system—probably makes it the most important verification technique in practice. No train, no driving assistant, no autopilot, no satellite-control software will get certification credit without being tested.

However, testing is usually incomplete. While formal proofs and model checking allow for absolute statements about the validity of certain properties, it is not usually feasible to test all possible combinations of inputs. Considering the reactive nature of embedded systems, the enormous (possibly infinite) space of input combinations is further magnified, because sequences of element from this infinite space have to be considered. Thus, a finite number of test cases have to be selected from an un-countably infinite set of possibilities. The loss of generality based on this selection process cannot be prevented. Yet, an increase in the number of test cases can raise confidence in the correctness of a system.

In many cases, time to deliver is a critical factor. The success of an embedded-systems supplier is largely dependent on its ability to satisfy complex customer needs within restricted time and monetary budgets. Therefore, testing activities must always be conducted in the limited time-frame of the project. This is why a testing campaign is always assigned end criteria. Usually, a test-end criterion on the system level concerns requirements coverage. Requirement-based testing can be considered the state-of-the-art of functional system-level tests. Safety-related standards [Eur96, ECS09, RTC92] require that every requirement is tested. This is regarded as a practical guideline for a test engineer. Usually, tests will be specified for each requirement and then implemented. Since requirements are in principle as precise, concrete and atomic as possible so they can be implemented by a developer, they can in most cases provide adequate inputs for test-case specification as well. However, in practice, the manual

definition of test cases is time-consuming and error-prone. In most cases, the situation is even worse because of suboptimal requirements, changes in requirements and resulting inconsistencies between requirements, implementation and test cases. The task of keeping a higher-level test-case specification consistent with its concrete implementation is another challenge. All these challenges and issues make testing activities as vulnerable to faults and errors as the developed system-under-test (SUT) that is the target of the testing effort and whose reliability is to be proven by tests.

Model-Based Testing (MBT) can provide an answer to these challenges. MBT aims at the automatic generation of test cases from a formal test model. Instead, of manually defining test cases, a test engineer specifies a test model. Then the test-case specification and implementation can be automatically derived from the test model. This shifts the effort needed for manual test-case specification and implementation to the creation of a test model. Since the test model should be described in an abstract modelling language—e.g., Unified Modeling Language (UML)/Systems Modeling Language (SysML) or a Domain Specific Language (DSL)—the abstraction level of this test model should enable the test engineer to focus on the specification of the expected behaviour of the SUT. Additionally, the test model will describe the SUT or parts of the SUT as a whole, making it possible to specify the interaction of requirements in an abstract formalism. System properties that emerge from the implicit interplay of requirements can be described by such models.

The automation that results from the use of MBT allows for more tests in shorter time periods. Furthermore, inconsistencies can be prevented if test-case specification and implementation are generated from the same source. This single source of information eases change management. Requirement changes result in an update of the test model, and the generated test cases automatically reflect these changes. This is less costly and less error-prone than investigating the impact of changed requirements on a set of a thousands of test cases—especially considering the implicit changes that are attributed to the combination of multiple requirements.

If requirements are traced to model elements, MBT can generate test cases to cover the model elements related to a requirement. Thus, requirements coverage can be achieved by using a set of automatically generated tests. However, MBT allows for more sophisticated coverage criteria than requirements coverage. Having a test model that is independent of a concrete test approach usually makes it possible to apply arbitrary testing strategies and model-coverage criteria. Consequently, the use of MBT makes it possible to profit from any advances in MBT, given that these advances are applicable to the test model under consideration.

Most MBT tools are able to generate test suites which fulfil typical structural-coverage criteria—like covering all states or all transitions in a state machine model. These structural criteria are surpassed by testing strategies that are complete. Completeness cannot be achieved in most cases. If, however, assumptions can reasonably be made about the errors an SUT might exhibit, it is possible to derive test suites that are complete with respect to a fault model. The model describes a set of systems that may or may not contain errors. This set of systems and potential errors is, however, restricted by the a-priori assumptions concerning possible faults. Usually, the set of systems fulfilling the a-priori assumptions—named the *fault domain*—is of infinite size. Given that the SUT behaviour is contained in the fault domain, the test suite is complete: i.e., the test suite will pass if and only if the SUT is correct. For Finite-State Machines (FSMs) and Deterministic Finite-State Machines (DFSMs), many complete testing theories exist. This work deals with a testing strategy [HP16a] that is complete with respect to a fault domain but is applicable to a wider range of systems. Instead of FSMs, which require a finite input alphabet, our approach is applicable to state-transition systems with input variables of

potentially infinite domains. The internal and output variables of these systems are required to be of a finite domain.

We will demonstrate the applicability of our approach to real-world case studies from the railway domain. Railways are one of the oldest industries building complex systems with high safety criticality. Nowadays, legacy systems, like relay-based interlocking systems, are replaced by modern, digital, interlocking systems. Within the European Train-Control System (ETCS) programme, interlocking systems and on-board computers are digitalised and modernised. The high complexity and safety criticality of railways makes the use of formal verification methods mandatory. For modern interlocking systems, a formal model exists, which has been model checked and proven [VHP17] to fulfil safety properties. Our work builds on these results and uses the formal model for HW/SW integration (HSI) tests of modern interlocking systems. Additionally, a speed-monitoring function of the ETCS train on-board computer serves as another case study, demonstrating the applicability of our MBT approach for railways.

1.2 Goals and Contribution

1.2.1 Relation to Previous Work

This work implements a novel MBT testing method originally proposed in [HP16a]. This approach, called Equivalence Class Partition Testing (ECPT) in the following, is based on a sophisticated input-equivalence partitioning which makes it possible to reduce arbitrarily large input domains to a finite number of equivalence classes. Furthermore, the approach has guaranteed (mathematically proven) fault-detection capabilities. This property ensures completeness of the ECPT approach with respect to a formal fault domain: i.e., with respect to a very large, potentially infinite set of potential SUTs. For all members from the fault domain, it is guaranteed that the test suite generated by the ECPT approach is passed if and only if the SUT is correct. Every fault in an SUT that is member of the fault domain will be detected by the test suite.

1.2.2 Main Goal of this Work

This work aims to prove that MBT as a formal verification technique is applicable to a wide range of systems—including complex, real-world case studies from the railway domain. We promote ECPT testing as a way to thoroughly test an SUT with guaranteed completeness with respect to a fault model. To give a convincing argument in favour of ECPT testing, we show that the completeness of our approach actually results in measurable failure-detection capabilities. Therefore, we use a novel mutation analysis that focuses on the test strength evaluation of HSI testing.

1.2.3 Main Contributions

The main contributions of our work to the field of MBT can be summarised as follows:

1. *Extension of the ECPT approach:* This work presents an implementation of the ECPT approach and extensions to the approach invented by the authors of [HP16a]. The extensions include a heuristic optimisation of concrete test data selection: Tests are characterised

by a sequence of Input Equivalence Classes (IECs). To make these "symbolic" test cases executable against an implementation, concrete values from IECs have to be selected. We will show that the concrete input selection from the IEC has a strong impact on the test strength under "real-world" conditions: i.e., for systems that cannot be proven to be captured by a fault domain that can be anticipated a-priori in case of black-box tests. Our aim is to *improve* ECPT for the detection of real faults that are not necessarily part of an anticipated fault domain.

2. *Thorough experimental evaluation:* To gain evidence for improvements of the fault-detection capabilities, we need a thorough experimental evaluation. This evaluation shall investigate the test strength (defined as the fault-detection capability of a testing approach) of the implemented ECPT approach with different input-selection heuristics. Furthermore, it shall allow for comparison with other testing approaches. This work compares all presented heuristics with Random Testing (RT), which serves as a minimal benchmark that has to be surpassed by any sophisticated testing methodology.

3. *Proposal of a Novel HW/SW integration test (HSI test) Evaluation Approach:* The ECPT approach offers functional testing of embedded systems at high abstraction levels: i.e., at the system and HSI levels. Mutation analysis is a way to evaluate testing approaches. By systematic fault injection to a correct version of the SUT (resulting in an erroneous version of the SUT called mutant), the fault-detection capabilities of a test suite can be measured. A variety of work has been done on SW mutation analysis, some of which yields evidence that artificial mutants serve as good surrogates for real faults. However, in the domain of HSI testing, no real means exist by which to evaluate the test strength. State-of-the-art metrics like code coverage are of limited use if testing approaches are evaluated [JJI+14]. This lack of HSI test mutation analysis led to the development of a novel, completely automated, mutant generator of SystemC models. This mutant generator not only mimics typical HW/SW errors, but also errors that can be introduced by HW/SW incompatibilities.

4. *Efficiency considerations for the implementation of the ECPT approach:* This work also aims to provide some insight into the implementation of the testing approach. We present some core algorithms that render the ECPT approach realisable for real-world examples of considerable complexity.

5. *Proposal of a generic approach for compositional testing of route controllers:* Finally, we provide a formalisation of the generic behaviour of a route controller: i.e., a safety-critical sub-component of modern interlocking systems that controls train movements in a railway network. This formalisation can be used to automatically generate a model of route-controller behaviour that can be used to apply our ECPT to components from modern interlocking systems.

1.3 Structure of this Thesis

Chapter 2 summarizes the basics that are needed to understand our approach and to make this work self-contained. Chapter 3 introduces four case studies. Our approach is applied to these models and experiments are conducted on these models. Next, we detail the ECPT approach in Chapter 4 and present our extensions of the ECPT approach. This chapter also details the generic approach for testing route controllers of modern interlocking systems. Chapter 5 introduces our

novel approach for test-strength evaluation of HSI tests. We briefly depict the state-of-the-art of mutation analysis and then motivate and present our new mutation-analysis method by using SystemC mutations. This approach is then used for our experiments. Chapter 6 exhibits and discusses experimental results. Following this, Chapter 7 presents a thorough discussion of related work before this work is concluded in Chapter 8.

2 Background

This chapter summarises the basics that are needed to understand our approach and to make this work self-contained. First, we introduce some basics of the railway domain. Section 2.2 introduces two state-of-the-art approaches to the modelling of systems: SysML and SystemC. Section 2.3 and the following introduce terminology related to testing, a complete testing theory for FSMs and finally the complete testing theory based on input equivalence classes that we will apply in this work.

2.1 The Railway Domain

2.1.1 ERTMS and ETCS

The European Rail-Traffic Management System (ERTMS) is a European standard for rail-traffic management. It has been established by a directive of the European Council [Eur96]. The objectives of ERTMS are to improve the interoperability, capacity and *safety* of rail traffic. The reason for ERTMS is the wide variety of existing legacy signalling and train-control systems used in European countries. The existence of these heterogeneous and incompatible systems hampers the cross-border interoperability of different signalling and train-control systems. This is the main challenge that has to be overcome for train traffic at an international level. It is addressed by the ERTMS standard.

ERTMS is composed of ETCS and the Global System for Mobile Communications – Railway (GSM-R). GSM-R is a railway-specific variant of the widely used mobile-communication standard, GSM. ETCS is the subsystem of ERTMS that includes the signalling, train control and train protection functionality.

Safety is a major concern of the ERTMS and ETCS standard. The goal of enhanced safety in rail transport makes the use of formal methods for verification—as presented in this work for example—mandatory. The European Commission Decision [Eur02] mandates to all railway operators of the European member states that newly created and renewed Trans-European, high-speed railway systems must conform to the ERTMS standard. Thus, the following decades will bring the development of modern railway systems that apply to ERTMS. We believe that the high confidence level needed to ensure safe train operation can only be guaranteed by advanced verification techniques—especially testing methods with guaranteed fault-detection capabilities.

Before we present the basics of the formal methods needed to understand this work, we briefly introduce the core elements of modern railway systems to give a general understanding of the technical domain our verification approaches are applied to. These core elements are the signalling system, which comprises the interlocking system and the on-board train-protection system.

2.1.2 Interlocking Systems

Under *signalling*, we understand the process of controlling train movements by signals, block sections and points. A signal can be a physical track-side element or a virtual on-board signal that is visible to the train driver. In general, block sections are fragments of track that are used for the separation of trains. "A train must generally not enter a block section until it has been cleared by the train ahead" [Pac02]. Signalling has to ensure safe operation of trains in a railway network. The signalling includes a number of technical procedures and non-technical operating rules that ensure the safe operation of trains travelling through the network. An *interlocking system* is the technical part of a signalling system: i.e., the system controlling the signals, points and block sections. Again, the main task of an interlocking system is to ensure the safe travel of trains through the railway network. Therefore, the interlocking system ensures that trains are guided through routes, for which all points are set properly. Conflicting routes are locked, and the track is guaranteed to be clear [Pac02].

This work uses the notions and notations of interlocking systems from [VHP17, PHH16a].

Different types of interlocking systems exist. This work is concerned with route-based interlocking systems. This type of interlocking system is widely used: for example, the complete Danish signalling system is currently replaced by an ETCS-conforming, route-based interlocking system in the context of the Danish signalling programme.[1] In route-based systems, trains are guided through predefined routes. Each route can be exclusively locked for at most one train at a time. This concept makes it possible to prevent hazardous situations such as collisions and derailments. The remainder of this manuscript uses the term interlocking system to refer to route-based interlocking systems.

2.1.2.1 Components of an Interlocking System

An interlocking system is composed of a railway network and–in case of route-based interlocking systems–of an interlocking table.

A *railway network* is the physical/geographical part of an interlocking system. It can be comprehended as the part of a track network that is supervised by the interlocking system. It is composed of different track-side elements: *linear sections, points* and *marker boards*.

A *linear section* is a part of a railway network with at most two neighbouring elements, which can be other linear sections or points. A *point* is a section of the network with three neighbouring elements. Linear sections and points are so called *train-detection sections* (or *sections* for convenience), because the occupancy status (i.e., the status "occupied by train" or "free") can be detected for each section. This detection is usually performed by physical equipment: *axle counters* or *track circuits*. The ends of a linear section are named *down end* and *up end*. Each linear section can be travelled in an *up* or *down* direction. The *up* and *down* directions are the directions in which the distance to a reference location from the network is increasing or decreasing, respectively. The ends of a point are called *stem end*, *plus end* and *minus end*. The plus and minus ends are the diverging branches of the stem. The path from stem end to plus end is the straight path through the point, and the path from stem end to minus end is the

[1] The Danish signalling programme is of importance, because we will use some of the results elaborated in the context of this initiative [Vu15, VHP17]. [Vu15] proposes a formal method for the verification of route-based interlocking systems. The results in this work will be used by us for the application of MBT to modern interlocking systems.

branching path. A point can be switched from its *PLUS* position to its *MINUS* position and vice versa. In the PLUS position, the train can travel from the stem end to the plus end of the point or from the plus end to the stem end, depending on the travelling direction of the train. In the minus position, it is possible for trains to travel from the stem to the minus end or vice versa. *Marker boards* are virtual signals which perform the same task as physical signals from legacy interlocking systems. Marker boards are virtual in the sense that they have no physical representation. Each marker board is associated with a section together with a position and a direction along this section. If a marker board is in state PASS, trains are allowed to traverse the associated location on the section in the associated direction; they have to stop if the marker board is in state HALT.

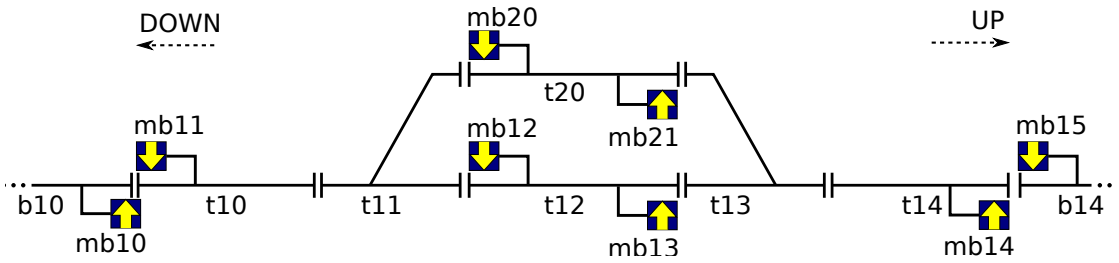


Figure 2.1: Example Railway Network from [VHP17]

Example 1. Consider the railway network from [VHP17], shown in Figure 2.1.

The network is composed of the linear sections t10, t12, t20, t14 and of the points t11, t13. Linear section t20 is connected to point t13 on its up end and to point t11 on its down end. t11 is connected to t10 on its stem and to t12 and t20 on its plus and minus ends. The marker board mb13 is associated with linear section t12 in the up direction.

Additionally, a route-based interlocking system is composed of an *interlocking table*. The *interlocking table* lists a number of predefined routes, which can be allocated by trains. A *route* is a sequence of successive sections to be traversed on the route (called *path elements* of the route), and every route starts and ends at a marker board (called *source* and *destination*). Each route defines the states of its elements that are needed to ensure safe traversal of a train from source to destination. This includes the state of points along the path of the route and the state of protecting points and marker boards. To prevent other trains from entering the route from the branching ends (*flank protection*) of the points or from approaching from the opposite direction (*front protection*), points from outside the path of the route and marker boards can be used. The state of these additional protecting elements guarantees that no other train can accidentally enter the route while a train is traversing this route. Additionally, each route in the interlocking table is associated with a list of *conflicting routes*. The list of conflicting routes lists all routes that must not be used simultaneously because of path elements that are common to these routes or because of conflicting states for protecting elements.

Example 2. Table 2.1 is an example of a possible interlocking table for the network shown in Figure 2.1. Consider for example route 1. The route starts from marker board mb10 and covers the path from t10 over t11 to t12. The point t11 has to be switched to its plus position. Additionally, t13 is switched to its minus position offering front protection of route 1. Flank protection of point t11 is provided by the marker board mb20. Marker boards mb11 and mb12 offer additional front protection for the sections along the path of route 1.

Table 2.1: Interlocking Table for the Network Layout in Fig. 2.1 (Taken from [VHP17]; `p` means PLUS, `m` means MINUS.)

id	src	dst	path	points	signals	conflicts
1	mb10	mb13	t10;t11;t12	t11:p;t13:m	mb11;mb12;mb20	2;3;4;5;6;7
2	mb10	mb21	t10;t11;t20	t11:m;t13:p	mb11;mb12;mb20	1;3;6;7;8
3	mb12	mb11	t11;t10	t11:p	mb10;mb20	1;2;5;6;7
4	mb13	mb14	t13;t14	t13:p	mb15;mb21	1;5;6;8
5	mb15	mb12	t14;t13;t12	t11:m;t13:p	mb13;mb14;mb21	1;3;4;6;8
6	mb15	mb20	t14;t13;t20	t13:m	mb10;mb12;mb13;mb14;mb21	1;2;3;4;5;8
7	mb20	mb11	t11;t10	t11:m	mb10;mb12	1;2;3
8	mb21	mb14	t13;t14	t13:m	mb13;mb15	2;4;5;6

2.1.2.2 Interlocking Principles

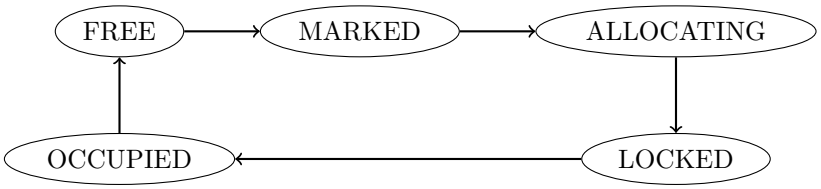


Figure 2.2: State-based Visualisation of the Interlocking Principles

Safety in interlocking systems is ensured by the following rule: A train is only allowed to travel on locked routes and every route is locked exclusively for at most one train at a time.

The locking of a route is performed as visualised in Figure 2.2. Before a route can be locked, it has to be requested or *marked* by the train driver, signalman or traffic management system. The interlocking system starts *allocating* the route, if no conflicting route is currently in allocating or locked state and all path elements of the route are vacant. Allocation means that all points listed in the interlocking table for the respective route are set to their requested position and all marker boards listed for the route are set to HALT. Once all these track elements have come to their requested position, the route is said to be *locked*. The points are physically locked preventing undesired switches of the points. Thereon the source marker board is switched to PASS allowing the allocating train to enter the route. Once the route is *occupied*, this marker board is switched back to HALT to prevent other trains from entering the route. When the train finally leaves the route the route is set back from *occupied* to *free*.

Additionally to these principles, modern interlocking systems provide the *sequential release* feature. *Sequential release* is the concept of releasing, i.e., unlocking, parts of the route that are completely traversed by a train. Subsequently, the sequentially released path elements of the route can be reused by other routes. This allows for higher concurrency of routes.

Example 3. Consider the network from Figure 2.1 and the route table shown in Table 2.1. When a train approaches mb10, it might request route 1 to stay on the main track. In this case, route 1 transits to state MARKED. Given that none of the conflicting routes is in state ALLOCATING or LOCKED and that all path elements (t10, t11, t12) are vacant, the route will transit to state ALLOCATING. Point t11 is commanded to position PLUS and point t13 is commanded

to position MINUS. Signals mb11, mb12 and mb20 are commanded to state HALT. As soon as all of these elements have taken their required state, route 1 transits to state LOCKED. The source marker board mb10 is commanded to state PASS. As soon as the train receives the pass signal from mb10, it is allowed to enter route 1. As soon as the detection status of t10 changes to occupied, route 1 transits to state OCCUPIED and commands mb10 to state HALT. Consider a second train approaching mb10. This train has to stop in front of mb10 because this marker board is in state HALT. The second train might request route 2 from mb10 to mb21. Route 2 transits from FREE to MARKED. Because route 2 conflicts with route 1, the controller of route 2 stays in the MARKED state. As soon as train 1 completely passed t11, route 2 can transit to state ALLOCATING because t10 and t11 were sequentially released by route 1. t11 is commanded to switch to state MINUS. Finally, train 2 is allowed to enter its requested route 2, while route 1 is still in use. When train 1 approaches mb13, it has to request route 4 to continue. Finally, it will be able to enter this route and completely leave t12. This causes route 1 to transit to state FREE again.

2.1.3 On-Board Train Protection

A *train-protection system* has the task to supervise train movements and–in case of human errors– trigger an automatic intervention. This includes the supervision and enforcement of HALT signals and the supervision of the train speed in accordance with the speed limits. In ETCS systems, this task is performed by an on-board system: the European Vital Computer (EVC). The EVC implements speed- and distance-monitoring functionality, among other things, as described in [UNI12]. The speed and distance monitoring function enforces agreement with speed limits. Therefore, the on-board unit of the train reads information from *Eurobalises*: i.e., trackside beacons with a fixed location. Based on the information communicated by Eurobalises, the train knows its position and the maximum allowed speed. Additionally, the train knows the reference location at which it has to stop: the so-called end of authority (EOA). Based on this information, the speed and distance monitoring function operates and supervises the observance of speed and distance limits. The monitoring function operates in three exclusive modes.

Ceiling-speed Monitoring, Target-speed Monitoring and Release-speed Monitoring. The Ceiling Speed Monitor (CSM) is active when a train is travelling on its route and is still far away from its target location (EOA). In this mode, adherence to the track-dependent speed limit has to be supervised. As soon as the train approaches its EOA, the distance to the target location has to be taken into account. In these situations, the Target Speed Monitor (TSM) is active. Braking curves of the train are calculated to ensure that the train decelerates early enough to allow it to come to a standstill before the EOA. The speed limit at the location of the EOA is zero. Because of inaccuracies of the measured location, this can sometimes lead to situations in which the train is not able to reach the exact target location. Therefore, it is desired to allow the train driver to drive up to a very low speed limit (the release speed limit) when the train is very close to the EOA. This mode is supervised by the Release Speed Monitor (RSM).

2.2 System Modelling

This section presents different state-of-the-art approaches to the modelling of systems. We briefly introduce SysML: a very popular modelling language that is favourable because of its wide application in the academic and industrial community and its increasing tool support.

Second, we introduce SystemC. SystemC provides a mode of description for embedded systems and aims at system modelling. We use SystemC as an implementation language in this work. Because of its nature, SystemC is a good candidate for an implementation language which can be used for our mutation experiments aiming at the test-strength evaluation of HSI testing.

2.2.1 Systems Modeling Language

SysML [Obj15b] is a standardised modelling language for the abstract description of systems. SysML is standardised by the Object Management Group (OMG) and is closely related to the UML [Obj15a].

Compared to UML, SysML extends UML by some types of diagrams that are specifically needed for the modelling of systems. SysML makes it possible to define the system composition and the interconnection of subsystems (called *blocks*) by block definition diagrams (BDDs) and internal block definition diagrams (IBDs). Furthermore, SysML allows for the definition of requirements in requirement diagrams and supports the tracing from requirements to other model elements. Another system-modelling feature is the modelling of constraints through constraint blocks that can be visualised in parametric diagrams. For behavioural modelling, SysML provides activity diagrams and state machines. Both are adapted from UML and are only extended in some minor details. Therefore, readers who are familiar with activities and state machines in UML can consider the SysML counterparts to be equivalent.

This work uses state machines as a description means for system behaviour. Since state machines are almost equal in UML and SysML, we will call these behavioural diagrams SysML state machines—although the term UML state machines would be correct as well. If no confusion arises, we sometimes use the term state machines.

SysML State Machines The remainder of this work uses a subset of the SysML state-machine concept for the behavioural description of systems. We present only the features of state machines that are needed for this work to be self-contained. For a thorough and complete definition of SysML/UML state machines, please refer to the standards [Obj15b, Obj15a].

SysML state machines can be considered a variant of the *statecharts* originally invented by Harel [Har87]. State machines, based on Harel's statecharts, are visual, graph-based description means which combine control flow and data flow. A statechart is comprised of states (visualised by nodes with rounded-corners) and transitions (visualised by directed edges between nodes) that connect in direction of the arrow, respectively, a source and a target state. Transitions, i.e., arrows $\xrightarrow{e[g]/a}$ can be labelled with an event e, a guard-condition g and an action a. A transition is taken if the state machine currently resides in the source state, event e occurs and the guard-condition g is fulfilled. The state machine transits to the target state and performs action a. Data flow in state machines is mainly modelled by actions. An action can be an abstract event or, as in our case, a sequence of assignments to variables. Actions are generally performed in zero-time, which means that a sequence of assignments is performed instantaneously. For the modelling of non-instantaneous reactions, state machines allow for the activation of activities, which are by definition executed in non-zero-time. An action may be associated with a transition. For convenience, state machines allow for *entry-* and *exit-actions* and for *do-activities* in states. *Entry-* and *exit-actions* actions are triggered when the associated state is entered, or when the state is left. Do-activities are activities that are started when a state is entered; finally, these activities are stopped when the state is left (or as soon as the activity terminates).

Besides this basic functionality, state machines allow for hierarchy: States are allowed to contain *substates* (Harel uses the term *XOR-states* to indicate that the state machine resides in one of these states at a time). In the context of UML and SysML, states containing substates are called *composite states* or *submachine states*. These states themselves are state machines which define parts of the system's behaviour. Additionally, states are allowed to contain *orthogonal regions* (Harel uses the term *AND-states* to indicate that the state machine resides in one state in all of its orthogonal regions at a time). A region is a subgraph of a state machine (again possibly hierarchical) which defines behaviour that is executed concurrently with its orthogonal regions. Thus, orthogonal regions allow for the modelling of parallelism. Both orthogonal regions and hierarchy allow for comprehensible models even for very complex systems. On the other hand, orthogonal regions and hierarchy impose challenges in the definition of a formal semantics. SysML and UML encounter this challenge by providing only a semi-formal semantics of state machines. Many implementation details are not defined by the standards and are thereby implementation dependant. A formal semantics for Harel's statecharts has been presented in [HN96] together with a thorough discussion of other semantics. This work uses the formal semantics used by the RT-Tester MBT component. The concrete semantics of this tool is laid out in [PVL11]. This tool is based on the semi-formal semantics of the UML standard, though it may violate UML semantics in some details.

The remainder presents some state machine models for real-world systems in the scope of this work. All of these models have in common the fact that they can be expressed (in a comprehensible way) without the use of hierarchy and orthogonal regions. Nonetheless, the reader will find that these models are of considerable complexity. Consider, for example, a route controller that is responsible for the safe operation of trains in routes through large, real-world railway networks. We therefore neglect further explanation of the formal semantics of hierarchic state-machine states and parallelism in state machines. Instead, we intuitively introduce "flat" SysML state machines without hierarchy and parallelism. Note that this is no restriction on the implementation of our approach: Since our approach relies on the RT-Tester MBT component and the semantics [PVL11] implemented therein, the implementation of our approach can be applied to more complex models than are presented in this work. The formal semantics that is based on Reactive Input-Output State-Transition Systems (RIOSTSs) is introduced below, in Section 2.5.1.

Definition 1 (Flat SysML State Machines)**.** A flat SysML state machine is described by a tuple $\texttt{SM} = (S, s_0, T, e, x)$. S denotes the set of states of the state machine that are visualised by rectangular nodes with rounded corners in the graphical representation. $T \subset S \times G \times A \times S$ denotes the set of transitions in the state machine. A transition $(s, g, a, s') \in T$ is visualised as an arrow from state s to state s' and is labelled $[g]/a$. g is called the guard-condition. In this work, guard-conditions are considered first-order-logic predicates over system variables. A guard condition $g \in G$ is a member of the universe of predicates over system variables, denoted by G. $a \in A$ is the action associated with the transition. An action a is a finite sequence of assignments to system variables. An assignment a has the form $l = r$ where the left-hand side l is a variable name and r is an expression over system variables. The effect of the assignment is that, in the target state of the transition, the variable referenced by l is set to the value the expression r evaluates to in the current system state. The universe of possible actions is denoted by A. A transition $(s, g, a, s') \in T$ is taken as soon as guard-condition g becomes true, given that the state machine resides in state s. The state machine resides in exactly one state at any moment in time. Taking the transition causes the state machine to switch the state it resides in to the target state s' and perform the exit action $x(s)$ of s, the action a associated with the transition and the entry action $e(s')$ of the target state s'. e and x are partial functions which map a

state to its entry and exit actions, which are again a sequence of assignments. Initially, the state machine resides in the pseudo-initial state s_0, which is visualised by a black-filled circle. Entry- and exit-actions will be visualised inside of the nodes of a state with the prefixes "entry/" and "exit/". The action of a transition can be empty. In this case, $/a$ is dropped from the graphical representation.

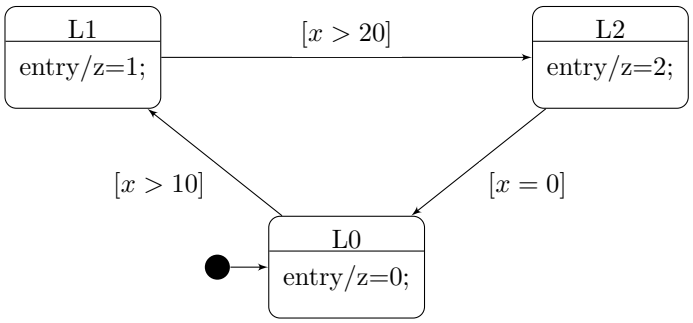


Figure 2.3: Example State Machine

Example 4. Figure 2.3 is an example for a flat SysML state machine. Consider the system variable x to be an input variable to the system. Let z be an output variable of the system. Initially, the system resides in the state-machine state "L0". As soon as x becomes greater than 10, the system transitions to state "L1" and the entry action causes z to be set to one. If x becomes greater than 20 the system transitions from "L1" to "L2" and z becomes two. As soon as x becomes zero, the system transitions back to state-machine state $L0$ and z is set to zero. If the system resides in state "L0" and x changes its value to a value greater than 20, the system will transition to state "L1", and because the guard condition of the outgoing transition of "L1" is fulfilled, the state machine will immediately make this transition as well. The two successive transitions from "L0" over "L1" to "L2" are called a *compound transition* in the context of UML. The *run-to-completion* semantics causes the two transitions to be made at once: i.e., in zero-time. The *run-to-completion* semantics prescribes that transitions are made until a stable system state is reached. In this case, $L2$ is stable, because no guard condition of an outgoing transition is fulfilled. The system input has to change before the next transition can be made.

Readers who are familiar with the UML/SysML standard will notice that the flat state machines introduced above do not include many of the features that UML/SysML defines. For example, we completely neglect events that are used as triggers in state-machine transitions. It might seem unnatural to use an originally event-based approach like SysML state machines in this case, but the models we present as case studies in Chapter 3 all model control systems that somehow rely on input variables from large domains (analogue inputs or train constellations in a railway network) and make concrete control decision based on these inputs. We believe that systems of this kind are well described by a state-based approach using the state-machine subset as introduced above. However, it has to be emphasised that the testing approach introduced below does not rely on this subset of state machines; it does not even rely on a concrete description means. We will show that every concrete description means that can be translated to a state-transition system with dedicated input and output variables can be equipped with our testing methodology. Note that every event-based model can be translated to an equivalent state-based formal model. Thus, SysML state machines or the variation of state machines as introduced above should be considered an interchangeable front-end to the testing approach implemented in this work.

2.2.2 SystemC

SystemC is a high-level system modelling and design language. It is integrated into C++ by means of class libraries. SystemC is technically not a language of its own but rather a set of class libraries in C++ that allow for the modelling of systems. The main goal of SystemC is to bridge the gap between different levels of system abstraction and to allow the co-design of SW- and HW-architecture. A key feature of SystemC therefore is the simulation kernel. This simulation kernel makes it possible to simulate a SystemC model: i.e., to execute the model. The main task of this kernel is the concurrent simulation of functional units, called *modules*. It is the nature of integrated embedded systems that many modules run in parallel. SystemC therefore mostly focuses on the definition of modules, the functionality of modules, communication between modules, and the concurrent execution of the complete model composed of concurrent modules. For a thorough introduction to SystemC, refer to [BABJ10].

The most important elements of the SystemC language that we use throughout this work are *modules*, *signals* and *ports*. These are introduced in subsequent paragraphs. We conclude with a small example of a SystemC model.

Modules SystemC makes it possible to design a system by defining modules. A module is a unit that implements some functionality. The behaviour of a module is implemented by an arbitrary number of methods. These methods can either be called directly or called in reaction to a value change of a signal.

Modules make it possible to define a hierarchy: modules can contain other modules, which in turn might contain other modules. This induces a hierarchy that is typical for complex systems that are modelled compositionally.

Technically, modules are implemented by custom classes that extend the common base class `sc_module`.

SystemC Threads and Methods Since modules are instances of custom user classes that extend `sc_module`, the functionality of modules is to be implemented in member functions of the module's class. Besides traditional C++ member functions that can be called by the user, SystemC provides two ways to define simulation processes. A *simulation process* is an executing instance. This may be a SW-thread or a SW-process running on an operating system. In case of hardware, this may be an independently timed hardware module. Simulation processes are executed by the SystemC simulation kernel. They must not be called by the user through function calls, but may only be called by the SystemC simulation kernel. The invocation by the simulation kernel in turn is indirectly caused by sensitivity, events and notification. Because simulation processes are to be called by the simulation kernel, these processes need a common signature. Processes are therefore required to return void and have no parameters. Functionality in these processes therefore needs to use the module's variables as means for input and output.

SystemC distinguishes two types of simulation processes: *methods* and *threads*. While methods are expected to always terminate, a thread may be non-terminating and can be suspended and resumed. Methods are called multiple times by the simulation kernel and the timing model for methods dictate that no (simulated) time passes between the invocation and return of the method. Contrary, threads are called exactly once by the simulation kernel—usually at the beginning of the simulation. These threads usually do not terminate but suspend, allowing simulated time to pass and other simulation processes to be executed. Methods can be registered

to the simulation kernel by using the `SC_METHOD` macro, and threads are registered using the `SC_THREAD` macro. The registration has to be performed in the constructor of a module. The phase in which the constructors of all modules are called is named *elaboration phase*.

Channels, Interfaces and Ports An important aspect of system modelling is communication. In SystemC, communication between modules is modelled by *channels*. Channels interconnect modules and allow the interconnected modules to communicate with each other. A *channel* is an abstract way to model a communication channel. Channels may be used to model every kind of communication means including high-level communication means like sockets, buses, FIFO-queues; and low-level communication means like physical wires transferring logical values. SystemC distinguishes between two types of channels: *primitive* and *hierarchical* channels. *Primitive channels* (base class `sc_prim_channel`) represent low-level (i.e., fast and simple) communications. Primitive channels are not hierarchical: i.e., they must neither contain modules nor simulation processes. `sc_signal` is an example of a primitive channel. In contrast, *hierarchical channels* (base class `sc_channel`) usually model higher-level communications. A hierarchical channel itself is a module that implements the functionality of the communication. This implies that a channel may contain modules and simulation processes.

Channels provide functionality through *interfaces*. An *interface* (base class `sc_interface`) declares a set of methods that the channel has to provide. Usually, a channel implements several interfaces, such as one writer/producer interface and one reader/consumer interface. The concept of interfaces makes it possible to separate the interface of a communication channel from its functionality. Technically, the channel implements interfaces by extending them and overriding virtual member functions defined in the interfaces. Usually, the virtual member functions in the interface are pure virtual functions.[2]

As stated earlier, modules are interconnected via channels. Therefore, *ports* (class `sc_port`) are used to connect (*bind*) a module to the channel. A module may have multiple ports and every port is bound to a channel (it is also possible to bind one port to multiple channels). Ports are parametrised `sc_port<I>` to an interface `I`. This interface determines the functions that may be called on this port (namely the functions the interface `I` declares) and the channels that the port may bind to (namely, all channels that implement `I`). The ports act as proxies that forward an interface function call to the channel the port is bound to.

The connection of modules (i.e., the binding of ports to channels) has to be done in the elaboration phase: i.e., before the simulation has been started. A port that is not bound to a channel is considered a modelling error and will result in an error when the model simulation is started.

Signals As mentioned, `sc_signal<T>` is a special form of a primitive channel. `sc_signal<T>` represents signals of type `T` in the common meaning originating from HW description languages such as VHDL. Thus, a signal can be considered a type of data storage, and the type of data a signal can hold is customizable. A signal in SystemC can hold values of any of the following C/C++ fundamental types: boolean type, character types (`char`, `signed char`, `wchar_t`, ...), integer types (`int`, `long int`, `unsigned int`, ...) or floating-point types (`float`, `double`, `long double`).

[2] In C++ pure virtual functions are abstract functions that are declared but not implemented. Pure virtual functions correspond to abstract methods in Java for example.

Signals follow the *evaluate-update paradigm*: The SystemC simulation kernel executes all simulation processes in cycles, so called *delta cycles*. The order of simulation processes in a delta cycle is not prescribed by the simulation kernel. Processes might be executed in arbitrary order. This may cause problems when multiple modules want to read and potentially update a signal's value. In this case, the order of execution might influence the final result. The *evaluate-update paradigm* overcomes this problem by the separation of a signal's value to the *current* and *new* value. Within a delta cycle, all modules that read the signal value will read the current value. An update of a signal value results in a write to the *new* value and a notification to the simulation kernel of the changed value. After all processes in a delta cycle have been executed, the change of the value becomes apparent. The current value will be set to the new value, and, in the next delta-cycle, other modules will be able to read this new value.

A special type of signal exists: `sc_clock` models a boolean signal that switches its values with a configurable frequency. This type can be used to model cyclic timing in SystemC models.

For the connection of modules to signals, two special ports can be used: `sc_in<T>` and `sc_out<T>`. `sc_in<T>` models an input port that allows to read a signal of type `signal<T>` and `sc_out<T>` models an output port that allows to write a signal of type `signal<T>`. The special port `sc_inout<T>` allows to read and write the signal the port is bound to.

Events Besides channels, SystemC provides events as a low-level communication and synchronisation scheme. Events are instances of a class `sc_event`. An event can be "fired" or notified by a call to the member function `notify()`. The firing/notification of an event causes all simulation processes (methods and threads) that are *sensitive* to this event to be simulated: Methods are called and threads, which are suspended, are resumed by the simulation kernel. Again, the order of execution of the processes is not prescribed.

Sensitivity As mentioned before, a module's processes are simulated whenever an event the module is sensitive to is notified. SystemC distinguishes between *static* and *dynamic sensitivity*. Static sensitivity to an event `e` is declared in the elaboration phase (i.e., in a module's constructor) by a statement: `sensitive<<e;`. Whenever this event is notified, the simulation process of the module is called. Besides, *dynamic sensitivity* can be changed during simulation. A call to `wait(e)` will cause a thread to suspend until event `e` is notified. The thread is only dynamically sensitive to this event, meaning that after the call of `wait(e)` returned, the thread will not be sensitive until `wait` is called again.

A module can be sensitive to channels as well. In this case, events are used as the low-level means for the activation of processes by channels. The special primitive channel, `sc_signal`, for example, implements member function `value_changed_event()`, which returns an event object that will be notified by the simulation kernel whenever the signal's value changes.

Example 5. Listing 1 displays a SystemC model that models a hardware shift register of four bits. The shift register (SystemC module `shift_register`) has a boolean input, `data` (input port `data`), and is synchronised on a clock (clock input port `sync`). Every clock cycle, all register values are shifted to the next register and the value of `data` is written into the first register. The output of the shift register is a four-bit variable (type `sc_uint<4>`); see output port `out`.

The shift register is composed of four single-bit registers implemented by module `bool_register`. This module has two input ports: `sync` is a clock input and `in` is the input line of the register. The value on this line is read and written to the output of the register on every positive

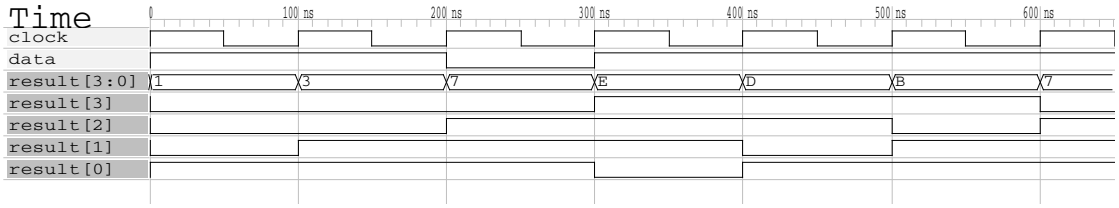


Figure 2.4: Waveform View of HW Shift Register Signals

clock edge. This is realised by the simulation process `update`. The module `bool_register` uses static sensitivity, which is declared in the constructor of `bool_register`. The statement `sensitive<<sync.pos();` makes the register sensitive to positive clock edges.

Module `shift_register` contains the four single-bit registers in a special vector (using the SystemC type `sc_vector<bool_register>`). This vector contains exactly four instances of type `bool_register`. During the elaboration phase, in the constructor of `shift_register`, the four `bool_register` modules are interconnected. Their input and output ports are bound to different signals (member variable `signals`). The `shift_register` module uses static sensitivity for its simulation process (member function `update`). `update` reads the signal values and composes the boolean values to a four-bit integer value that is written to the output port.

Note that the SystemC model works correctly because of the *evaluate-update paradigm* used for signals. If we used traditional member variables instead of signals and ports for the registers, the order of execution of the simulation processes of the `bool_register` instances would influence the result. The result would only be correct if `registers[3]` was scheduled before `registers[2]` which was scheduled before `registers[1]` and so on. Thus, additional synchronisation to ensure a specific ordering of simulation processes would be needed. Instead, the *evaluate-update paradigm* ensures that all registers see the signal values that were valid at the beginning of the clock cycle. No additional synchronisation is needed in this case.

Function `sc_main()`, shown in Listing 2, is the entry point for the execution of SystemC programs. In this function, an instance of type `shift_register` is created and the input ports are bound to an instance of type `sc_clock` named `clock` and an instance of type `sc_signal<bool>` named `data`. The output port is bound to a four-bit integer signal, named `result`.

Afterwards, a trace file is created to log the values of the signals `clock`, `data` and `result` during simulation. Afterwards, the inputs are set and the simulation is performed by `sc_start( 100,SC_NS)`, which causes the simulation of 100 nanoseconds, which is exactly one clock cycle.

Figure 2.4 shows the waveform view that is created from the traces of the SystemC signals `clock`, `data` and `result`. The waveform view visualises the time-dependent evolution of the signal values.[3]

[3]The waveform view has been created using the open source tool GTKWave `http://gtkwave.sourceforge.net/`

Listing 1 SystemC HW Shift Register Example

```
#include "systemc.h"

SC_MODULE(bool_register) {
        //input ports
        sc_in_clk sync;
        sc_in<bool> in;

        //output ports
        sc_out<bool> val;

        SC_CTOR(bool_register) {
                SC_METHOD(update);
                sensitive<<sync.pos();
        }

        void update() {
                val.write(in.read());
        }
};

SC_MODULE(shift_register) {
        //input ports
        sc_in_clk sync;
        sc_in<bool> data;
        //out port
        sc_out<sc_uint<4> > out;

        //signals to be used for registers
        sc_signal<bool> signals[4];
        //vector of 4 single registers
        sc_vector<bool_register> registers{"R",4};

        SC_CTOR(shift_register) {
                //interconnect registers: bind ports in and val to signals
                registers[0].sync(sync);
                registers[0].in(data);
                registers[0].val(signals[0]);
                signals[0].write(false);
                for(unsigned int i=1;i<4;i++) {
                        registers[i].sync(sync);
                        registers[i].in(signals[i-1]);
                        registers[i].val(signals[i]);
                        signals[i].write(false);
                }

                //make shift register sensitive on sync
                SC_METHOD(update);
                sensitive<<sync.pos();
        }

        void update() {
                int sum=0;
                for(int i=3; i>=0; i--) {
                        sum=sum<<1;
                        sum=sum|signals[i].read();
                }
                out.write(sum);
        }
};
```

Listing 2 SystemC Example for Main Method

```
int sc_main(int argc, char* argv[]) {
        sc_clock clock("clock", 100, SC_NS);
        sc_signal<bool> data;
        sc_signal<sc_uint<4>> result;

        shift_register sr("SHIFT_REGISTER");
        sr.sync(clock);
        sr.data(data);
        sr.out(result);

        //open VCD file
        sc_trace_file *trace_file = sc_create_vcd_trace_file("register");
        //dump the desired signals
        sc_trace(trace_file, clock, "clock");
        sc_trace(trace_file, data, "data");
        sc_trace(trace_file, result, "result");

        data.write(true);
        //simulate one clock cycle
        sc_start(100, SC_NS);
        //simulate one clock cycle
        sc_start(100, SC_NS);

        data.write(false);
        //simulate one clock cycle
        sc_start(100, SC_NS);

        data.write(true);
        //simulate one clock cycle
        sc_start(100, SC_NS);
        //simulate another clock cycle
        sc_start(100, SC_NS);
        //simulate another clock cycle
        sc_start(100, SC_NS);
        //simulate another clock cycle
        sc_start(100, SC_NS);
        sc_close_vcd_trace_file(trace_file);
        return 0;

}
```

2.3 Testing

What follows introduces definitions and nomenclature related to testing.

2.3.1 Definitions and Nomenclature

Testing is an approach that checks whether an SUT fulfils a set of requirements, i.e., its *specification*. In general, this is done by running *test cases* against the SUT. A *test case*, according to [RTC92], is comprised of a sequence of input data, preconditions and a set of expected outputs. Each test case aims to verify that a subset of requirements has been correctly implemented. In the remainder we will distinguish between *abstract test cases* and *concrete test cases*. *Abstract test cases*, sometimes referred to as *symbolic test cases*, are abstract descriptions of a test case in the sense that input or output data might not be specified in a concrete but rather in an abstract way. In our approach, an abstract test case is a test case on the DFSM abstraction level. Therefore, an abstract test case is specified by a sequence of IECs and a sequence of expected outputs which are concrete, because we consider systems with a finite output domain in our work. In contrast, a *concrete test case* is a test case in which inputs and outputs are assigned concrete values or tuples of concrete values for systems with multiple input and output variables. In our work, we derive the concrete test cases from abstract test cases by selecting concrete members from IECs. These concrete test cases are directly applicable to the SUT using the concrete input values or value tuple as stimuli and observing the expected concrete output values. To make a concrete test case executable by a machine, a test procedure is derived from the test case. A *test procedure* is the representation of a test case in a specific (interpreted or compilable) programming language to make the test case executable on a machine. The term *test suite* will be used interchangeably to denote a set of test cases or test procedures.

For system and software development, different development processes exist. A widely known development process is the *V-Model*. Following this process model, a system is developed in stages, as shown in Figure 2.5. The phases are arranged in two branches: the development branch and the V&V branch. Both branches form the letter V, which grounds the name "V-Model". The system is designed and implemented following the left branch, following a waterfall approach in which later phases of the design become more detailed in refinement steps. First the requirements are detailed in an architectural design, and this first design is subsequently elaborated in more detailed steps, finally resulting in an executable system. This system is then assessed for correctness in the verification and validation (V&V) branch. *Verification* subsumes all techniques that aim to demonstrate that a developed artefact fulfils its specified purpose. In contrast, *validation* activities aim to demonstrate that the specified purpose is identical to the intended purpose. As such, verification activities ensure that "the system is built right" while validation activities ensure that we "build the right system". Testing is a typical verification method, and in this work, we focus on testing and thus on verification. By analogy to the phases of the development branch, the system is verified at different levels. Thus, different testing approaches exist. During *unit testing*, single modules (software units, classes, functions, and hardware subsystems) are tested in isolation. In later phases, integrated modules are tested by integration testing. A special case of *integration testing* is HSI testing, which comprises integration tests that examine integrated (sub-) systems which contain software and hardware components. Usually, integration tests are followed by *system testing* in which the final integrated system is tested.

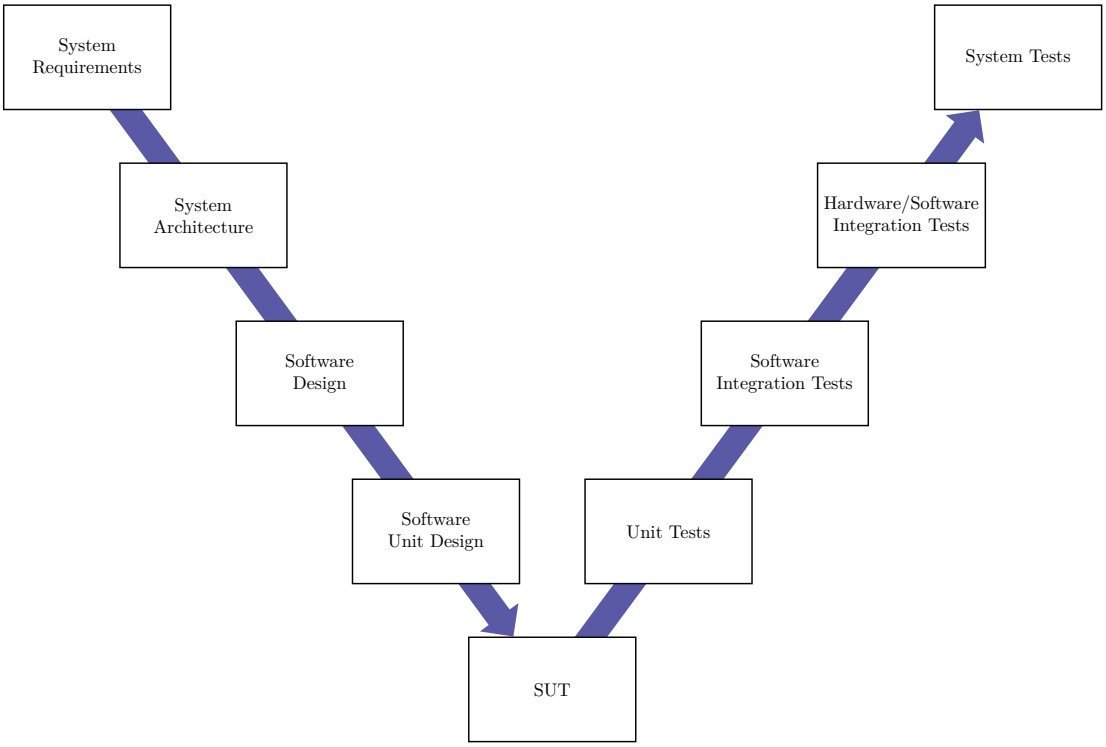


Figure 2.5: Phases of the V-Model

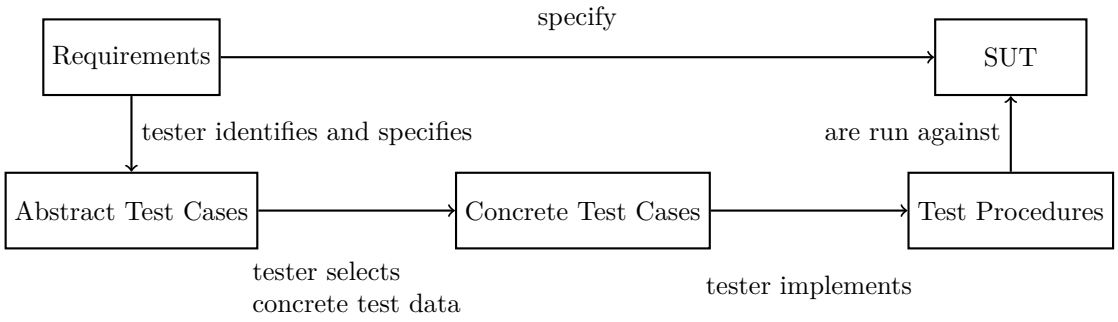


Figure 2.6: Illustration of the Requirement-based Testing Paradigm

Testing approaches can further be categorised as *white-box* and *black-box* approaches. *Black-box testing* is based on the interfaces and behavioural specifications of a system. Based on this, test cases are used to verify that the system behaves as specified on its external interfaces. For *white-box testing*, additional implementation-specific information is considered as well. In most cases, source code and source code coverage is considered in white-box approaches. While white-box tests allow for more detailed observations of the SUT, these approaches require the observability and sometimes controllability of the internal state of a system. This can be achieved by code instrumentation techniques, for example. In many cases, this is neither possible nor desirable. Therefore, black-box approaches are desired mainly on higher levels of the V&V life cycle, because the high abstraction and complexity level precludes a reasonable use of low-level information. Note that, for final system tests, safety-related standards [RTC92, Eur01, ECS09] mandate that tests are run on the original system, precluding instrumentation.

The approach presented in this work is a black-box approach. This approach is applicable to different test levels. It is in general applicable to unit, integration and system testing. However, because the approach is specific to reactive systems, we believe that the main scope of our approach is not at the unit testing level but at the integration and system level. The evaluation approach that is presented in Chapter 5 aims to demonstrate that our approach is applicable as an HSI test approach with reasonable fault-detection capabilities. There is, however, no reason why it could not be applied to lower levels of V&V activities.

2.3.2 Requirement-Based Testing

All safety-related standards [RTC92, Eur01, ECS09] require full requirements coverage of verification activities. Therefore, the state-of-the-art approach for verification of safety-related systems is to show that a set of usually manually specified test cases associated with manually implemented test procedures covers all requirements. This is done by *requirements tracing*. Each test case defines a set of requirements that is tested by the test case. This results in a traceability matrix from test cases to requirements and vice versa. One test case may be designed for each single requirement. But in general, the traceability matrix may be an n-to-m mapping. Requirements that are not verifiable by tests must be covered by other verification means, but this is an exception that has to be justified. Figure 2.6 illustrates the concept of requirement-based testing.

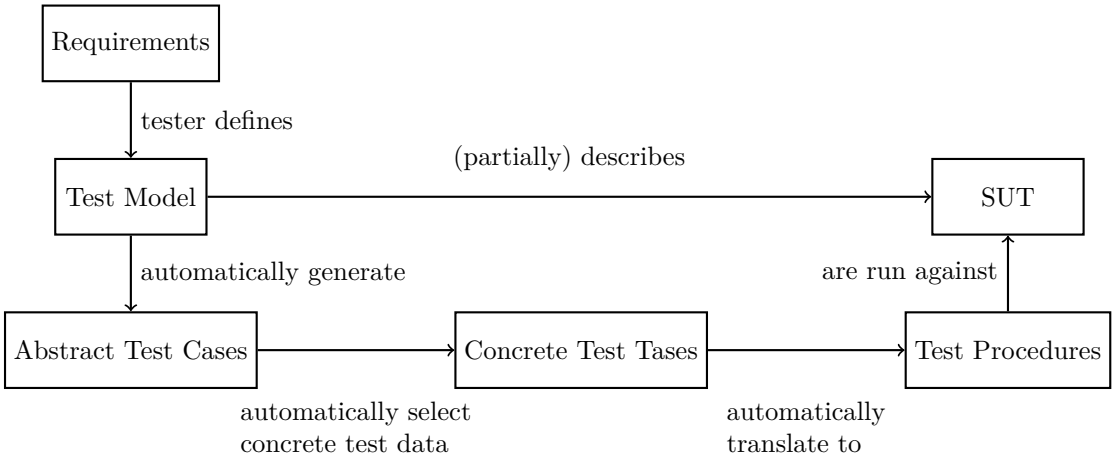


Figure 2.7: Illustration of the MBT Paradigm

Although further criteria for verification activities exist (e.g., code-coverage criteria are prescribed depending on the safety criticality of the SUT), the testing activities at the system level satisfy the requirements of safety standards as soon as every requirement is covered by at least one test. Given the informal nature of requirements and the fact that the interaction of requirements and evolving system properties are not considered in these requirements, it is easy to recognise that such an approach in isolation is far from complete. It has to be noted that safety standards recommend the use of complementary methods like MBT and model checking in addition to requirement-based testing activities.

2.3.3 Model-Based Testing

No unique definition of MBT exists. MBT can be understood to do one of the following:

1. use models to describe tests, or
2. derive tests from a model.

MBT as defined by the first definition often uses scenario-oriented notations to model sequences of inputs and outputs: e.g., UML sequence diagrams. From these diagrams, executable test cases can automatically be generated. In this case, the test case itself, the sequence of inputs and outputs is manually defined. In contrast to this, we understand MBT, following the second definition, as follows:

"Model-based testing usually means functional testing for which the test specification is given as a test model. [...] In model-based testing, test suites are derived (semi-) automatically from the test model" [Wei09, p. 31].

This definition, which defines MBT as the formal process of deriving tests from a formal model of the SUT, is illustrated in Figure 2.7.

First a test model is defined. Usually, this test model is designed by a tester or a test team. This model can be derived from the requirements or, in case of *model-based development*, this model

can be deduced from the development model. In any case, the model itself is an abstract formal description of the intended behaviour of the SUT. Depending on the MBT approach, this may be a complete model of the SUT I/O behaviour or a partial model that describes only some of the behavioural aspects of the SUT or a subsystem.

The test model itself does not contain the test cases to be run against the SUT, since a test case itself is one execution (i.e., a trace) of the model. Thus, the test model defines a generally infinite universe of possible test cases. An MBT approach identifies relevant abstract test cases usually based on some test criteria. For each test case, concrete test data (concrete input and output values) can be calculated from the model. Finally, for a given target language and test framework, an executable test procedure can be generated automatically from the identified set of relevant test cases.

Most MBT approaches differ in the modelling formalism that is supported, ranging from FSMs, through Labelled Transition Systems (LTSs) to state-oriented notations like the UML/SysML state machines, and in the test selection/test case generation algorithms. For an overview of different approaches, refer to Section 7.1. The approach that we present in this work uses state-oriented formalisms (e.g., SysML state machines), but other formalisms that can be expressed by a special variant of State-Transition Systems (STSs) can be used as well.

MBT should be used for the same reasons as model-based development. It allows a higher level of abstraction than hand-written tests. It shifts the focus from specifying test cases and writing test procedures to the design of a test model. The manual process of test-case identification and implementation of test procedures can be automated. This process is less error-prone and cumbersome than the manual approach. Furthermore, it makes it possible to identify more relevant tests than a single tester, even with high experience, could identify. Completeness with respect to some formal criteria can be guaranteed using MBT. The automation ensures a high test quality by preventing inconsistencies between test specifications and test procedures, missing assertions or wrong input stimuli. Given that test execution can be automated in an efficient way, the execution of a large sets of test cases can be enabled. Given that the MBT approach identifies relevant test cases, a larger set of test cases usually should result in higher fault-detection capabilities (test strength) and thus in a more reliable final system. Finally, given that the modelling formalism supports tracing of model elements to requirements, the traceability matrix that is needed to show completeness of tests with respect to requirements can automatically be generated.

It has to be noted that testing is in general an *incomplete* verification method. Most likely, not all possible inputs or sequences of inputs for reactive systems can be tested. Therefore, the selection of a finite subset of all possible test cases always results in the incompleteness of the test results. A system that passes a test suite is not necessarily correct. This fact is stated in the following quotation of Dijskstra [Dij72]:

> Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.

Given the incompleteness of testing, the question is how to make tests, if not complete, at least as close as possible to complete. Therefore, special care must be taken in the process of test-case selection. Given a formal model of possible faults that are expected in an SUT, one can generate test suites that are complete with respect to a given fault model.

For FSMs, complete testing theories exist: i.e., test generation algorithms that result in finite test suites that are complete with respect to a fault model.

2.4 Testing Theory of Finite-State Machines

Finite-state machines offer a means for modelling finite systems. The semantics of FSMs have been established since the early days of computer science. For an introduction, please refer to [Gil62, Gin62, Boo67, HMU06]. FSMs lend themselves well to the modelling of communication protocols. Plenty of work exists that focuses on test-case creation from FSM models [Cho78, FvBK+91, NT81, Gon70, SD85, ADLU91, SLD92]. These test approaches guarantee the completeness of the resulting test suite. This property is excellent for the testing of safety-critical systems, as it assures that certain types of errors will be revealed by the test approach. The background of complete testing theories for FSMs is presented in this section. The next section generalizes these testing theories to cope with systems that are not finite in their input domains.

2.4.1 Finite-State Machine Definitions and Notations

Definition 2 (Finite-State Machine)**.** An FSM is a tuple $M = (Q, \mathfrak{q}, \Sigma_I, \Sigma_O, h)$. Q is a finite set of *states*, which contains the *initial state* $\mathfrak{q}$. Σ_I is the *input alphabet*: i.e., a finite set of inputs $\mathfrak{x}$. Σ_O is the *output alphabet*, again a finite set of outputs $\mathfrak{y}$.

h is a *transition relation* $Q \times \Sigma_I \times \Sigma_O \times Q$ that relates a pre-state q and an input $\mathfrak{x}$ to an output $\mathfrak{y}$ and a post-state q'.

The fact that two states q and q' are related by h via an input $\mathfrak{x}$ and an output $\mathfrak{y}$: i.e., $(q, \mathfrak{x}, \mathfrak{y}, q') \in h$, is denoted by $q \xrightarrow{\mathfrak{x}/\mathfrak{y}} q'$. State q' is said to be a post-state of q. This state can be *reached* when input $\mathfrak{x}$ is applied to M, while M resides in state q. In this case, the output $\mathfrak{y}$ is produced.

An FSM is *completely specified* if, for every pair of states and inputs $(q, \mathfrak{x})$, there exists at least one pair of output and post-state $(\mathfrak{y}, q')$ such that $(q, \mathfrak{x}, \mathfrak{y}, q') \in h$.

Definition 3 (Deterministic Finite-State Machine)**.** A DFSM is an FSM M with the following properties: M is completely specified, and for every pair of states and inputs $(q, \mathfrak{x})$ there exists *exactly* one pair of output and post-state $(\mathfrak{y}, q')$ such that $(q, \mathfrak{x}, \mathfrak{y}, q') \in h$.

The property of h makes it possible to define two functions, δ and ω.

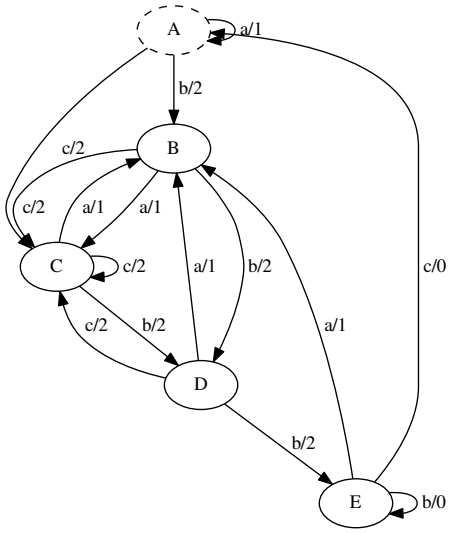
δ is the *transition function* $Q \times \Sigma_I \to Q$ mapping a state q and an input $\mathfrak{x}$ to a target state q'.

ω is the *output function* $Q \times \Sigma_I \to \Sigma_O$ mapping a state q and an input $\mathfrak{x}$ to an output $\mathfrak{y}$.

Since δ and ω contain the same information as h, both can be used instead of h. Therefore, a DFSM M is defined as $M = (Q, \mathfrak{q}, \Sigma_I, \Sigma_O, \delta, \omega)$.

There are two common ways to represent a DFSM. *State chart notation* is used as a visual representation of a DFSM M. It represents M by a graph. The vertices of the graph represent the states Q. The edges in the graph represent δ and ω. Every edge is labelled with $\mathfrak{x}/\mathfrak{y}$ and is unidirectional. An edge from state q to state q' that is labelled by $\mathfrak{x}/\mathfrak{y}$ represents the fact that $\delta(q, \mathfrak{x}) = q'$ and $\omega(q, \mathfrak{x}) = \mathfrak{y}$. Instead of the state-chart notation, a DFSM can be represented by a *transition table*. The transition table is a tabular form that defines δ and ω.

Example 6. Figure 2.8a is an example of the state-chart notation. The state chart defines a DFSM $M_1 = (Q, \mathfrak{q}, \Sigma_I, \Sigma_O, \delta, \omega)$ with state set $Q = \{A, B, C, D, E\}$, input alphabet $\Sigma_I = \{a, b, c\}$ and output alphabet $\Sigma_O = \{0, 1, 2\}$. Figure 2.8b is the transition table of the same DFSM M_1.



(a) State Chart Notation

	δ			ω		
	a	b	c	a	b	c
A	A	B	C	1	2	2
B	C	D	C	1	2	2
C	B	D	C	1	2	2
D	B	E	C	1	2	2
E	B	E	A	1	0	0

(b) Transition Table

Figure 2.8: Example DFSM M_1

For DFSMs, we further use the following notions and notations for convenience. A finite sequence of states $q^* = q_0 \cdot q_1 \cdot \ldots \cdot q_l$ where all states are connected by a transition in M $q_0 \to q_1 \to \ldots \to q_{l-1} \to q_l$ is called a *trace*. Likewise, a finite sequence of inputs $\mathfrak{x}^* = \mathfrak{x}_1 \cdot \mathfrak{x}_2 \cdot \mathfrak{x}_l$ is called an *input trace*.

Operator $\cdot$ is used as a binary operator to concatenate two sequences of arbitrary elements of type T (e.g., states, inputs, ...): $\cdot : T^* \times T^* \to T^*$. We also reuse this operator to concatenate sequences of type T with elements of T: $\cdot : T^* \times T \to T^*$, or to combine two elements of type T into a sequence: $\cdot : T \times T \to T^*$. We also use $\cdot$ as a binary operator for sets of sequences: $\cdot : \mathbb{P}(T^*) \times \mathbb{P}(T^*) \to \mathbb{P}(T^*)$. In this case, $\cdot$ is defined as $A \cdot B \triangleq \{a \cdot b | a \in A, b \in B\}$.

Applying the input trace $\mathfrak{x}^* = \mathfrak{x}_1 \cdot \mathfrak{x}_2 \cdot \ldots \cdot \mathfrak{x}_l$ to a DFSM state q_0 yields a trace $q^* = q_0 \cdot q_1 \cdot \ldots \cdot q_l$ determined by the state transitions $q_0 \xrightarrow{\mathfrak{x}_1/\mathfrak{y}_1} q_1 \xrightarrow{\mathfrak{x}_2/\mathfrak{y}_2} \ldots \xrightarrow{\mathfrak{x}_l/\mathfrak{y}_l} q_l$. The sequence of outputs $\mathfrak{y}^* = \mathfrak{y}_1 \cdot \mathfrak{y}_2 \cdot \ldots \cdot \mathfrak{y}_l$ that is generated by this transition sequence is called *output trace*. The sequence $\mathfrak{l}^* = (\mathfrak{x}_1, \mathfrak{y}_1) \cdot (\mathfrak{x}_2, \mathfrak{y}_2) \cdot \ldots \cdot (\mathfrak{x}_l, \mathfrak{y}_l)$ is called *I/O trace*. A transition sequence that is performed when applying an input trace $\mathfrak{x}^*$ is abbreviated by $q_0 \xrightarrow{\mathfrak{x}^*/\mathfrak{y}^*}{}^* q_l$. If the output trace is not relevant, this transition sequence will be abbreviated as $q_0 \xrightarrow{\mathfrak{x}^*}{}^* q_l$.

The extensions of the DFSM transition δ and output function ω to input traces are defined by

$$\omega^* : Q \times \Sigma_I^* \to \Sigma_O^* \tag{2.1}$$

$$\omega^*(q, <>) : <> \tag{2.2}$$

$$\omega^*(q, \mathfrak{x} \cdot \mathfrak{x}^*) : \omega(q, \mathfrak{x}) \cdot \omega^*(\delta(q, \mathfrak{x}), \mathfrak{x}^*) \tag{2.3}$$

$$\delta^* : Q \times \Sigma_I^* \to Q^* \tag{2.4}$$
$$\delta^*(q, <>) = q \tag{2.5}$$
$$\delta^*(q, \mathfrak{x} \cdot \mathfrak{x}^*) = q \cdot \delta^*(\delta(q, \mathfrak{x}), \mathfrak{x}^*). \tag{2.6}$$

Furthermore, let $\bar{\delta} : Q \times \Sigma_I^* \to Q$ be the function mapping a state q and an input trace $\mathfrak{x}^*$ to the target state that is reached when applying $\mathfrak{x}^*$.

$$\bar{\delta} : Q \times \Sigma_I^* \to Q \tag{2.7}$$
$$\bar{\delta}(q, <>) : q \tag{2.8}$$
$$\bar{\delta}(q, \mathfrak{x}.\mathfrak{x}^*) : \bar{\delta}(\delta(q, \mathfrak{x}), \mathfrak{x}^*) \tag{2.9}$$

Definition 4 (Language of States and DFSMs)**.** The *language* $\mathcal{L}(q)$ of a DFSM state q is the set of all I/O traces that originate from this state: $\mathcal{L}(q) \triangleq \{\mathfrak{l}^* | q \xrightarrow{\mathfrak{l}^*}{}^* q'\}$. The language of a DFSM M is the language of the initial state of M $\mathcal{L}(M) \triangleq \mathcal{L}(\underline{q})$.

Definition 5 (FSM State Equivalence)**.** Two DFSM states q_1 and q_2 are *I/O-equivalent*, or simply *equivalent*, $q_1 \sim q_2$, if both states share the same language: $q_1 \sim q_2 \iff \mathcal{L}(q_1) = \mathcal{L}(q_2)$. Intuitively, this means that two I/O-equivalent states produce the same output traces for every possible input trace.

Definition 6 (FSM Equivalence)**.** Two DFSMs M_1 and M_2 are I/O-equivalent $M_1 \sim M_2$ if their initial states $\underline{q}_1$ and $\underline{q}_2$ are I/O-equivalent: $M_1 \sim M_2 \iff \underline{q}_1 \sim \underline{q}_2$.

Definition 7 (Minimal Deterministic Finite-State Machine)**.** A DFSM $M = (Q, \underline{q}, \Sigma_I, \Sigma_O, \delta, \omega)$ is *minimal* if no two states from Q are I/O-equivalent:

$$M \text{ is } \textit{minimal} \iff \forall q_i, q_j \in Q : q_i \sim q_j \Rightarrow q_i = q_j. \tag{2.10}$$

Every non-minimal DFSM M can be minimised to an equivalent minimal DFSM M'. [Gil62] proposes an algorithm that is based on P_k-tables. We shortly recall the functionality of this algorithm.

Definition 8 (k-Equivalence)**.** The *k-language* $\mathcal{L}_{\leq k}(q)$ of a DFSM state q is the set of all I/O traces of length up to k that originate from this state: $\mathcal{L}(q) \triangleq \{\mathfrak{l}^* | q \xrightarrow{\mathfrak{l}^*}{}^* q', \text{length}(\mathfrak{l}^*) \leq k\}$.

Two DFSM states q_1 and q_2 are *k-equivalent*, $q_1 \sim_k q_2$, if both states share the same k-language: $q_1 \sim_k q_2 \iff \mathcal{L}_{\leq k}(q_1) = \mathcal{L}_{\leq k}(q_2)$. Intuitively, this means that two k-equivalent states produce the same output traces for every possible input trace of length up to k. Two states that are not k-equivalent are said to be *k-distinguishable*.

k-equivalence is an equivalence relation and therefore induces a state equivalence partitioning $Q/_{\sim_k}$. A P_k-table is a special form of the transition table. This table has an additional column mapping a state to its k-equivalence class. Let this column be denoted by function $\mathcal{C}_k : Q \to Q/_{\sim_k}$

mapping a state $q \in Q$ to its k-equivalence class $c \in Q/_{\sim_k}$. Because the P_k table is completely defined by the original transition table and $\mathcal{C}_k$, we use the terms P_k-table and $\mathcal{C}_k$ interchangeably.

Note that the P_1-table can be calculated from the ω-column of the transition table. Two states q_i and q_j are in the same 1-equivalence class iff they produce the same outputs for all inputs $\mathfrak{x} \in \Sigma_I$ from the input alphabet: i.e., if they share the same entries in the ω-column of the transition table.

Given the P_k-table, the P_{k+1} table can be calculated as follows: Two states q_i and q_j that are k-distinguishable are also $k+1$-distinguishable. Therefore, $\mathcal{C}_{k+1}(q_i) \neq \mathcal{C}_{k+1}(q_j)$ must hold. This means that only pairs of states from the same k-equivalence class have to be investigated. Two states q_i and q_j that are k-equivalent ($\mathcal{C}_k(q_i) = \mathcal{C}_k(q_j)$), will be $k+1$-distinguishable ($\mathcal{C}_{k+1}(q_i) \neq \mathcal{C}_{k+1}(q_j)$), iff at least one input symbol $\mathfrak{x} \in \Sigma_I$ exists that leads from q_i and q_j to target states that are themselves k-distinguishable.

$$\begin{aligned} \forall q_i, q_j \in Q : &\mathcal{C}_k(q_i) = \mathcal{C}_k(q_j) \Rightarrow \\ &((\exists \mathfrak{x} \in \Sigma_I : \mathcal{C}_k(\delta(q_i, \mathfrak{x})) \neq \mathcal{C}_k(\delta(q_j, \mathfrak{x}))) \iff \mathcal{C}_{k+1}(q_i) \neq \mathcal{C}_{k+1}(q_j)) \end{aligned} \tag{2.11}$$

Note that Equation 2.11 implies that the state partitioning $Q/_{\sim_i}$ becomes more fine-grained with increasing i.

The minimisation of a DFSM $M = (Q, \mathfrak{q}, \Sigma_I, \Sigma_O, \delta, \omega)$ is performed by successively calculating P_{k+1}-tables from the predecessor P_k-table until a fixed point is reached, meaning $P_{k+1} = P_k$. Such a fixed point implies that all subsequent P_i-tables for $i \geq k$ will be the same as well. In particular, this proves, for all pairs of k-equivalent states $q_i \sim_k q_j$, that these states are I/O-equivalent as well $q_i \sim q_j$. The existence of such a fixed point is ensured by the fact that (1) Q is finite and (2) the state partitionings $Q/_{\sim_i}$ become more fine-grained with increasing i. At the latest, a fixed point is reached when the partitioning is composed of singletons only.

The last P_k-table defines the state equivalence partitioning $Q/_{\sim}$. The minimised DFSM $M_m = (Q_m, \mathfrak{q}_m, \Sigma_I, \Sigma_O, \delta_m, \omega_m)$ is obtained by setting $Q_m = Q/_{\sim}$, $\mathfrak{q}_m = \mathcal{C}_k(\mathfrak{q})$,
$\delta_m = \{(\mathcal{C}_k(q), \mathfrak{x}) \mapsto \mathcal{C}_k(q') | q, q' \in Q, \mathfrak{x} \in \Sigma_I, \delta(q, \mathfrak{x}) = q'\}$ and
$\omega_m = \{(\mathcal{C}_k(q), \mathfrak{x}) \mapsto \mathfrak{y} | q \in Q, \mathfrak{x} \in \Sigma_I, \omega(q, \mathfrak{x}) = \mathfrak{y}\}$. Algorithm 1 summarises the algorithm for the DFSM minimisation, as described above.

Algorithm 1 DFSM Minimisation

Input: $M = (Q, \mathfrak{q}, \Sigma_I, \Sigma_O, \delta, \omega)$ as DFSM to be minimised
Output: $(M_m = (Q_m, \mathfrak{q}_m, \Sigma_I, \Sigma_O, \delta_m, \omega_m), \{\mathcal{C}_1, \ldots, \mathcal{C}_k\})$ a minimal DFSM and the P_k-tables

```
function DFSMMinimisation
    C_1 := calculateP1Table(M)
    k := 0
    repeat
        k := k + 1
        C_{k+1} := calculateNextPkTable(M, C_k)        ▷ using Equation 2.11
    until C_k = C_{k+1}
    return ((Q_m, q_m, Σ_I, Σ_O, δ_m, ω_m), {C_1, ..., C_k})   ▷ Q_m, q_m, δ_m, ω_m as described above
end function
```

Example 7. Consider M_1 defined in Figure 2.8. We will see that B and C are I/O-equivalent $B \sim C$. Thus, M_1 is not minimal. This claim can be proven by applying the minimisation

	$\mathcal{C}_1$	δ			ω		
		a	b	c	a	b	c
A	1	A_1	B_1	C_1	1	2	2
B	1	C_1	D_1	C_1	1	2	2
C	1	B_1	D_1	C_1	1	2	2
D	1	B_1	E_2	C_1	1	2	2
E	2	B_1	E_2	A_1	1	0	0

(a) P_1-table

	$\mathcal{C}_2$	δ		
		a	b	c
A	3	A_3	B_3	C_3
B	3	C_3	D_4	C_3
C	3	B_3	D_4	C_3
D	4	B_3	E_2	C_3
E	2	B_3	E_2	A_3

(b) P_2-table

	$\mathcal{C}_3$	δ		
		a	b	c
A	5	A_5	B_6	C_6
B	6	C_6	D_4	C_6
C	6	B_6	D_4	C_6
D	4	B_6	E_2	C_6
E	2	B_6	E_2	A_5

(c) P_3-table

	$\mathcal{C}_4$	δ		
		a	b	c
A	5	A_5	B_6	C_6
B	6	C_6	D_4	C_6
C	6	B_6	D_4	C_6
D	4	B_6	E_2	C_6
E	2	B_6	E_2	A_5

(d) P_4-table

Table 2.2: P_k-tables for DFSM M_1

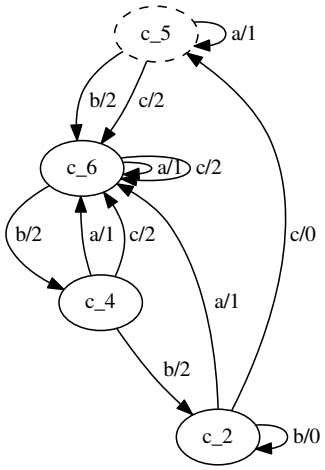
algorithm (Algorithm 1) to M_1. Table 2.2 shows the P_k-tables for M_1. The column labelled $\mathcal{C}_1$ in the P_1 table maps all states $A, \ldots, E$ to their 1-equivalence class from $Q/_{\sim_1}$. There are two 1-equivalence classes $c_1 = \{A, B, C, D\}$ and $c_2 = \{E\}$. The states A, B, C and D are all 1-equivalent, while E is 1-distinguishable from A for example. The two 1-equivalence classes result from the fact that the outputs (see the column labelled ω in the P_1-table) are identical for $\{A, B, C, D\}$ and only E differs in the outputs. In the P_1-table, we subscripted all target states in the δ-column with the value of $\mathcal{C}_1$.

To calculate the P_2-table, we use the rule described in Equation 2.11. All pairs of states in c_1 remain in the same equivalence class, if for all inputs the target states are in the same 1-equivalence class. Otherwise, the two states are 2-distinguishable and therefore have to be spread to different 2-equivalence classes. Intuitively, this step can be performed by taking a close look at the subscripts used in the δ-column of the P_1-table. The subscripts for A, B and C are (1,1,1) in the P_1-table. These three states are 2-equivalent. However, the subscripts for D are (1,2,1). Thus, D is 2-distinguishable from A, B and C. This yields two new 2-equivalence classes $c_3 = \{A, B, C\}$ and $c_4 = \{D\}$. The 1-equivalence class c_2 is already singleton and therefore cannot be further partitioned.

Note that we dropped the ω-columns for the P_2 and all subsequent P_k-tables because these columns are no longer needed for the calculation of the successive tables. A close look at the subscripts in the δ-columns of the P_2-table reveals that A is 3-distinguishable from B and C, which in turn are 3-equivalent. Hence, the 2-equivalence class c_3 is split to the 3-equivalence classes $c_5 = \{A\}$ and $c_6 = \{B, C\}$.

The P_4-table equals the P_3-table because B and C are 4-equivalent. This causes Algorithm 1 to terminate. The fact that a fixed point is reached ($\mathcal{C}_3 = \mathcal{C}_4$) proves that B and C are not only 1, 2, 3, and 4-equivalent but also i-equivalent for arbitrary $i \in \mathbb{N}_+$. Therefore, both states are I/O-equivalent $B \sim C$.

The minimal DFSM M_{1m} that is equivalent to M_1 is shown in Figure 2.9. M_{1m} is obtained by



(a) State Chart

	δ			ω		
	a	b	c	a	b	c
c_5	c_5	c_6	c_6	1	2	2
c_6	c_6	c_4	c_6	1	2	2
c_4	c_6	c_2	c_6	1	2	2
c_2	c_6	c_2	c_5	1	0	0

(b) Transition Table

Figure 2.9: Minimal DFSM M_{1m} that is Equivalent to M_1

replacing all states in M_1 by their state equivalence class, as determined by $\mathcal{C}_3$. This results in a new transition table in which the redundant line for state-equivalence class c_6 has been dropped. The transition table and the state chart for M_{1m} are shown in Figure 2.9.

2.4.2 Complete Testing Theories

Definition 9 (Correctness, Conformance, Equivalence)**.** Given a specification or test model defined as a DFSM $M = (Q, \mathfrak{q}, \Sigma_I, \Sigma_O, \delta, \omega)$ and an *implementation* defined as a DFSM $\mathfrak{I} = (Q_\mathfrak{I}, \mathfrak{q}_\mathfrak{I}, \Sigma_I, \Sigma_O, \delta_\mathfrak{I}, \omega_\mathfrak{I})$. The implementation is considered *correct*, if it is *conforming* to the specification $\mathfrak{I} \leq M$ according to a conformance relation $\leq$. Typically, two types of conformance relations exist: *refinement* and *equivalence*. Refinement means that the language defined by $\mathfrak{I}$ is a subset of the language of the specification $\mathcal{L}(\mathfrak{I}) \subseteq \mathcal{L}(M)$. In the remainder we use *I/O-equivalence* $\sim$ as a conformance relation. Thus, the languages of the implementation and the specification must be identical $\mathcal{L}(\mathfrak{I}) = \mathcal{L}(M)$.

The following paragraphs introduce some complete testing theories. Complete testing theories, in contrast to test heuristics, are guaranteed to reveal certain kinds of faults. To achieve this, a formal model of faults is needed. We first introduce the fault model for DFSMs and then present two complete testing theories: namely, the W-method and the Wp-method.

Definition 10 (Test Cases, Oracles, Pass and Fail)**.** For a specification/test model given as a DFSM $M = (Q, \mathfrak{q}, \Sigma_I, \Sigma_O, \delta, \omega)$, let **TS** denote a *test suite*: i.e., a finite set of *test cases*. A *test case* $t \in$ **TS** is an input trace. The application of a test case to the specification M in the initial state $\mathfrak{q}$ will produce an observable output trace $\mathfrak{y}^* = \omega^*(\mathfrak{q}, t)$. This output trace describes the

expected behaviour of the specification when t is applied. It will be called a *test oracle* in the remainder, and for convenience the test oracle will be denoted by $O(t) \triangleq \omega^*(\underline{q}, t)$.

Given an *implementation* defined as a DFSM $\mathfrak{I} = (Q_\mathfrak{I}, \underline{q}_\mathfrak{I}, \Sigma_I, \Sigma_O, \delta_\mathfrak{I}, \omega_\mathfrak{I})$, the implementation may *pass* or *fail* a test case t. The implementation passes the test case (denoted by $\mathfrak{I}$ pass t), if $\omega^*_\mathfrak{I}(\underline{q}_\mathfrak{I}, t) = O(t)$ and the implementation fails the test case otherwise (denoted by $\mathfrak{I}$ fail t). The implementation passes the test suite ($\mathfrak{I}$ pass **TS**) if it passes all test cases and it fails the test suite ($\mathfrak{I}$ fail **TS**) if it fails at least one test case.

Definition 11 (Fault Model for DFSMs)**.** The common fault model for DFSMs assumes that the test model is given as a DFSM M. An implementation is assumed to show the I/O behaviour of another DFSM $\mathfrak{I}$. Thus, the fault model requires that the implementation shows deterministic behaviour. Furthermore, it is assumed that specification M and $\mathfrak{I}$ use the same input and output alphabets and that the implementation provides a correctly implemented reset operation. This reset operation allows for the application of input traces to the initial state of the implementation.

Faults of the implementation can be categorised as two types: *transfer* and *output* faults. A transfer fault can be introduced by changing the target of a transition, and an output fault is introduced by changing the output of a transition.

The *fault model* $\mathcal{F}(M, \leq, \mathcal{D}(m))$ for DFSMs is comprised of the specification M, a conformance relation $\leq$ (we will use I/O-equivalence $\sim$ in this work) and a *fault domain*, which is denoted by $\mathcal{D}(m)$. The *fault domain* is the set of all possible DFSMs M' that use the same input and output alphabet as M, have a reset function and have at most m states after DFSM minimisation. The members of $\mathcal{D}(m)$ include the automaton M itself and all variations of M that result from multiple applications of the two fault operators (*transfer* and *output fault*) and have at most m states in the minimised DFSM.

Definition 12 (Completeness of Test Suites and Testing Theories for FSMs)**.** A test suite **TS** is said to be complete with respect to a fault model, if it is *sound* and *exhaustive*.

Soundness means that every correct implementation $\mathfrak{I}$ passes the test suite.

$$\mathfrak{I} \sim M \Rightarrow \mathfrak{I} \text{ pass } \mathbf{TS} \tag{2.12}$$

A test suite **TS** is exhaustive with respect to fault domain $\mathcal{D}(m)$ if every member of the fault domain that is not I/O-equivalent to the specification M fails the test suite.

$$\forall \mathfrak{I} \in \mathcal{D}(m) : \mathfrak{I} \not\sim M \Rightarrow \mathfrak{I} \text{ fail } \mathbf{TS} \tag{2.13}$$

A *complete testing theory* is a mapping from a fault model $\mathcal{F}(M, \sim, \mathcal{D}(m))$ to a test suite **TS** that is complete with respect to this fault model.

For DFSMs, many complete testing theories exist. We present the W-method and the Wp-method below. Both methods require the calculation of the transition cover and the characterisation set.

Definition 13 (Transition Cover)**.** Given a minimal DFSM M, the transition cover TC is a set of input traces that contains an input trace $\mathfrak{x}^* = \mathfrak{x}_1 \dots \mathfrak{x}_n$ for every transition $t = q_i \xrightarrow{\mathfrak{x}_n/\mathfrak{y}} q_j$ of M such that this transition is taken when the input trace is applied to the initial state: $\mathfrak{q} \xrightarrow{\mathfrak{x}\dots\mathfrak{x}_{n-1}}{}^* q_i \wedge q_i \xrightarrow{\mathfrak{x}_n/\mathfrak{y}} q_j$.

Note that, for technical reasons regarding the W-method, the empty trace <> is considered to be an element of TC.

Definition 14 (Characterisation Set)**.** Given a minimal DFSM M, the characterisation set CS is a set of input traces with the property that every pair of states $q_i, q_j : q_i \neq q_j$ can be distinguished by a member of CS (i.e., the output of M is different when an input trace from CS is applied): $\forall q_i, q_j : q_i \neq q_j \implies \exists \mathfrak{x}^* \in \text{CS} : \omega^*(q_i, \mathfrak{x}^*) \neq \omega^*(q_j, \mathfrak{x}^*)$.

The characterisation set of a DFSM can be calculated by using an approach presented in [Gil62, Section 4.10]. The algorithm presented there needs a DFSM M and the P_k-tables of this DFSM, which might have been generated using the minimisation algorithm shown in Algorithm 1. The algorithm for the calculation of the characterisation set is shown in Algorithm 2.

Algorithm 2 Algorithm for the Calculation of the Characterisation Set CS

Input: $M = (Q, \mathfrak{q}, \Sigma_I, \Sigma_O, \delta, \omega)$ a DFSM
Input: $\{\mathcal{C}_1, \dots, \mathcal{C}_k\}$ the P_k tables of M
Output: CS the characterisation set of M

```
function CHARACTERISATIONSET
    CS := ∅
    G = {{elem(C_k^{-1}[c]) | c ∈ Q/∼_k}}        ▷ G is a set of sets of states, every set of states
represents states that are not yet distinguishable by the characterisation set calculated so far
    while G ≠ ∅ do
        g := elem(G)                              ▷ selects a pair of states that are not yet distinguishable
        G := G \ {g}
        (q1, q2) := elem({(qi, qj) | qi, qj ∈ g, qi ≠ qj})
        σ := DISTINGUISHINGSEQUENCE(M, {C1, ..., Ck}, q1, q2)        ▷ σ distinguishes (q1, q2)
        CS := CS ∪ {σ}
        {g1, ..., gl} := PARTITION(g, σ)    ▷ partition set g by the output traces triggered by σ
        for each gi ∈ {g1, ..., gl} do
            if |gi|>1 then                        ▷ partitions with more than one element
                G := G ∪ {gi}                     ▷ ... still have to be distinguished
            end if
        end for
    end while
    return CS
end function
```

Note that M may or may not be minimal already. The P_k-tables of M are again defined by functions $\{\mathcal{C}_1, \dots, \mathcal{C}_k\}$ mapping states to their i-equivalence class. The minimal DFSM $M_m = (Q_m, \mathfrak{q}_m, \Sigma_I, \Sigma_O, \delta_m, \omega_m)$ that is equivalent to M is implicitly determined by M and $\{\mathcal{C}_1, \dots, \mathcal{C}_k\}$. Therefore, this minimal DFSM is not needed as input for Algorithm 2.

The characterisation set is calculated by iterative partitioning of the set of distinguishable states. The set of sets of states G contains sets of states that are not yet distinguishable by input traces

in CS. Initially, G is a singleton containing the set of all distinguishable states in M. The set of distinguishable states is defined by $\{\texttt{elem}(\mathcal{C}_k^{-1}[c])|c \in Q/_{\sim_k}\}$. `elem`() denotes an operator that returns an arbitrary element of a non-empty set. For every state-equivalence class $c \in Q/_{\sim_k}$ one representative is chosen by using `elem`() on the inverse image $\mathcal{C}_k^{-1}[c]$ of function $\mathcal{C}_k$.

Then, successively, an element $g \in G$ is taken from G, and from the states in g two states are selected. By construction of G, these states are distinguishable. A distinguishing sequence σ for these two states is calculated using Algorithm 3. This input trace distinguishes at least q_1 and q_2 from each other. σ is added to the characterisation set CS. Then, operation `partition` partitions the states in g by their response to σ. The response of a state q to some input trace σ is the output trace that is produced when the input trace is applied to the state $\omega^*(q, \sigma)$. The response induces an equivalence relation $\sim_\sigma$. Two states are $\sim_\sigma$-equivalent iff they produce the same output when σ is applied. `partition` calculates the partitioning $g/_{\sim_\sigma}$. Every partition $g_i \in g/_{\sim_\sigma}$ which is not singleton represents a set of states that are not yet distinguishable. Therefore, g_i is added to G. The algorithm must finally terminate. This is reasoned by the fact that G is a partitioning of Q (with singletons dropped). Because the partitioning is refined in every iteration of the loop, G will finally be empty when all partitions have become singleton.

Algorithm 3 Algorithm for the Calculation of a Distinguishing Sequence for Two States

Input: $M = (Q, \mathfrak{q}, \Sigma_I, \Sigma_O, \delta, \omega)$ a DFSM
Input: $\{\mathcal{C}_1, \ldots, \mathcal{C}_k\}$ the P_k tables of M
Input: q_1 a state from Q
Input: q_2 a state from Q
Output: σ an input traces that distinguishes q_1 from q_2

1: **function** DISTINGUISHINGSEQUENCE
2: $\sigma :=<>$
3: $l = \min(\{i|\mathcal{C}_i(q_1) \neq \mathcal{C}_i(q_2)\})$ ▷ q_1 and q_2 are l-distinguishable and $l-1$-equivalent
4: $q_i := q_1$ ▷ q_i and q_j start from q_1 and q_2
5: $q_j := q_2$
6: $\mathfrak{x} := \perp$
7: **for** $k = 1$ to $l - 1$ **do** ▷ q_i and q_j are $l-k$-equivalent
8: $(q_i, q_j, \mathfrak{x}) := \texttt{elem}(\{(q_i', q_j', \mathfrak{x})|\delta(q_i, \mathfrak{x}) = q_i' \wedge \delta(q_j, \mathfrak{x}) = q_j' \wedge \mathcal{C}_{l-k}(q_i') \neq \mathcal{C}_{l-k}(q_j')\})$
9: $\sigma := \sigma \cdot \mathfrak{x}$ ▷ select an $\mathfrak{x}$ to go from q_i and q_j to $l-k$-distinguishable states
10: **end for**
11: $\mathfrak{x} := \texttt{elem}(\{\mathfrak{x}|\omega(q_i, \mathfrak{x}) \neq \omega(q_j, \mathfrak{x})\})$ ▷ select an $\mathfrak{x}$ that distinguishes q_i and q_j
12: $\sigma := \sigma \cdot \mathfrak{x}$
13: **return** σ
14: **end function**

Algorithm 3 calculates a distinguishing sequence for two distinguishable DFSM states q_1 and q_2 of minimal length. First l is calculated. l is the minimal index, such that q_1 and q_2 are l-distinguishable ($\mathcal{C}_l(q_1) \neq \mathcal{C}_l(q_2)$). Because l is chosen minimally, q_1 and q_2 are $l-1$-equivalent. Thus, the shortest distinguishing sequence σ for q_1 and q_2 has exactly length $\text{length}(\sigma) = l$. This sequence is calculated by iteratively taking transitions in the loop (line 7). Variables q_i and q_j are used to keep track of the current states, starting at q_1 and q_2, respectively.

Note that the distinguishing sequence σ of length l must begin with an input symbol $\mathfrak{x}_1$ that causes the transition from q_1 and q_2 to $l-1$-distinguishable states q_1' and q_2'.[4] The construction

[4] **Proof:** Assume that q_1' and q_2' are not $l-1$-distinguishable, i.e., $l-1$-equivalent. Then, a sequence τ of length

of the P_k-tables ensures that such an input exists. Likewise, the next input symbol $\mathfrak{x}_2$ must cause transitions from q_1' and q_2' to $l-2$-distinguishable states q_1'' and q_2''. This argument can be repeatedly applied until states q_i and q_j are reached, which are 1-distinguishable. In this case, an input symbol $\mathfrak{x}_l$ is selected that produces different outputs for q_i and q_j. See line 11. $\sigma = \mathfrak{x}_1 \cdot \mathfrak{x}_2 \cdot \ldots \cdot \mathfrak{x}_l$ is a minimal distinguishing sequence for q_1 and q_2.

Example 8. Consider DFSM M_1, as shown in Figure 2.8. The P_k-tables are shown in Table 2.2. We show the functionality of Algorithm 2 on M_1.

Initially, G is set to $G := \{\{A, B, D, E\}\}$. Note that C is missing in G because it is equivalent to B and the algorithm of `characterisationSet` considers only one representative for each equivalence class $c \in Q/_{\sim_k}$.

Suppose that B and D are selected to be distinguished in the first loop iteration. `distinguishingSequence` will be called for this pair of states to obtain a distinguishing sequence σ_1. In this case, $l = 2$ because 2 is the minimal index that fulfills $\mathcal{C}_2(B) \neq \mathcal{C}_2(D)$. The first input symbol for σ_1 can be determined by a close look at the P_{l-1}-table: i.e., the P_1-table. The first input symbol is b, because this is the only sub-column in the δ-column of the P_1-table, where the subscripts differ for B and D. This means that b is the only input symbol fulfilling $\delta(B, b) = D \wedge \delta(D, b) = E \wedge \mathcal{C}_1(D) \neq \mathcal{C}_1(E)$. The loop in `distinguishingSequence` terminates after the first iteration. The second input symbol can be obtained by comparing the outputs for the target states of the last iteration D and E. We choose c as the last element in σ. Note that b is possible as well, because D and E differ for both input symbols in the ω-column of the transition table. Thus, $\sigma_1 = b \cdot c$ is a distinguishing sequence for B and D and becomes part of the characterisation set CS.

Now the set of states $g = \{A, B, D, E\}$ is partitioned by $\sim_{\sigma_1}$. The partitioning is $g/_{\sim_{\sigma_1}} = \{\{A, B\}, \{D\}, \{E\}\}$. The response of the states A, B, D and E to σ_1 is $2 \cdot 2, 2 \cdot 2, 2 \cdot 0$ and $0 \cdot 0$, respectively. Both singletons from $g/_{\sim_{\sigma_1}}$ are dropped. $\{A, B\}$ is added to G and becomes the only remaining element in G. Thus, A and B are the pair of states in the second loop iteration of `characterisationSet`.

`distinguishingSequence` for A and B yields $l = 3$. To obtain the first input symbol for σ_2, have a look at the $P_{l-1=2}$-table. The first input symbol for σ_2 is b (because of the different subscripts for A and B in the P_2-table). q_i and q_j are updated to B and D for the second loop iteration. The second input symbol is b (because of the different subscripts for B and D in the P_1-table). q_i and q_j are updated to D and E. Finally, b might be selected as last element of σ_2 because D and E produce different outputs for b (though c would be possible as well). $\sigma_2 = b \cdot b \cdot b$ is added to the characterisation set.

Afterwards, `characterisationSet` will terminate because $g = \{A, B\}$ will be partitioned to $g/_{\sim_{\sigma_2}} = \{\{A\}, \{B\}\}$. Therefore, no more states need to be distinguished and the characterisation set is $\mathrm{CS} = \{\sigma_1, \sigma_2\}$.

Note that the characterisation set calculated by `characterisationSet` is not necessarily a minimal characterisation set. For example, $\mathrm{CS}' = \{\sigma_2\}$ is a characterisation set for M_1 as well. CS' would have been obtained if we had chosen A and B in the first loop iteration of `characterisationSet`. If, instead, we first calculated a distinguishing sequence for B and D but chose b as the second element in σ_1, the characterisation set would be $\mathrm{CS}'' = \{b \cdot b, b \cdot b \cdot b\}$. Because the first sequence in CS'' is a prefix of the second one, this sequence can be dropped,

$k \geq l$ would be necessary to distinguish q_1' and q_2'. σ, which is $\mathfrak{x}_1$ concatenated with τ, thus has a length $\mathrm{length}(\sigma) > l$. This contradicts $\mathrm{length}(\sigma) = l$.

as every pair of states distinguished by a prefix $\mathfrak{x}_p^*$ will also be distinguished by any trace $\mathfrak{x}_p^* \cdot \mathfrak{x}_e^*$. Therefore, a minimal characterisation set CS′ could be obtained by dropping prefixes in CS″.

The example above demonstrates that Algorithm 2 does not necessarily generate a minimal characterisation set. Sometimes the generated characterisation set can be minimised by dropping prefixes, although this is not always possible. Section 4.3.5.3 describes an algorithm to calculate a minimal characterisation set.

W-method

The W-method is a complete testing theory that generates a test suite $\mathcal{W}$ for a given fault model $\mathcal{F}(M, \sim, \mathcal{D}(m))$. The set of test cases for the W-method is calculated by the following formula:

$$\mathcal{W} = \mathrm{TC} \cdot \bigcup_{i=0}^{m-n} X^i \cdot \mathrm{CS}. \tag{2.14}$$

m is an upper bound for the number of states implementation $\mathfrak{I}$ is assumed to have. Equation 2.14 can be interpreted as follows: Each input trace from the transition cover is appended by every input trace of length up to $m - n$. This assures that every state in $\mathfrak{I}$ is reached at least once by the elements of $\mathrm{TC} \cdot \bigcup_{i=0}^{m-n} X^i$. Finally, the target states that are reached after application of these inputs must be identified. Therefore, the sequences from CS are used.

A formal proof of the completeness of the W-method can be found in [Cho78].

Wp-method

An improvement of the W-method is the Wp-method proposed in [FvBK+91] and generalised in [LvBP94] to non-deterministic FSMs. The Wp-method has the advantage that the number of test cases is reduced. This is done by using a modified version of the W-Method in a first step to show that every state in the implementation is correct before applying a second step that tries to verify the correctness of all transitions in the implementation.

For the application of the Wp method, the state cover $\mathrm{SC} \subseteq X_I^*$ is needed. The state cover is a set of input traces that contains an input trace $\mathfrak{x}^*$ for every state $q_i \in Q$ of M such that this state is reached when the input trace is applied to the initial state: $\mathfrak{q} \stackrel{\mathfrak{x}^*}{\Longrightarrow} q_i$. The transition cover can be calculated from the state cover by setting:[5]

$$\mathrm{TC} = \mathrm{SC} \cdot (<> \cup \Sigma_I). \tag{2.15}$$

The set of input traces $\mathrm{R} \subseteq X_I^*$

[5] As mentioned above, the empty trace <> is considered part of TC. For the Wp-method this empty trace is not needed (see definition of R) and can be dropped. In this case, $\mathrm{TC} = \mathrm{SC} \cdot \Sigma_I$ holds.

$$\mathrm{R} = \mathrm{TC} \setminus \mathrm{SC} \tag{2.16}$$

contains all traces that are contained in TC but not in SC.

For the reduction of test cases, the Wp-method uses the fact that every single state q_i can be distinguished from all other states $q_j \in Q$ by a subset of input traces $\mathrm{W}_i \subseteq \mathrm{CS}$ from the characterisation set. W_i is called a state-identification set for q_i. For every other state q_j, there is at least one input trace $\mathfrak{x}^* \in \mathrm{W}_i$ that distinguishes q_i and q_j: $\forall q_i, q_j \in Q : q_i \neq q_j \implies \exists \mathfrak{x}^* \in \mathrm{W}_i : \omega^*(q_i, \mathfrak{x}^*) \neq \omega^*(q_j, \mathfrak{x}^*)$. Intuitively, these sets $\{\mathrm{W}_1, \ldots, \mathrm{W}_n\}$ can be calculated by dropping sequences from CS that are not needed to distinguish q_i from other states. Section 4.3.5.3 describes an algorithm for the calculation of minimal state-identification sets.

When applying arbitrary input sequences $V \subseteq X_I^*$ the target state of the application of such an input sequence $v \in V$ can be verified by applying the state-identification set W_i of the expected target state $q_i = \bar{\delta}(\underline{\mathfrak{q}}, v)$. Thus, we define the operator $\oplus$ as follows:

$$V \oplus \{\mathrm{W}_1, \ldots, \mathrm{W}_n\} = \{v \cdot w | v \in V, \underline{\mathfrak{q}} \overset{v}{\Longrightarrow} q_i, w \in \mathrm{W}_i\}. \tag{2.17}$$

Finally, the test cases of the Wp-method are calculated by the following formula:

$$\mathcal{W}_p = \mathcal{W}_1 \cup \mathcal{W}_2 \tag{2.18}$$

$$\mathcal{W}_1 = \mathrm{SC} \cdot \bigcup_{i=0}^{m-n} X^i \cdot \mathrm{CS} \tag{2.19}$$

$$\mathcal{W}_2 = (\mathrm{R} \cdot X^{m-n}) \oplus \{\mathrm{W}_1, \ldots, \mathrm{W}_n\}. \tag{2.20}$$

The intuition of the Wp-method is as follows: If an implementation $\mathfrak{I}$ successfully passes the tests of $\mathcal{W}_1$, it is guaranteed, that every state in $\mathfrak{I}$ is CS equivalent to exactly one state of S. $\mathrm{SC} \cdot \bigcup_{i=0}^{m-n} X^i$ is the state cover of implementation $\mathfrak{I}$, which is assumed to have no more than m states in the minimal FSM. By complementing the traces in $\mathcal{W}_1$ with CS, the target states can be uniquely determined. Thus, $\mathcal{W}_1$ tests the correctness of states. Additionally, the transitions have to be checked (to uncover transfer faults). Therefore, the transition cover of the implementation is needed. $\mathrm{TC} \cdot \bigcup_{i=0}^{m-n} X^i$ is the transition cover of the implementation. Since we have already checked many input sequences with $\mathcal{W}_1$, we need only to consider the sequences that have not already been checked. These are $\mathrm{R} \cdot X^{m-n}$. Because the states have already been verified by $\mathcal{W}_1$, we only need to verify that the target state of a transition is correct for $\mathcal{W}_2$; therefore, we do not apply the complete characterisation set CS, but only the state-identification set of the expected target state.

A formal proof of the completeness property of the Wp-method can be found in [FvBK+91].

2.5 Complete Model-Based Tests by Equivalence Class Partition Testing

This section shows how the completeness of the W/Wp-method can be generalised to systems with infinite input domains but internal and output variables from finite domains. Examples of such systems include controllers that observe analogue inputs and make discrete control decisions, including speed-monitoring systems, airbag controllers, smoke detectors and thrust-reversal systems.

2.5.1 Reactive Input-Output State-Transition Systems

RIOSTSs are description means for state-based systems. RIOSTSs are a specialization of STSs.

Definition 15 (State-Transition System)**.** An STS $\mathcal{S}$ is a tuple $\mathcal{S} = (S, s_0, R)$. S is a set of states. $s_0 \in S$ is the initial state and $R \subseteq S \times S$ is a transition relation that relates pre and post states.

Definition 16 (Reactive Input-Output State-Transition System)**.** A RIOSTS $\mathcal{S}$ is a tuple $\mathcal{S} = (S, s_0, R, V, D)$. S, s_0 and R are defined as above. For RIOSTSs, the states $s \in S$ are valuation functions $s : V \to D$ that map variable symbols V to the domain D, which is the union of all variable domains. Every variable $x \in V$ has a domain D_x it is mapped to: $\forall x \in V : s(x) \in D_x$.

RIOSTSs extend STSs in the following way:

1. The variables can be partitioned to *input variables* I, *model variables* M and *output variables* O with $V = I \cup M \cup O$ and $I \cap M = M \cap O = I \cap O = \emptyset$.
2. The state space can be partitioned to *quiescent states* S_Q and *transient states* S_T with $S = S_Q \cup S_T$ and $S_Q \cap S_T = \emptyset$.

In the remainder, $s|_U$ $U \subseteq V$ denotes the restriction of a state s to variables in U only: i.e., $\mathrm{dom}(s|_U) = U \wedge \forall u \in U : s|_U(u) = s(u)$. The valuation of a concrete state s is denoted by $\{v_1 \mapsto c_1, v_2 \mapsto c_2, \ldots, v_n \mapsto c_n\}$ and describes that s maps variable v_i to concrete value c_i with $c_i \in D_{v_i}$. The fact that two states are connected via R, i.e., $(s_1, s_2) \in R$ is denoted by $R(s_1, s_2)$. Additionally, $R^*(s_0 s_1 \ldots s_k)$ denotes the fact that all states s_i with $i \in \{0, \ldots, k\}$ are connected via R:

$$R^*(s_0 s_1 \ldots s_k) \triangleq \forall i \in \{0, \ldots, k-1\} : R(s_i, s_{i+1}).$$

Usually, we use x to refer to an input variable, m to refer to a model variable and y to refer to an output variable. Let $\vec{x} = (x_1, \ldots, x_{|I|})$ be the vector of input variables. Similarly, $\vec{m}$ and $\vec{y}$ denote the vectors of model and output variables, respectively. D_I denotes the domain of the input variables $\vec{x}$ (i.e., $D_I = D_{x_1} \times \ldots \times D_{x_{|I|}}$, where $x_1, \ldots, x_{|I|} \in I$). A concrete input vector (i.e., a tuple with $|I|$ elements) is denoted by $\vec{c} \in D_I$. The concrete value for the i-th input variable $x_i \in I$ can be obtained from a concrete input vector $\vec{c}$ by selecting the i-th component of the vector. This will be denoted by $\vec{c}(x_i)$. Analogously, the domains of model and output variables are denoted by D_M and D_O, respectively. Concrete value vectors for variables from M and O will be written as $\vec{d} \in D_M$ and $\vec{e} \in D_O$.

Note that the state space S of a RIOSTS may be infinite. Therefore, the state space and transition relation R are not explicitly enumerated. If not otherwise stated, we assume in the remainder that state space S is the set of all possible valuations $s : V \to D$. The transition relation R is described by a first-order logic predicate $\mathcal{R}$ over variables from $V \cup V'$. Because R relates pre-states s and post-states s', we need to use a set of primed variables V' to denote variable values in the post state. Unprimed variables from V denote variable values in the pre state. $\mathcal{R}$ implicitly describes the potentially infinite transition relation as follows: Pre- and post-state s and s' are related by R iff $\mathcal{R}$ yields true when all occurrences of unprimed variables $v \in V$ are replaced by their valuation $s(v)$ in the pre-state and all primed variables $v' \in V'$ are replaced by their valuation $s'(v')$ in the post-state. We denote the replacement of variables $v \in V$ by their valuation $s(v)$ in an expression $\mathcal{E}$ by $\mathcal{E}[s(v)/v \in V]$.

$$R(s, s') \iff \mathcal{R}[s(v)/v \in V, s'(v')/v' \in V'] \tag{2.21}$$

Definition 17 (Semantics of RIOSTSs)**.** The transition relation R of a RIOSTS has to fulfil the following conditions:

$$\forall s_q \in S_Q, s' \in S : (s_q, s') \in R \Rightarrow s_q|_{M \cup O} = s'|_{M \cup O} \tag{2.22}$$

In a quiescent state $s_q \in S_Q$, only inputs may be changed by the transition relation R while model and output variables' values must stay unchanged. This means that quiescent states are "stable" in the sense that the systems remains in this state as long as the inputs do not change, and as long as the inputs do not change, model variables and the observable output variables retain their values. A change of the system inputs might result in a quiescent state or a transient state.

$$\forall s_t \in S_T, s' \in S : (s_t, s') \in R \Rightarrow s_q|_I = s'|_I \tag{2.23}$$

In a transient state $s_t \in S_T$, only the internal model variables and output variables are allowed to change while the input variables have to retain their values.

The intuition behind quiescent and transient states is as follows. The system starts in an initial (quiescent state). Whenever the input values of the system change, these value changes might or might not trigger a recalculation of the internal-state variables. Whether a recalculation is needed depends on the post state of the input change. If the post state is a quiescent state, no internal-state update is performed. If, however, the post state is a transient state, the values for internal variables and output variables are recalculated and updated. In summary, quiescent states wait for a change of input variables and the possible transient states that are reached through such a change perform the recalculation of internal variable values.

In the context of testing, the distinction between quiescent and transient states is important, as only quiescent states are observable. It is assumed that transitions from transient states happen in zero-time such that system outputs are observable only in stable quiescent states. Furthermore, input stimuli are only applicable to the SUT when residing in a quiescent state, as prescribed by Equation 2.23.

Recall that we used the term *state* in the context of SysML/UML state machines (cf. Section 2.2.1) to denote statechart nodes. To avoid confusion, from now on, we will use the term *state* to refer to a system state of a RIOSTS. To refer to a statechart node of a SysML state machine, we will use the term *location*. We choose the term *location* because a state-machine state can be understood as a location the RIOSTS, which is described by the state machine, resides in.

Example 9. Consider the state machine from Figure 2.3.

The transition relation of the RIOSTS that is described by this UML state machine can be defined by a first-order logic predicate $\mathcal{R}$. In this predicate, we introduce the auxiliary variable l as an internal model variable to indicate the location that the state machine resides in.

$$
\begin{aligned}
\mathcal{R} =& \varphi_1 \vee \varphi_2 \vee \\
& \varphi_3 \vee \varphi_4 \vee \\
& \varphi_5 \vee \varphi_6 \\
\varphi_1 =& (l = \mathrm{L0} \wedge x <= 10) \wedge (l' = l \wedge z' = z) \\
\varphi_2 =& (l = \mathrm{L0} \wedge x > 10) \wedge (l' = \mathrm{L1} \wedge z' = 1 \wedge x' = x) \\
\varphi_3 =& (l = \mathrm{L1} \wedge x <= 20) \wedge (l' = l \wedge z' = z) \\
\varphi_4 =& (l = \mathrm{L1} \wedge x > 20) \wedge (l' = \mathrm{L2} \wedge z' = 2 \wedge x' = x) \\
\varphi_5 =& (l = \mathrm{L2} \wedge x \neq 0) \wedge (l' = l \wedge z' = z) \\
\varphi_6 =& (l = \mathrm{L2} \wedge x = 0) \wedge (l' = \mathrm{L0} \wedge z' = 0 \wedge x' = x)
\end{aligned}
$$

In this predicate, the subclauses φ_1 and φ_2 define possible transitions from location L0. The precondition of subclause φ_1 ($l = \mathrm{L0} \wedge x <= 10$) describes all quiescent states of the system that are residing in location L0 and that fulfil Equation 2.22. One concrete example of a quiescent state is $s_1 = \{x \mapsto 7, l \mapsto \mathrm{L0}, z \mapsto 0\}$. As long as input variable x fulfils $x <= 10$ in state s with $s(l) = 0$, the resulting post states s' are not allowed to change the values of internal variable l or output variable z: $(l' = l \wedge z' = z)$ means $s'(l) = s(l)$ and $s'(z) = s(z)$. The system state is stable in this case. The value of the input variable x, however, is allowed to change arbitrarily (no statement about x' in subclause φ_1). Thus, the post state of a transition described by φ_1 might be a transient state.

The precondition of subclause φ_2 ($l = \mathrm{L0} \wedge x > 10$) describes all transient states of the system that are residing in location L0 and that fulfil Equation 2.23. If the value of input variable x fulfils $x > 10$, the resulting post state must keep the value of input variable x; according to the semantics of entry actions in state machines, the output variable z is assigned to 1 and the resulting location is L1 ($l' = \mathrm{L1} \wedge z' = 1 \wedge x' = x$).

Analogously, φ_3 and φ_4 describe the quiescent and transient states residing in location L1, while φ_5 and φ_6 describe the quiescent and transient states residing in state machine location L2.

In the context of RIOSTSs, the term *trace* is used for finite or infinite sequences of states, usually denoted by $\tau = s_1 \dots s_n$. The terms *input trace* and *output trace* are used for finite or infinite sequences of inputs and outputs, respectively. S^* denotes the (infinite) set of all finite sequences of states while S^ω denotes the set of all infinite sequences of states.

Given a RIOSTS $\mathcal{S}$ with initial state s_0, a state s is said to be *reachable* iff s can be reached from the initial state by a finite sequence of intermediate states τ that are related according to the transition relation: $R^*(s_0 \tau s)$. The set of quiescent system states that are reachable from the initial state are called *reachable quiescent states* and are denoted by $\underline{S_Q}$. The set of *reachable transient states* is denoted by $\underline{S_T}$.

Definition 18 (Livelock-free RIOSTSs)**.** A RIOSTS is called *livelock-free* if there is no trace $s_q s_{t1} s_{t2} \ldots$ of infinite length starting in a reachable quiescent state s_q followed by infinitely many transient states $s_{t1} s_{t2} \ldots = \upsilon$:

$$\forall s_q \in \underline{S_Q} : \nexists \upsilon \in {S_T}^{\omega} : R^*(s_q \upsilon). \tag{2.24}$$

Definition 19 (Elimination of transient intermediate states)**.** Definition 17 can be further restricted without loss of generality if livelock-free RIOSTSs are considered:

$$\forall s_t \in \underline{S_T}, s' \in S : (s_t, s') \in R \Rightarrow s' \in S_Q. \tag{2.25}$$

Every post state of a reachable transient state has to be a quiescent state.

Note that every livelock-free RIOSTS $\mathcal{S} = (S, s_0, R, V, D)$ that does not fulfil Equation 2.25 can be transformed to a RIOSTS $\mathcal{S}' = (S', {s_0}', R', V', D')$ with the same observable I/O-behaviour as $\mathcal{S}$ while fulfilling Equation 2.25. To this end, let $S' = S$, ${s_0}' = s_0$, $V' = V$ and $D' = D$. R' is constructed using the following rule: All possible *reachability traces* (i.e., traces τ that are allowed according to the transition relation R: $R^*(\tau)$, of the form $\tau = s_{q0} s_{t1} \ldots s_{tk-1} s_{qk}$ with $s_{q0}, s_{qk} \in S_Q$, $s_{t1}, \ldots, s_{tk-1} \in S_T$) are abstracted by two tuples $(s_{q0}, s_{t1}) \in R'$ and $(s_{t1}, s_{qk}) \in R'$.

Although the intermediate transient states $s_{t2}, \ldots, s_{tk-1}$ are dropped from R', the observable behaviour of the resulting RIOSTS S' will be the same, because only quiescent states are observable. The construction ensures that, for every reachability trace $s_{q0} s_{t1} \ldots s_{tk-1} s_{qk}$ in R: $R^*(s_{q0} s_{t1} \ldots s_{tk-1} s_{qk})$, there will be an analogous trace $s_{q0} s_{t1} s_{qk}$ in R': $R'^*(s_{q0} s_{t1} s_{qk})$.

Example 10. Consider the RIOSTS from Example 9. The transition relation is described by $\mathcal{R}$. As described, the possible post states from a quiescent state s_{q0} that resides in state-machine location L0: $s_{q0}(l) = \text{L0} \wedge s_{q0}(x) <= 10$ are described by subclauses φ_1 and φ_2. Consider a possible transient post state s_{t1} with $s_{t1} = \{x \mapsto 25, l \mapsto \text{L0}, z \mapsto 0\}$. s_{t1} fulfils the precondition of subclause φ_2 ($l = \text{L0} \wedge x > 10$) and the only possible post state s_{t2} is $\{x \mapsto 25, l \mapsto \text{L1}, z \mapsto 1\}$. This state is again a transient state, as it fulfils the precondition of subclause φ_4 ($l = \text{L1} \wedge x > 20$). The only admissible post state of s_{t2} is $s_{q3} = \{x \mapsto 25, l \mapsto \text{L2}, z \mapsto 1\}$. This state is a quiescent state, because it fulfils the precondition of subclause φ_5 ($l = \text{L2} \wedge x \neq 0$).

To eliminate transient intermediate states, the transition relation R' of an I/O-equivalent RIOSTS $\mathcal{S}'$ will contain the two tuples (s_{q0}, s_{t1}) and (s_{t1}, s_{q3}). A close look at $\mathcal{R}$ reveals that only φ_2 allows for a post state that is transient. Hence, we can define a predicate $\mathcal{R}'$ that describes RIOSTS $\mathcal{S}'$ which fulfils Definition 19.

$$
\begin{aligned}
\mathcal{R}' =& \varphi_1 \vee \varphi_{2a} \vee \varphi_{2b} \vee \\
& \varphi_3 \vee \varphi_4 \vee \\
& \varphi_5 \vee \varphi_6 \\
\varphi_{2a} =& (l = \text{L0} \wedge x > 10 \wedge x <= 20) \wedge (l' = \text{L1} \wedge z' = 1 \wedge x' = x) \\
\varphi_{2b} =& (l = \text{L0} \wedge x > 20) \wedge (l' = \text{L2} \wedge z' = 2 \wedge x' = x)
\end{aligned}
$$

$\varphi_1, \varphi_3, \varphi_4, \varphi_5, \varphi_6$ as described in Example 9.

Note that the transient states are redundant as well and can be dropped without loss of information. Any two tuples $(s_{q_0}, s_{t1}) \in R'$ and $(s_{t1}, s_{q_k}) \in R'$ of the RIOSTS $\mathcal{S}'$ can be abstracted to one tuple $(s_{q_0}, s_{q_k}) \in \bar{R}$ in the transition relation $\bar{R}$ of an I/O-equivalent RIOSTS $\bar{\mathcal{S}}$. $\bar{\mathcal{S}}$ shows the same I/O-behaviour as the original RIOSTS $\mathcal{S}'$ with transition relation R', and the original information of R' can always be restored, because the transient state s_{t1} is identical to the state coinciding with s_{q_0} in $M \cup O$ and coinciding with s_{q_k} in the inputs I.

Definition 20 (Deterministic RIOSTS)**.** A RIOSTS $\mathcal{S}$ is called deterministic (and livelock-free) if, for every reachable transient state $s_t \in S_T$, there exists exactly one quiescent post state that is reachable from this state via transition relation R–assuming that all intermediate transient states have been eliminated before.

$$\forall s_t \in \underline{S_T} : \exists! s_q \in S_Q : R(s_t, s_q) \tag{2.26}$$

Example 11. The RIOSTS from Example 10 is deterministic, because each quiescent post state of a transient state is uniquely determined.

In the remainder of this work, we consider only deterministic RIOSTSs. The basics introduced below can, however, be generalised to non-deterministic RIOSTSs. In [HP16b], the authors have shown how the testing approach that we implemented in this work can be generalised and that this generalised approach is complete for non-deterministic systems as well.

2.5.2 Input Equivalence Class Partitioning

In the remainder of this work, the uniquely determined quiescent post state that results from applying input vector $\vec{c}$ to a quiescent state s of a deterministic RIOSTS is denoted by $s/\vec{c}$. The input variables $\vec{x}$ in the resulting quiescent post state evaluate to the input vector applied: $(s/\vec{c})(\vec{x}) = \vec{c}$. Restricting this state to output variables O yields the system's visible output behaviour. Thus, $(s/\vec{c})|_O$ is the visible output after applying concrete input $\vec{c}$ to a quiescent system state s.

When applying a sequence of input vectors (called input trace) $\iota = \vec{c}_1 \cdot \vec{c}_2 \cdot \ldots \cdot \vec{c}_n$ to a quiescent state s_0, the resulting sequence of quiescent post states (called state trace) $\tau = s_1 \cdot \ldots \cdot s_n$ is denoted by s_0/ι. The restriction of this state trace τ to output variables only $(s/\iota)|_O$ is the output trace that results from applying input trace ι to state s_0.

2.5.2.1 I/O-Equivalence of States

Two quiescent states s_1 and s_2 are *I/O-equivalent* $s_1 \sim_S s_2$ if both states produce the same output traces for every possible input trace ι:

$$s_1 \sim_S s_2 \Leftrightarrow \forall \iota \in D_I^* : (s_1/\iota)|_O = (s_2/\iota)|_O. \tag{2.27}$$

$\sim_S$ is an equivalence relation (i.e., a reflexive, symmetric and transitive relation), and therefore the quiescent state space can be partitioned to $S_Q/_{\sim_S}$. In the remainder, we will always be interested in the partitioning of the reachable quiescent states $\underline{S_Q}/_{\sim_S}$.

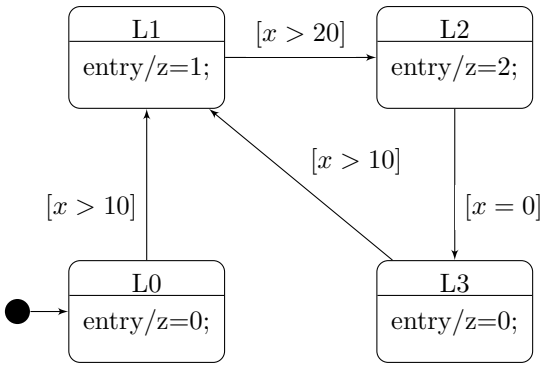


Figure 2.10: Example State Machine 2

Example 12. Consider the state machine from Figure 2.10. This state machine defines a RIOSTS $\mathcal{S}_2$ with a transition relation that can be described by predicate $\mathcal{R}_2$:

$$\begin{aligned}
\mathcal{R}_2 =& \varphi_1 \vee \varphi_{2a} \vee \varphi_{2b} \vee \\
& \varphi_3 \vee \varphi_4 \vee \\
& \varphi_5 \vee \varphi_6 \vee \\
& \varphi_7 \vee \varphi_{8a} \vee \varphi_{8b} \\
\varphi_1 =& (l = \text{L0} \wedge x <= 10) \wedge (l' = l \wedge z' = z) \\
\varphi_{2a} =& (l = \text{L0} \wedge x > 10 \wedge x <= 20) \wedge (l' = \text{L1} \wedge z' = 1 \wedge x' = x) \\
\varphi_{2b} =& (l = \text{L0} \wedge x > 20) \wedge (l' = \text{L2} \wedge z' = 2 \wedge x' = x) \\
\varphi_3 =& (l = \text{L1} \wedge x <= 20) \wedge (l' = l \wedge z' = z) \\
\varphi_4 =& (l = \text{L1} \wedge x > 20) \wedge (l' = \text{L2} \wedge z' = 2 \wedge x' = x) \\
\varphi_5 =& (l = \text{L2} \wedge x \neq 0) \wedge (l' = l \wedge z' = z) \\
\varphi_6 =& (l = \text{L2} \wedge x = 0) \wedge (l' = \text{L3} \wedge z' = 0 \wedge x' = x) \\
\varphi_7 =& (l = \text{L3} \wedge x <= 10) \wedge (l' = l \wedge z' = z) \\
\varphi_{8a} =& (l = \text{L3} \wedge x > 10 \wedge x <= 20) \wedge (l' = \text{L1} \wedge z' = 1 \wedge x' = x) \\
\varphi_{8b} =& (l = \text{L3} \wedge x > 20) \wedge (l' = \text{L2} \wedge z' = 2 \wedge x' = x).
\end{aligned}$$

Consider the quiescent states $s_1 = \{x \mapsto 0, l \mapsto \text{L0}, z \mapsto 0\}$ and $s_2 = \{x \mapsto 0, l \mapsto \text{L3}, z \mapsto 0\}$. These two states are I/O-equivalent ($s_1 \sim_S s_2$), because Equation 2.27 holds: For every input trace, the same output trace is produced when applying the input trace to s_1 and s_2.

Consider, for example, the input trace $\iota = (5) \cdot (15) \cdot (4)$. The application of this input trace to s_1 results in a state trace of quiescent post states:

$s_1/\iota = \{x \mapsto 5, l \mapsto \text{L0}, z \mapsto 0\} \cdot \{x \mapsto 15, l \mapsto \text{L1}, z \mapsto 1\} \cdot \{x \mapsto 4, l \mapsto \text{L1}, z \mapsto 1\}$.

Applying the same input trace ι to state s_2 results in the following:

$s_2/\iota = \{x \mapsto 5, l \mapsto \text{L3}, z \mapsto 0\} \cdot \{x \mapsto 15, l \mapsto \text{L1}, z \mapsto 1\} \cdot \{x \mapsto 4, l \mapsto \text{L1}, z \mapsto 1\}$.

The output trace is identical for both states: $(s_1/\iota)|_O = (s_2/\iota)|_O = (0) \cdot (1) \cdot (1)$.

It is clear that all other input traces produce the same output when applied to s_1 and s_2, because the only outgoing edges from location L0 and L3 have the same guard condition and the same target location. That is why $s_1 \sim_S s_2$.

On the contrary, state $s_3 = \{x \mapsto 15, l \mapsto \text{L1}, z \mapsto 1\}$ is not equivalent to s_1. Applying concrete input $\vec{c}_1 = (7)$, for example, produces output $(s_1/\vec{c}_1)|_O = (0)$ when applied to s_1 and output $(s_3/\vec{c}_1)|_O = (1)$ when applied to s_3.

Every equivalence relation can be refined. $\sim'$ is a *finer* equivalence relation than (a *refined* equivalence relation of) $\sim$ if it fulfills $x \sim' y \Rightarrow x \sim y$. Conversely, $\sim$ is said to be *coarser* than $\sim'$. In the remainder of this work, we use the short-hand notation $\sim' \leq \sim$ to indicate that $\sim'$ is a refined equivalence relation of $\sim$.

Note that every state equivalence relation $\sim_r \subseteq S \times S$ that fulfills

$$s_1 \sim_r s_2 \Rightarrow \forall \iota \in D_I^* : (s_1/\iota)|_O = (s_2/\iota)|_O \tag{2.28}$$

is a refined equivalence relation of the I/O-equivalence relation $\sim_S$ defined in Equation 2.27. This implies that two states s_1 and s_2 that are equivalent according to any refined equivalence relation $\sim_r \leq \sim_S$ will always produce the same output traces for every possible input trace ι.

The partitioning of the reachable quiescent-state space by any refined equivalence relation $\sim_r \leq \sim_S$ is called State Equivalence Class Partitioning (SECP) and is denoted by $\mathcal{A} = \underline{S_Q}/_{\sim_r}$.

Note that, for every State Equivalence Class (SEC) $\mathbf{a} \in \mathcal{A}$, two states from $\mathbf{a}$, which are equivalent according to the refined relation $\sim_r$, will always produce the same output traces for every possible input trace ι, because $\sim_r$ is a refinement of $\sim_S$.

Every refinement $\mathcal{A}'$ of an SECP $\mathcal{A}$ is again an SECP. This is denoted by $\mathcal{A}' \subseteq \mathcal{A}$. The SECP $\mathcal{A}_c$ resulting from $\sim_S$ is called coarsest SECP because all other SECPs are refinements of the coarsest SECP.

2.5.2.2 MO-Partitioning

Note that two quiescent states s_1 and s_2 that differ only in the values of the input variables $s_1|_{M \cup O} = s_2|_{M \cup O}$ are always I/O-equivalent. This can be seen from the fact that applying any input vector $\vec{c}$ to these states will result in the same post state: $s_1 \oplus \{\vec{x} \mapsto \vec{c}\} = s_2 \oplus \{\vec{x} \mapsto \vec{c}\}$. $s \oplus \{\vec{x} \mapsto \vec{c}\}$ denotes the application of concrete input vector $\vec{c}$ to state s, which means that all input variables are set according to the input vector while the model and output variables remain unchanged. According to this, two quiescent states s_1 and s_2 with the same valuation for

model and output variables $s_1|_{M \cup O} = s_2|_{M \cup O}$, which will be denoted by $s_1 \sim_{\mathrm{MO}} s_2$, are always I/O-equivalent; therefore, $\sim_{\mathrm{MO}}$ fulfills Equation 2.28.

$$\forall s_1, s_2 \in S_Q : s_1 \sim_{\mathrm{MO}} s_2 \Rightarrow s_1 \sim_S s_2 \tag{2.29}$$

$\sim_{\mathrm{MO}}$ is an equivalence relation. Thus, the state space of reachable quiescent states can be partitioned by $\sim_{\mathrm{MO}}$ resulting in an equivalence class partitioning $\underline{S_Q}/_{\sim_{\mathrm{MO}}}$. This partitioning is an SECP, because $\sim_{\mathrm{MO}}$ is a finer relation than (a refinement of) $\sim_S$. Consequently, the SECP $\mathcal{A}_{\mathrm{MO}} = \underline{S_Q}/_{\sim_{\mathrm{MO}}}$ is a refinement of the coarsest SECP $\mathcal{A}_c = \underline{S_Q}/_{\sim_S}$: $\mathcal{A}_{\mathrm{MO}} \subseteq \mathcal{A}_c$.

Example 13. Recall the RIOSTS $\mathcal{S}_2$ as defined in Example 12. The MO-Partitioning $\mathcal{A}_{\mathrm{MO}}$ of the reachable quiescent states in $\mathcal{S}_2$ is as follows:

$$\mathcal{A}_{\mathrm{MO}} = \{\mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3\} \tag{2.30}$$
$$\mathbf{a}_0 = \{s \in \underline{S_Q} | s(l) = \mathrm{L0}, s(z) = 0\} \tag{2.31}$$
$$\mathbf{a}_1 = \{s \in \underline{S_Q} | s(l) = \mathrm{L1}, s(z) = 1\} \tag{2.32}$$
$$\mathbf{a}_2 = \{s \in \underline{S_Q} | s(l) = \mathrm{L2}, s(z) = 2\} \tag{2.33}$$
$$\mathbf{a}_3 = \{s \in \underline{S_Q} | s(l) = \mathrm{L3}, s(z) = 0\}. \tag{2.34}$$

As seen in Example 12, the two states, $s_1 = \{x \mapsto 0, l \mapsto \mathrm{L0}, z \mapsto 0\}$ and $s_2 = \{x \mapsto 0, l \mapsto \mathrm{L3}, z \mapsto 0\}$, are I/O-equivalent, although $s_1 \sim_{\mathrm{MO}} s_2$ does not hold. From the fact (1) that $s_1 \sim_S s_2$ and the fact (2) that all $\sim_{\mathrm{MO}}$-equivalent states are $\sim_S$-equivalent, the MO-classes of s_1 and s_2 can be united, yielding a new equivalence class $\mathbf{a}_0 \cup \mathbf{a}_3$ of states. The resulting SECP is guaranteed to be an SECP. Indeed, the resulting SECP is the coarsest SECP for $\mathcal{S}_2$.

$$\mathcal{A}_c = \{\mathbf{a}_0 \cup \mathbf{a}_3, \mathbf{a}_1, \mathbf{a}_2\} \tag{2.35}$$

No coarser partitioning exists, as can be demonstrated by application of input $\vec{c} = (3)$. $\mathbf{a}_0 \cup \mathbf{a}_3$, $\mathbf{a}_1$ and $\mathbf{a}_2$ produce the outputs (0), (1) and (2) respectively. Hence, all states from these classes cannot be I/O-equivalent to any state from any other SEC.

2.5.2.3 RIOSTSs Equivalence

Two RIOSTSs $\mathcal{S} = (S, s_0, R, V, D)$ and $\mathcal{S}' = (S', s_0', R', V', D')$ are I/O-equivalent iff the initial states are I/O-equivalent:

$$\mathcal{S} \sim \mathcal{S}' \Leftrightarrow s_0 \sim_S s_0'. \tag{2.36}$$

Example 14. Recall the RIOSTS $\mathcal{S}$ defined in Example 9 and the RIOSTS $\mathcal{S}_2$ defined in Example 12. Assuming that the initial state of $\mathcal{S}$ is $s_0 = \{x \mapsto 0, l \mapsto \mathrm{L0}, z \mapsto 0\}$ and the initial state of $\mathcal{S}_2$ is $s_1 = \{x \mapsto 0, l \mapsto \mathrm{L0}, z \mapsto 0\}$. A close look at the transition relation of both RIOSTSs reveals that both initial states are I/O-equivalent $s_0 \sim_S s_1$; therefore, both RIOSTSs are I/O-equivalent $\mathcal{S} \sim \mathcal{S}_2$.

2.5.2.4 Input Equivalence

Two inputs, $\vec{c}_1$ and $\vec{c}_2$, are I/O-equivalent $\vec{c}_1 \sim_I \vec{c}_2$ if the same output is produced for every reachable quiescent state when each of the two inputs is applied:

$$\vec{c}_1 \sim_I \vec{c}_2 \Leftrightarrow \forall s \in \underline{S_Q} : (s/\vec{c}_1)|_O = (s/\vec{c}_2)|_O. \tag{2.37}$$

Again, $\sim_I$ is an equivalence relation and therefore the input domain can be partitioned to $D_I/_{\sim_I}$.

Note that every input equivalence relation $\sim_{Ir} \subseteq D_I \times D_I$ that fulfills

$$\vec{c}_1 \sim_{Ir} \vec{c}_2 \Rightarrow \forall s \in \underline{S_Q} : (s/\vec{c}_1)|_O = (s/\vec{c}_2)|_O \tag{2.38}$$

is a refined equivalence relation of the I/O-equivalence relation $\sim_I$ defined in Equation 2.37. This implies that two inputs, $\vec{c}_1$ and $\vec{c}_2$, that are equivalent according to any refined equivalence relation $\sim_{Ir} \leq \sim_I$ will always produce the same output when applied to any quiescent state s.

The partitioning of the input domain by any refined equivalence relation $\sim_{Ir} \leq \sim_I$ is called Input Equivalence Class Partitioning (IECP) and will be be denoted by $\mathcal{I} = D_I/_{\sim_{Ir}}$.

Note that, for every IEC $X \in \mathcal{I}$, two concrete inputs from this class, which are equivalent according to the refined relation $\sim_{Ir}$, will always produce the same output for every quiescent state when each of the two inputs is applied, because $\sim_{Ir}$ is a refinement of $\sim_I$.

Every refinement $\mathcal{I}'$ of an IECP $\mathcal{I}$ is again an IECP. This is denoted by $\mathcal{I}' \subseteq \mathcal{I}$. The IECP $\mathcal{I}_c$ resulting from $\sim_I$ is called the coarsest IECP, because all other IECPs are refinements of the coarsest IECP.

Example 15. A precise look at the transition relation of $\mathcal{S}_2$ from Example 12 reveals that all concrete inputs with $x > 20$ are equivalent. For example, $\vec{c}_1 = (21)$ is I/O-equivalent to concrete input $\vec{c}_2 = (27)$: $\vec{c}_1 \sim_I \vec{c}_2$.

The same applies to all concrete inputs with $x > 10 \wedge x \leq 20$. Equally, all inputs with $x < 10 \wedge x \neq 0$ are I/O-equivalent.

This observation leads to an IECP $\mathcal{I}_c$ of $\mathcal{S}_2$.

$$\mathcal{I}_c = \{X_0, X_1, X_2, X_3\} \tag{2.39}$$
$$X_0 = \{(0)\} \tag{2.40}$$
$$X_1 = \{\vec{c} \in D_I | \vec{c} = (x), x \leq 10 \wedge x \neq 0\} \tag{2.41}$$
$$X_2 = \{\vec{c} \in D_I | \vec{c} = (x), x > 10 \wedge x \leq 20\} \tag{2.42}$$
$$X_3 = \{\vec{c} \in D_I | \vec{c} = (x), x > 20\} \tag{2.43}$$

This IECP is the coarsest IECP. Inputs from X_0 cannot be I/O-equivalent to any input from X_1, since, for a state s_2 from SEC $\mathbf{a}_2$, concrete inputs from both classes produce different outputs:

$(s_2/(0)|_O) = 0 \neq 2 = (s_2/(7)|_O)$.

The same argument applies for inputs from IECs X_0 and X_2 and for inputs from X_0 and X_3:

$(s_2/(0)|_O) = 0 \neq 2 = (s_2/(14)|_O)$

$(s_2/(0)|_O) = 0 \neq 2 = (s_2/(25)|_O)$.

Inputs from X_1 and X_2 produce different outputs when applied to a state s_0 from SEC $\mathbf{a}_0$:

$(s_0/(7)|_O) = 0 \neq 1 = (s_0/(14)|_O)$.

The same argument applies for inputs from X_1 and X_3:

$(s_0/(7)|_O) = 0 \neq 2 = (s_0/(25)|_O)$.

Inputs from X_2 and X_3 produce different outputs when applied to a state s_1 from SEC $\mathbf{a}_1$:

$(s_1/(14)|_7) = 1 \neq 2 = (s_1/(25)|_O)$.

A refined IECP $\mathcal{I}' \subseteq \mathcal{I}_c$ could be the following:

$$\mathcal{I}' = \{X_0, X_1, X_2, X_3\} \tag{2.44}$$
$$X_0 = \{(0)\} \tag{2.45}$$
$$X_1 = \{\vec{c} \in D_I | \vec{c} = (x), x \leq 10 \wedge x \neq 0\} \tag{2.46}$$
$$X_{2a} = \{\vec{c} \in D_I | \vec{c} = (x), x > 10 \wedge x \leq 15\} \tag{2.47}$$
$$X_{2b} = \{\vec{c} \in D_I | \vec{c} = (x), x > 15 \wedge x \leq 20\} \tag{2.48}$$
$$X_3 = \{\vec{c} \in D_I | \vec{c} = (x), x > 20\}. \tag{2.49}$$

2.5.2.5 FSM Abstraction of RIOSTSs

Given a RIOSTS $\mathcal{S}$ and two equivalence relations $\sim_r \leq \sim_S$ and $\sim_{Ir} \leq \sim_I$, where $\sim_r$ is a refinement of the state I/O-equivalence relation $\sim_S$ of $\mathcal{S}$ and $\sim_{Ir}$ is a refinement of the input I/O-equivalence relation $\sim_I$ of $\mathcal{S}$, the SECP of this RIOSTS is $\mathcal{A} = \underline{S_Q}/_{\sim_r}$ and the IECP is $\mathcal{I} = D_I/_{\sim_{Ir}}$.

Let $\delta : \mathcal{A} \times \mathcal{I} \to \mathcal{A}$ be a transition function. This function maps an SEC $\mathbf{a}$ and an IEC X to a unique SEC that contains exactly the target quiescent states that result from applying the concrete inputs from X to concrete states from $\mathbf{a}$:

$\forall \mathbf{a} \in \mathcal{A}, X \in \mathcal{I} : \forall s \in \mathbf{a}, \vec{c} \in X : (s/\vec{c}) \in \delta(\mathbf{a}, X)$.

Furthermore, let $\omega : \mathcal{A} \times \mathcal{I} \to D_O$ be an output function. This function maps SEC $\mathbf{a}$ and IEC X to the output that is produced when applying any input from X to any state from $\mathbf{a}$:

$\forall \mathbf{a} \in \mathcal{A}, X \in \mathcal{I} : \forall s \in \mathbf{a}, \vec{c} \in X : (s/\vec{c})(\vec{y}) = \omega(\mathbf{a}, X)$.

The authors in [HP16a] have proven that, for every RIOSTS $\mathcal{S}$ with finite domains for model and output variables, there exists a finite SECP $\mathcal{A}$, a finite IECP $\mathcal{I}$ and well-defined functions δ and ω. With these prerequisites at hand, $\mathcal{S}$ can be abstracted to a DFSM $M = (\mathcal{A}, \mathbf{a}_0, \mathcal{I}, D_O, \delta, \omega)$. The SECs from $\mathcal{A}$ serve as the finite set of states of M; $\mathcal{I}$ is the finite input alphabet and D_O is the output alphabet, which is finite by definition. δ and ω as defined above already provide themselves as the transition and output function of M. The initial state of M is $\mathbf{a}_0$, which is the SEC of the initial state s_0 of $\mathcal{S}$: $s_0 \in \mathbf{a}_0$.

Example 16. Given the RIOSTS $\mathcal{S}_2$ from Example 12 with SECP $\mathcal{A}_{\text{MO}}$ from Example 13 and IECP $\mathcal{I}'$ from Example 15, the transition function δ is defined by the following:

$$\begin{aligned}\delta(\mathbf{a}, X) = \{&(\mathbf{a}_0, X_0) \mapsto \mathbf{a}_0, (\mathbf{a}_0, X_1) \mapsto \mathbf{a}_0, (\mathbf{a}_0, X_{2a}) \mapsto \mathbf{a}_1, (\mathbf{a}_0, X_{2b}) \mapsto \mathbf{a}_1, (\mathbf{a}_0, X_3) \mapsto \mathbf{a}_2,\\ &(\mathbf{a}_1, X_0) \mapsto \mathbf{a}_1, (\mathbf{a}_1, X_1) \mapsto \mathbf{a}_1, (\mathbf{a}_1, X_{2a}) \mapsto \mathbf{a}_1, (\mathbf{a}_1, X_{2b}) \mapsto \mathbf{a}_1, (\mathbf{a}_1, X_3) \mapsto \mathbf{a}_2,\\ &(\mathbf{a}_2, X_0) \mapsto \mathbf{a}_3, (\mathbf{a}_2, X_1) \mapsto \mathbf{a}_2, (\mathbf{a}_2, X_{2a}) \mapsto \mathbf{a}_2, (\mathbf{a}_2, X_{2b}) \mapsto \mathbf{a}_2, (\mathbf{a}_2, X_3) \mapsto \mathbf{a}_2,\\ &(\mathbf{a}_3, X_0) \mapsto \mathbf{a}_3, (\mathbf{a}_3, X_1) \mapsto \mathbf{a}_3, (\mathbf{a}_3, X_{2a}) \mapsto \mathbf{a}_1, (\mathbf{a}_3, X_{2b}) \mapsto \mathbf{a}_1, (\mathbf{a}_3, X_3) \mapsto \mathbf{a}_2\}.\end{aligned}$$

The output function ω is defined by the following:

$$\begin{aligned}\omega(\mathbf{a}, X) = \{&(\mathbf{a}_0, X_0) \mapsto (0), (\mathbf{a}_0, X_1) \mapsto (0), (\mathbf{a}_0, X_{2a}) \mapsto (1), (\mathbf{a}_0, X_{2b}) \mapsto (1), (\mathbf{a}_0, X_3) \mapsto (2),\\ &(\mathbf{a}_1, X_0) \mapsto (1), (\mathbf{a}_1, X_1) \mapsto (1), (\mathbf{a}_1, X_{2a}) \mapsto (1), (\mathbf{a}_1, X_{2b}) \mapsto (1), (\mathbf{a}_1, X_3) \mapsto (2),\\ &(\mathbf{a}_2, X_0) \mapsto (0), (\mathbf{a}_2, X_1) \mapsto (2), (\mathbf{a}_2, X_{2a}) \mapsto (2), (\mathbf{a}_2, X_{2b}) \mapsto (2), (\mathbf{a}_2, X_3) \mapsto (2),\\ &(\mathbf{a}_3, X_0) \mapsto (0), (\mathbf{a}_3, X_1) \mapsto (0), (\mathbf{a}_3, X_{2a}) \mapsto (1), (\mathbf{a}_3, X_{2b}) \mapsto (1), (\mathbf{a}_3, X_3) \mapsto (2)\}.\end{aligned}$$

The FSM M for $\mathcal{S}_2$ is $M = (\mathcal{A}_{\text{MO}}, \mathbf{a}_0, \mathcal{I}', D_O, \delta, \omega)$. The state chart of M is shown in Figure 2.11.

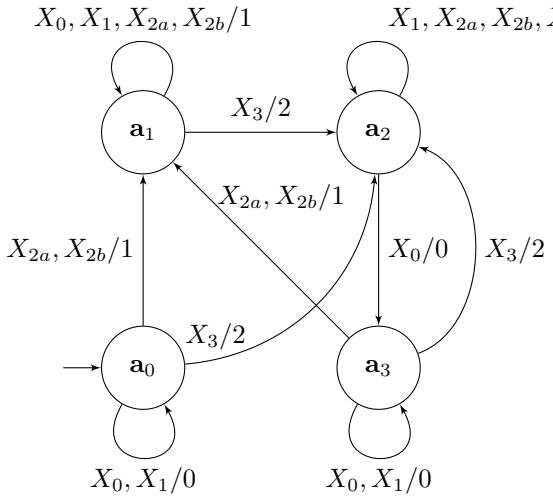


Figure 2.11: FSM of State Machine from Figure 2.10

2.5.2.6 Calculation of an Initial SECP

To obtain the FSM abstraction of a RIOSTS $\mathcal{S}$, an initial SECP of the reachable quiescent states is needed. This is calculated by reachability analysis using the predicate $\mathcal{R}$ describing the transition relation of $\mathcal{S}$.

All pairs of quiescent states (s_1, s_2) that are reachable through a finite (possibly empty) sequence of transient states $v \in {S_T}^*$ are collected $R^*(s_1 \cdot v \cdot s_2)$. These pairs will be called *reachability*

pairs. To avoid infinitely many reachability pairs, the MO-equivalence relation is used. States and state pairs (s_1, s_2) are abstracted to MO-equivalence classes $\mathbf{a} \in \mathcal{A}_{\mathrm{MO}}$. This yields pairs of MO-equivalence classes $(\mathbf{a}_i, \mathbf{a}_j)$ that represent concrete reachability pairs. These pairs will be called *reachability MO-pairs*. Symbolic evaluation of $\mathcal{R}$ makes it possible to define predicates $\alpha_{i,j}$ over input variables that describe the condition for a concrete input to cause a transition from a state in $\mathbf{a}_i$ to a state in $\mathbf{a}_j$. For a pair of MO-classes $\mathbf{a}_i$ and $\mathbf{a}_j$ that is not a reachability pair, let $\alpha_{i,j} \triangleq \mathrm{false}$.

For a RIOSTS $\mathcal{S}$ with finite domains for internal and output variables, the reachability analysis will terminate, because $\mathcal{A}_{\mathrm{MO}} = \underline{S_Q}/_{\sim_{\mathrm{MO}}}$ is finite.

Note that the reachability MO-pairs include pairs of the form $(\mathbf{a}_i, \mathbf{a}_i)$ for every MO-class $\mathbf{a}_i \in \mathcal{A}_{\mathrm{MO}}$. In this case, $\alpha_{i,i}$ describes the quiescence condition for MO-class $\mathbf{a}_i$.

The reachability MO-pairs $(\mathbf{a}_i, \mathbf{a}_j)$ and the predicates $\alpha_{i,j}$ can be used to define a transition-relation predicate $\underline{\mathcal{R}}$ of $\mathcal{S}$ in the so-called *Index-Normal-Form (INF)*. To this end, let $\mathrm{IDX} = \{1, \ldots, n\}$ denote the indices of the state MO-classes $\mathcal{A}_{\mathrm{MO}} = \underline{S_Q}/_{\sim_{\mathrm{MO}}} = \{\mathbf{a}_1, \ldots, \mathbf{a}_n\}$. Each state class $\mathbf{a}_i$ is associated with concrete model variable values $\vec{d}_i$ and concrete output variable values $\vec{e}_i$. Thus, for all states in $\mathbf{a}_i$, all model variables $\vec{m}$ are set to $\vec{d}_i$ and all output variables $\vec{y}$ are set to $\vec{e}_i$. The INF is a predicate of the following form:

$$\underline{\mathcal{R}} \triangleq \bigvee_{i \in \mathrm{IDX}} \alpha_{i,i} \wedge (\vec{m}, \vec{y}) = (\vec{d}_i, \vec{e}_i) \wedge (\vec{m}', \vec{y}') = (\vec{d}_i, \vec{e}_i) \tag{2.50}$$

$$\bigvee_{i,j \in \mathrm{IDX}, i \neq j} \alpha_{i,j} \wedge (\vec{m}, \vec{y}) = (\vec{d}_i, \vec{e}_i) \wedge (\vec{m}', \vec{y}') = (\vec{d}_j, \vec{e}_j). \tag{2.51}$$

[HP16a] presents an approach to transforming a transition-relation predicate in arbitrary form to a predicate in INF. Instead of this approach, we present our optimised reachability analysis in Section 4.3.5.1. We will show that this reachability analysis can be used to efficiently calculate an initial SECP and a transition-relation predicate in INF.

Example 17. Consider the RIOSTS $\mathcal{S}_2$ from Example 12 with SECP $\mathcal{A}_{\mathrm{MO}}$ depicted in Example 13. $\mathcal{A}_{\mathrm{MO}}$ can be calculated by reachability analysis starting from initial state $s_0 = \{x \mapsto 0, l \mapsto \mathrm{L0}, z \mapsto 0\}$. The first state MO-class is $\mathbf{a}_0$, as defined in Example 13. From this state, $\mathbf{a}_1$ and $\mathbf{a}_2$ can be reached by inputs fulfilling predicates $\alpha_{0,1} \triangleq x > 10 \wedge x \leq 20$ and $\alpha_{0,2} \triangleq x > 20$, respectively. All inputs fulfilling $\alpha_{0,0} \triangleq x \leq 10$ lead from a state in $\mathbf{a}_0$ to post states in the same state MO-class.

From $\mathbf{a}_1$, a transition to states in $\mathbf{a}_2$ is possible by inputs fulfilling $\alpha_{1,2} \triangleq x > 20$, and transitions to states from $\mathbf{a}_1$ itself are performed by inputs described by $\alpha_{1,1} \triangleq x \leq 20$.

From $\mathbf{a}_2$, transitions to $\mathbf{a}_2$ and $\mathbf{a}_3$ are possible by inputs described by $\alpha_{2,2} \triangleq x \neq 0$ and $\alpha_{2,3} \triangleq x = 0$, respectively.

From $\mathbf{a}_3$, transitions to $\mathbf{a}_3$, $\mathbf{a}_1$ and $\mathbf{a}_2$ are possible by inputs described by $\alpha_{3,3} \triangleq x \leq 10$, $\alpha_{3,1} \triangleq x > 10 \wedge x \leq 20$ and $\alpha_{3,2} \triangleq x > 20$, respectively.

The transition relation of $\mathcal{S}_2$ can be described by a predicate in INF:

$$\underline{\mathcal{R}}_2 \triangleq \quad \alpha_{0,0} \wedge (l,z) = (0,0) \wedge (l',z') = (0,0) \tag{2.52}$$
$$\vee \quad \alpha_{1,1} \wedge (l,z) = (1,1) \wedge (l',z') = (1,1) \tag{2.53}$$
$$\vee \quad \alpha_{2,2} \wedge (l,z) = (2,2) \wedge (l',z') = (2,2) \tag{2.54}$$
$$\vee \quad \alpha_{3,3} \wedge (l,z) = (3,0) \wedge (l',z') = (3,0) \tag{2.55}$$
$$\vee \quad \alpha_{0,1} \wedge (l,z) = (0,0) \wedge (l',z') = (1,1) \tag{2.56}$$
$$\vee \quad \alpha_{0,2} \wedge (l,z) = (0,0) \wedge (l',z') = (2,2) \tag{2.57}$$
$$\vee \quad \alpha_{1,2} \wedge (l,z) = (1,1) \wedge (l',z') = (2,2) \tag{2.58}$$
$$\vee \quad \alpha_{2,3} \wedge (l,z) = (2,2) \wedge (l',z') = (3,0) \tag{2.59}$$
$$\vee \quad \alpha_{3,1} \wedge (l,z) = (3,0) \wedge (l',z') = (1,1) \tag{2.60}$$
$$\vee \quad \alpha_{3,2} \wedge (l,z) = (3,0) \wedge (l',z') = (2,2). \tag{2.61}$$

2.5.2.7 Calculation of IECPs

Given an initial SECP and a transition relation in INF, an initial IECP $\mathcal{I}$ can be calculated. To this end, consider all possible functions: $f : \mathrm{IDX} \to \mathrm{IDX}$. If the predicate Φ_f

$$\Phi_f \triangleq \bigwedge_{i \in \mathrm{IDX}} \alpha_{i,f(i)} \tag{2.62}$$

has a solution, then the solution set of this predicate forms an IEC $X_f = \{\vec{c} \in D_I | \Phi_f[\vec{c}/\vec{x}]\}$.

[HP16a] gives a proof that the IECP $\mathcal{I} = \{X_{f_1}, \ldots, X_{f_n}\}$ composed of the solution sets of all solvable predicates Φ_{f_i} is an IECP, as defined in Section 2.5.2.4.

In Section 4.3.5.2, we describe our algorithm to efficiently calculate the IECP, as defined above.

Note that the transition function δ of the DFSM abstraction of a RIOSTS can be directly obtained from the functions $f : \mathrm{IDX} \to \mathrm{IDX}$. Given an IEC X_f, the target states can be obtained from f: $\delta(\mathbf{a}_i, X_f) = \mathbf{a}_{f(i)}$.

Example 18. Consider the RIOSTS $\mathcal{S}_2$ from Example 12 and the transition-relation predicate $\underline{\mathcal{R}}_2$ in INF, as defined in Example 17.

There are 256 possibilities for functions $f : \mathrm{IDX} \to \mathrm{IDX}$.

From these possibilities, four functions result in predicates Φ_f with at least one solution:
$f_0 = \{0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 3, 3 \mapsto 3\}$ $\quad \Phi_{f_0} \triangleq x \leq 10 \wedge x \leq 20 \wedge x = 0 \wedge x \leq 10$
$f_1 = \{0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3\}$ $\quad \Phi_{f_1} \triangleq x \leq 10 \wedge x \leq 20 \wedge x \neq 0 \wedge x \leq 10$
$f_2 = \{0 \mapsto 1, 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 1\}$ $\quad \Phi_{f_2} \triangleq x > 10 \wedge x \leq 20 \wedge x \leq 20 \wedge x \neq 0 \wedge x > 10 \wedge x \leq 20$
$f_3 = \{0 \mapsto 2, 1 \mapsto 2, 2 \mapsto 2, 3 \mapsto 2\}$ $\quad \Phi_{f_3} \triangleq x > 20 \wedge x > 20 \wedge x \neq 0 \wedge x > 20.$

For every other $f : \mathrm{IDX} \to \mathrm{IDX}$, no solution exists. For example, for $f_4 = \{0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 0\}$ no solution for Φ_{f_4} exists because $\alpha_{3,0} \triangleq$ false.

For $f_5 = \{0 \mapsto 0, 1 \mapsto 2, 2 \mapsto 2, 3 \mapsto 3\}$ and the associated predicate $\Phi_{f_5} \triangleq x \leq 10 \wedge x > 20 \wedge x \neq \wedge x \leq 10$, no solution exists because $x > 20$ contradicts $x \leq 10$.

Note that the IECP $\{X_{f_0}, X_{f_1}, X_{f_2}, X_{f_3}\}$ matches the IECP $\{X_0, X_1, X_2, X_3\}$ presented in Example 15.

2.5.2.8 Calculation of the Coarsest SECPs and IECPs

Given the initial SECP $\mathcal{A}_{\mathrm{MO}}$ and the initial IECP $\mathcal{I}$, a DFSM abstraction $M = (\mathcal{A}, \mathbf{a}_0, \mathcal{I}, D_O, \delta, \omega)$ of the RIOSTS $\mathcal{S}$ can be obtained. Because of the use of MO-equivalence $\sim_{\mathrm{MO}}$ for the initial SECP, this abstraction depends on the concrete syntactical model representation of $\mathcal{S}$, since internal model variables are considered for $\sim_{\mathrm{MO}}$ equivalence. The internal model variables may introduce redundant (i.e., I/O-equivalent) states in the DFSM.

However, any DFSM M can be minimised using Algorithm 1. The resulting minimal DFSM is $M' = (\mathcal{A}_c, \mathbf{a}'_0, \mathcal{I}, D_O, \delta', \omega')$. The state set of this minimal DFSM is the coarsest SECP $\mathcal{A}_c$ for $\mathcal{S}$, because in a minimal DFSM no two states with the same language exist. Therefore, no coarser SECP can exist.

The IECP $\mathcal{I}$, which has been calculated initially, is dependent on MO-classes $\mathbf{a} \in \mathcal{A}_{\mathrm{MO}}$ (cf. Section 2.5.2.7) and therefore is not guaranteed to be the coarsest IECP. There might exist two IECs $X_1, X_2 \in \mathcal{I}$ that produce the same output and target state for all states in M'. In this case, inputs from both IECs are I/O-equivalent, according to Equation 2.37. We will denote this fact as $X_1 \sim_I X_2$:

$$X_1 \sim_I X_2 \iff \forall \mathbf{a} \in \mathcal{A}_c : \delta'(\mathbf{a}, X_1) = \delta'(\mathbf{a}, X_2) \wedge \omega'(\mathbf{a}, X_1) = \omega'(\mathbf{a}, X_2). \tag{2.63}$$

If $X_1 \sim_I X_2$, all members of X_1 and X_2 are pairwise I/O-equivalent; therefore, the union $X_1 \cup X_2$ is an IEC. The coarsest IECP $\mathcal{I}_c$ can be obtained by successively uniting pairs of IECs that are I/O-equivalent until no pairs of I/O-equivalent IECs remain.

The construction of $\mathcal{I}_c$ ensures maximal coarseness. Every pair of non-united IECs X_1 and X_2 cannot be I/O-equivalent, because X_1 and X_2 produce either different outputs $\omega'(\mathbf{a}, X_1) \neq \omega'(\mathbf{a}, X_2)$ or lead to different target states $\delta'(\mathbf{a}, X_1) \neq \delta'(\mathbf{a}, X_2)$ that must be non-I/O-equivalent because of the minimality of M'.

Example 19. Consider the RIOSTS $\mathcal{S}_3$, whose behaviour is described by the state machine shown in Figure 2.12. The MO-partitioning for $\mathcal{S}_3$ is as follows:

$$\mathcal{A}_{\mathrm{MO}} = \{\mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3, \mathbf{a}_4, \mathbf{a}_5\} \tag{2.64}$$
$$\mathbf{a}_0 = \{s \in \underline{S_Q} | s(l) = \mathrm{L0}, s(z) = 0\} \tag{2.65}$$
$$\mathbf{a}_1 = \{s \in \underline{S_Q} | s(l) = \mathrm{L1}, s(z) = 1\} \tag{2.66}$$
$$\mathbf{a}_2 = \{s \in \underline{S_Q} | s(l) = \mathrm{L2}, s(z) = 2\} \tag{2.67}$$
$$\mathbf{a}_3 = \{s \in \underline{S_Q} | s(l) = \mathrm{L3}, s(z) = 2\} \tag{2.68}$$
$$\mathbf{a}_4 = \{s \in \underline{S_Q} | s(l) = \mathrm{L4}, s(z) = 0\} \tag{2.69}$$
$$\mathbf{a}_5 = \{s \in \underline{S_Q} | s(l) = \mathrm{L5}, s(z) = 1\}. \tag{2.70}$$

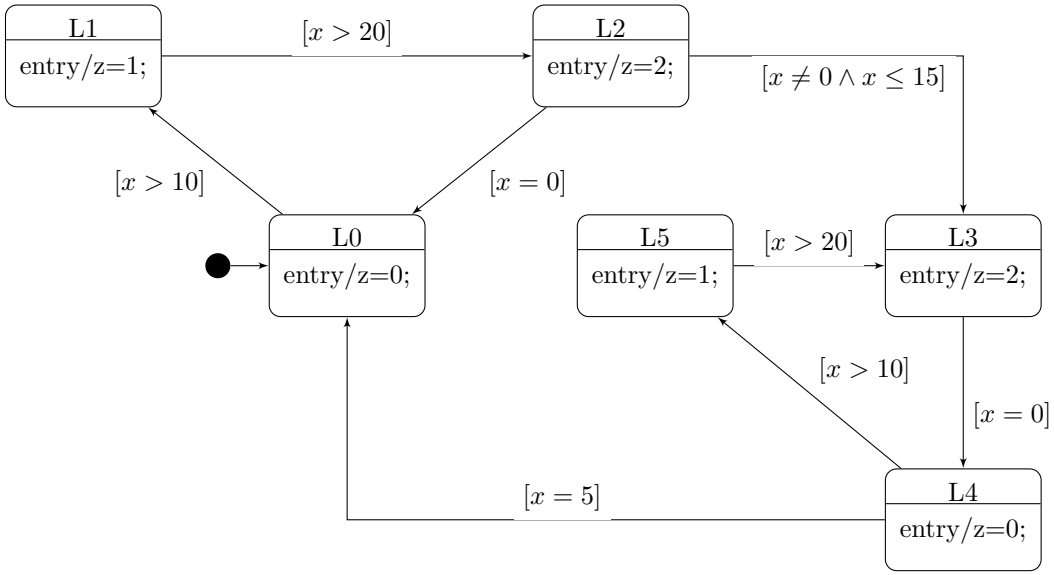


Figure 2.12: Example State Machine Describing the Behaviour of $\mathcal{S}_3$

The initial IECP that results from the application of the approach explained in Section 2.5.2.4 is as follows:

$$\mathcal{I} = \{X_0, X_1, X_2, X_3, X_4, X_5\} \tag{2.71}$$
$$X_0 = \{(0)\} \tag{2.72}$$
$$X_1 = \{(5)\} \tag{2.73}$$
$$X_2 = \{\vec{c} \in D_I | \vec{c} = (x), x \leq 10 \wedge x \neq 0 \wedge x \neq 5\} \tag{2.74}$$
$$X_3 = \{\vec{c} \in D_I | \vec{c} = (x), x > 10 \wedge x \leq 15\} \tag{2.75}$$
$$X_4 = \{\vec{c} \in D_I | \vec{c} = (x), x > 15 \wedge x \leq 20\} \tag{2.76}$$
$$X_5 = \{\vec{c} \in D_I | \vec{c} = (x), x > 20\}. \tag{2.77}$$

Most likely, it is not trivial to see whether $\mathcal{A}_{\text{MO}}$ and $\mathcal{I}$ are the coarsest partitionings. Figure 2.13 shows the DFSM abstraction M of $\mathcal{S}_3$. The minimal DFSM M' that results from minimisation of M is shown in Figure 2.13b. As can be seen from M', the coarsest SECP for $\mathcal{S}_3$ is

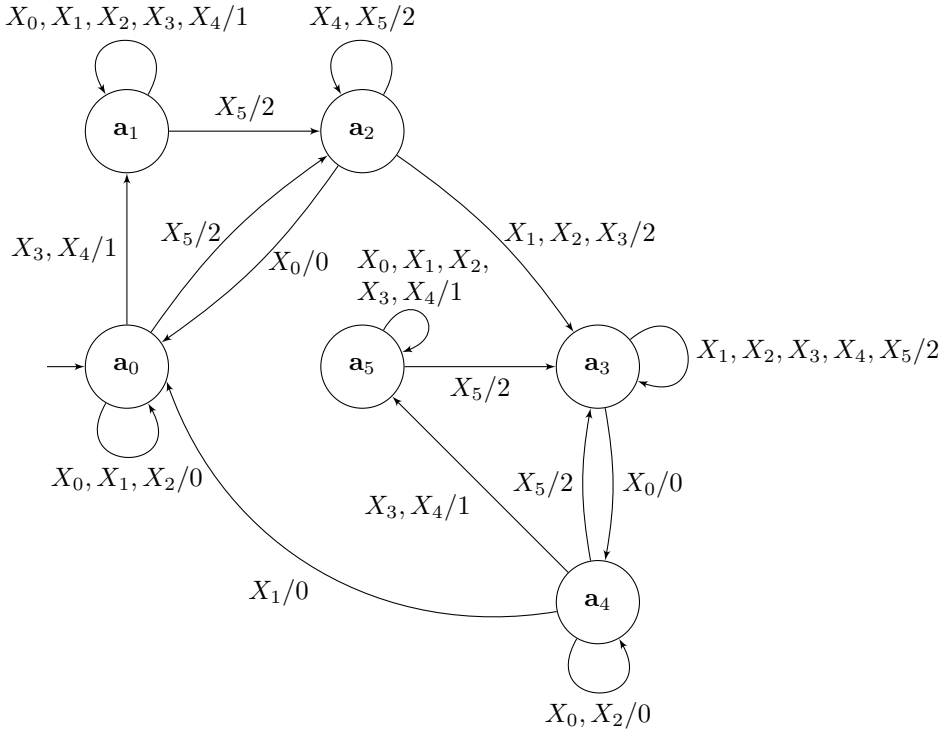
$\mathcal{A}_{\text{MO}} = \{\mathbf{a}_0 \cup \mathbf{a}_4, \mathbf{a}_1 \cup \mathbf{a}_5, \mathbf{a}_2 \cup \mathbf{a}_3\}$.

A precise look at M' also reveals that, $X_1 \sim_I X_2$ and $X_3 \sim_I X_4$. Therefore, the coarsest IECP is

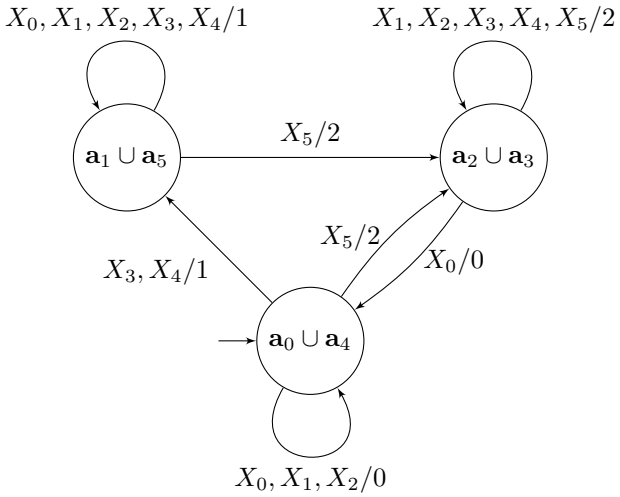
$\mathcal{I}_c = \{X_0, X_1 \cup X_2, X_3 \cup X_4, X_5\}$.

2.5.3 Fault Domain and the Completeness Property

Recall the fault model $\mathcal{F}(M, \sim, \mathcal{D}(m))$ for DFSMs. The application of the W/Wp-method to DFSMs guarantees completeness with respect to the fault domain $\mathcal{D}(m)$: All possible transfer



(a) DFSM M



(b) Minimal DFSM M'

Figure 2.13: DFSM Abstraction of $\mathcal{S}_3$

and output faults are revealed by the application of test suites generated by W/Wp-method, assuming that the SUT has at most m states in its minimal DFSM representation.

Definition 21 (Fault Model for RIOSTSs)**.** Let $\mathcal{F}(\mathcal{S}, \leq, \mathcal{D}(\mathcal{S}, m, \mathcal{I}))$ be the fault model for a RIOSTS $\mathcal{S}$ and a conformance relation $\leq$. In this work, we will use $\sim$ as a conformance relation.

The fault domain for RIOSTS depends on parameters m and $\mathcal{I}$. $\mathcal{I}$ is any (refined) IECP of the specification $\mathcal{S}$ that has been used as an input alphabet for the DFSM abstraction of $\mathcal{S}$.

The fault domain contains all RIOSTSs $\mathfrak{I}$ that fulfil the following conditions:

1. $\mathfrak{I}$ uses the same input and output variables $I \cup O$ with the same domains D_I and D_O as $\mathcal{S}$
2. the initial states s_0 of $\mathcal{S}$ and ${s_0}'$ of $\mathfrak{I}$ agree on the same input and output valuation: $s_0|_{I \cup O} = {s_0}'|_{I \cup O}$
3. $\mathfrak{I}$ is a deterministic RIOSTS with finite domain for internal and output variables
4. $\mathfrak{I}$ has a correctly implemented reset operation
5. Let $\mathcal{I}_c$ be the coarsest IECP of $\mathcal{S}$. Then, the coarsest IECP $\mathcal{I}_\mathfrak{I}$ of $\mathfrak{I}$ has to fulfil Equation 2.78.
6. the number of states in the minimal DFSM abstraction of $\mathfrak{I}$ is at most m

$$\forall X_\mathcal{S} \in \mathcal{I}_c, X_\mathfrak{I} \in \mathcal{I}_\mathfrak{I} : X_\mathcal{S} \cap X_\mathfrak{I} \neq \emptyset \Rightarrow \exists X' \in \mathcal{I} : X' \subseteq X_\mathcal{S} \cap X_\mathfrak{I} \tag{2.78}$$

The intuition of Equation 2.78 is as follows: a correct implementation will have a coarsest IECP $\mathcal{I}_\mathfrak{I}$ that matches the coarsest IECP $\mathcal{I}_c$ of $\mathcal{S}$. However, an erroneous implementation might show deviating behaviour from $\mathcal{S}$ only for a subset of some IEC. In this case, the coarsest IECPs of $\mathcal{S}$ and $\mathfrak{I}$ do not match. There will exist at least one non-empty intersection $X_\mathcal{S} \cap X_\mathfrak{I}$ for which the implementation shows erroneous behaviour in at least one system state. That is why the condition states that, for every non-empty intersection of IECPs in the specification $\mathcal{S}$ and in the implementation $\mathfrak{I}$, at least one member X' of the IECP $\mathcal{I}$ is needed that is completely inside the intersection. In this case, the deviation of observable behavior in $\mathfrak{I}$ can be revealed by any concrete member of X'. In summary, the IECP $\mathcal{I}$ used for test-case generation has to be fine-grained enough to uncover faults in the implementation that result from mismatching IECPs.

In [HP16a], the authors have shown that the W/Wp-method applied to the DFSM abstraction of a RIOSTS with finite domain for internal and output variables is complete with respect to the fault model $\mathcal{F}(\mathcal{S}, \leq, \mathcal{D}(\mathcal{S}, m, \mathcal{I}))$. This result builds the foundation for the testing methodology implemented in this work. In Chapter 4, we use this result to implement a testing methodology for RIOSTS.

First, let us conclude from the definitions and results presented in the previous paragraphs. An important fact to note is that the fault domain (and therefore the test strength that is guaranteed by our testing methodology) can be increased by refining the IECP used for DFSM abstraction of $\mathcal{S}$. This increases the size of the input alphabet and will therefore result in a polynomial increase of the number of test cases that are generated using the W/Wp-method. This approach will be presented in Section 4.3.1. Another way to enhance the fault domain is via increase of the value for m. In this case, the testing effort increases exponentially, but the fault domain

contains implementations with higher numbers of states. Both approaches greatly increase the test effort of our approach. Therefore, we present heuristics that substitute the IECP refinement (see Section 4.3.2) and the increase of m (see Section 4.3.4). While these heuristics represent best-effort approaches and do not guarantee better test strength (the fault domain is not increased), the number of tests that are generated is not increased; still, the evaluation (Chapter 6) will confirm a higher test strength resulting from these heuristics when mutation analysis is used as an evaluation means.

2.5.4 Independence on Syntactic Model Representations

Another property of our approach has been elaborated in [PH16]. Note that the coarsest SECP and coarsest IECP that are calculated as described above are independent of the concrete syntactic representation of a RIOSTS $\mathcal{S}$. All semantically equivalent concrete descriptions of $\mathcal{S}' \sim \mathcal{S}$ will result in the same minimised DFSM abstraction, and determination of the coarsest IECP will result in unique model representation that is based only on I/O-equivalence: i.e., on the I/O-language of the system. The resulting test suite that is generated using the W/Wp-method on the minimal DFSM abstraction is completely independent of the concrete behavioural description of $\mathcal{S}$. This result has been presented and proven by the authors of [PH16]. Besides the main advantage of the completeness with respect to a fault domain, our testing methodology offers the advantage that the generated test suites are independent of a concrete system description. According to this, it is guaranteed that the resulting test strength is not dependent on concrete models. This is a main threat to validity that applies to all testing methodologies that rely solely on a concrete syntactic representation of a system model. Our approach instead guarantees test suites of equal test strength for all possible concrete descriptions of an SUT.

3 Case Studies

In this chapter, we introduce the case studies that our approach is applied on. First we introduce the Ceiling Speed Monitor which is part of the speed and distance monitoring function of ETCS trains. Next, we introduce two case studies that model the behaviour of route controllers from modern interlocking systems featuring sequential release. Finally, this sections concludes with a last case study modelling an airbag controller.

3.1 Speed and Distance Monitoring of the European Train Control System

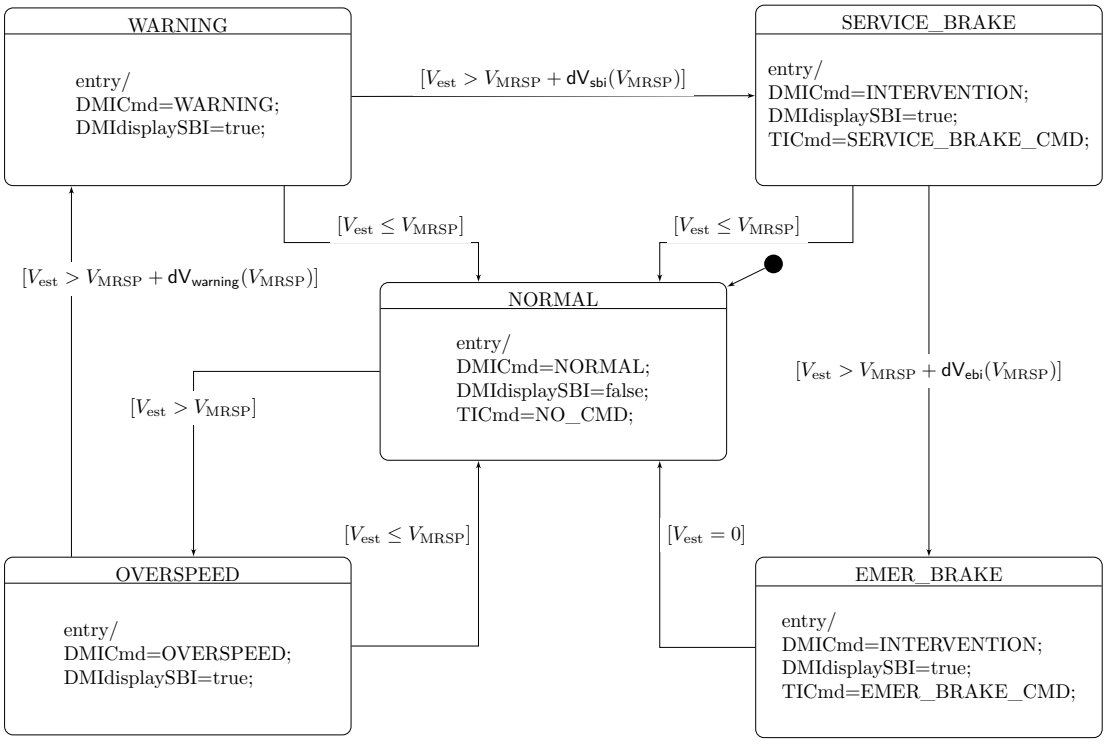


Figure 3.1: State Machine of the Ceiling Speed Monitor

Rationale Speed and distance monitoring is a main function of the EVC. This functionality operates in different modes, as described in Section 2.1.3. One mode of the speed and distance

monitoring function is the so-called ceiling speed mode. This mode is active when a train is travelling on a route and not yet reaching its destination. In this mode, the velocity of the train has to be supervised by the EVC. This functionality lends itself to our testing approach, as it is easy to model: The input domain is conceptually of infinite size (the train's current velocity and the maximum allowed speed are analogue values) and yet are complex enough to demonstrate the effectiveness of our approach.

While the train is in ceiling speed mode, the sub-component of the speed and distance monitoring called *CSM* is active. The CSM constantly supervises the current train speed (indicated by the system input variable V_{est}). If the speed exceeds the velocity that is prescribed by the most restrictive speed profile (mrsp) that applies to the current train position (the allowed velocity is indicated by system input variable V_{MRSP}), an intervention by the EVC is triggered. The intervention level reaches from an optical indication to the train-driver to the automatic activation of the train's emergency brakes and depends on the limit violation.

SysMLState Machine The state machine shown in Figure 3.1 describes the behaviour of the CSM. This state machine was first published in [HHP15]. The system initially resides in the NORMAL location. As soon as the train exceeds the maximum allowed speed $V_{\text{est}} > V_{\text{MRSP}}$, the state machine transitions to location OVERSPEED. A graphical sign in the driver-machine interface (DMI) indicates overspeeding to the train driver (output variable `DMIdisplaySBI`). Furthermore, the DMI operates in different modes that depend on the intervention level of the CSM. The DMI mode is determined by the CSM output variable `DMICmd`. If the train speed exceeds the speed limit by the threshold $\mathsf{dV}_{\mathsf{warning}}$, the state machine enters location WARNING, which results in a different DMI mode. As soon as the higher threshold $\mathsf{dV}_{\mathsf{sbi}}$ is exceeded as well, the location SERVICE_BRAKE is entered. Besides a change of the DMI mode, the train's service brakes are activated, which is indicated by output variable `TICmd`. A final threshold $\mathsf{dV}_{\mathsf{ebi}}$ exists. If this threshold is exceeded as well, the activation of the train's emergency brakes is triggered in state-machine location EMER_BRAKE. The intervention by CSM can only be released when the train velocity is back in the allowed range $V_{\text{est}} \leq V_{\text{MRSP}}$. However, location EMERGENCY_BRAKE can only be left when the train has completely stopped ($V_{\text{est}} = 0$).

The thresholds $\mathsf{dV}_{\mathsf{warning}}$, $\mathsf{dV}_{\mathsf{sbi}}$ and $\mathsf{dV}_{\mathsf{ebi}}$ depend on the speed limit.

$\mathsf{dV}_{\mathsf{warning}}$ is defined by the following step-function:

$$\mathsf{dV}_{\mathsf{warning}} \triangleq \begin{cases} 4 & \text{if } V_{\text{MRSP}} \leq 110 \\ \frac{V_{\text{MRSP}}}{30} + \frac{1}{3} & \text{if } 110 < V_{\text{MRSP}} \leq 140 \\ 5 & \text{if } V_{\text{MRSP}} > 140. \end{cases} \tag{3.1}$$

$\mathsf{dV}_{\mathsf{sbi}}$ and $\mathsf{dV}_{\mathsf{ebi}}$ are defined by the following:

$$\mathsf{dV}_{\mathsf{sbi}} \triangleq \begin{cases} 5.5 & \text{if } V_{\text{MRSP}} \leq 110 \\ 0.045 \cdot V_{\text{MRSP}} + 0.55 & \text{if } 110 < V_{\text{MRSP}} \leq 210 \\ 10 & \text{if } V_{\text{MRSP}} > 210 \end{cases} \tag{3.2}$$

$$\mathsf{dV_{ebi}} \triangleq \begin{cases} 7.5 & \text{if } V_{\text{MRSP}} \leq 110 \\ 0.075 \cdot V_{\text{MRSP}} - 0.75 & \text{if } 110 < V_{\text{MRSP}} \leq 210 \\ 15 & \text{if } V_{\text{MRSP}} > 210. \end{cases} \tag{3.3}$$

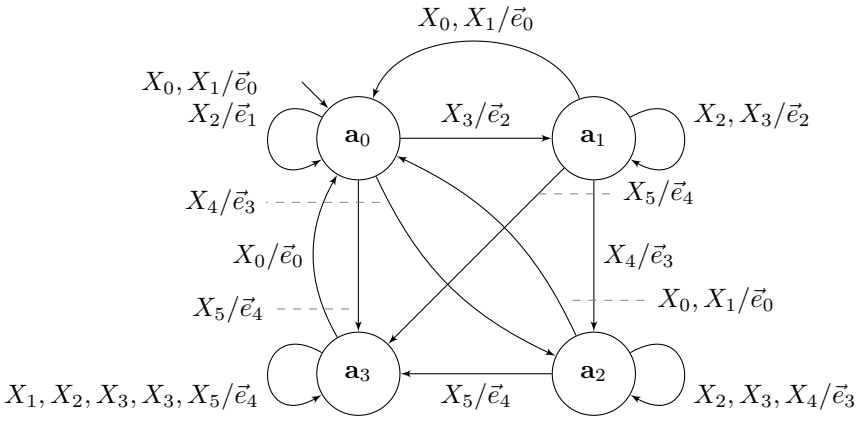


Figure 3.2: DFSM Abstraction of the Ceiling Speed Monitor

DFSM Abstraction The DFSM abstraction of the CSM is shown in Figure 3.2. The output vectors of the system–containing values for output variables `DMIdisplaySBI`, `DMICmd` and `TICmd` in that order–are defined as follows:

$$\vec{e}_0 = (\text{false,NORMAL,NO\_CMD}) \tag{3.4}$$
$$\vec{e}_1 = (\text{true,OVERSPEED,NO\_CMD}) \tag{3.5}$$
$$\vec{e}_2 = (\text{true,WARNING,NO\_CMD}) \tag{3.6}$$
$$\vec{e}_3 = (\text{true,INTERVENTION,SERVICE\_BRAKE\_CMD}) \tag{3.7}$$
$$\vec{e}_4 = (\text{true,INTERVENTION,EMER\_BRAKE\_CMD}). \tag{3.8}$$

The coarsest SECP of the CSM is given as follows:

$$\mathcal{A}_c \triangleq \{\mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3\} \tag{3.9}$$
$$\mathbf{a}_0 = \{s \in \underline{S_Q} | (s(l) = \text{NORMAL} \wedge s(\vec{y}) = \vec{e}_0) \vee (s(l) = \text{OVERSPEED} \wedge s(\vec{y}) = \vec{e}_1)\} \tag{3.10}$$
$$\mathbf{a}_1 = \{s \in \underline{S_Q} | s(l) = \text{WARNING} \wedge s(\vec{y}) = \vec{e}_2\} \tag{3.11}$$
$$\mathbf{a}_2 = \{s \in \underline{S_Q} | s(l) = \text{SERVICE\_BRAKE} \wedge s(\vec{y}) = \vec{e}_3\} \tag{3.12}$$
$$\mathbf{a}_3 = \{s \in \underline{S_Q} | s(l) = \text{EMER\_BRAKE} \wedge s(\vec{y}) = \vec{e}_4\}. \tag{3.13}$$

The coarsest IECP $\mathcal{I}_c$ is illustrated in Figure 3.3.

$$\mathcal{I}_c = \{X_0, X_1, X_2, X_3, X_4, X_5\} \tag{3.14}$$
$$X_0 = \{\vec{c} \in D_I | \vec{c} = (V_{\text{est}}, V_{\text{MRSP}}), V_{\text{est}} = 0\} \tag{3.15}$$
$$X_1 = \{\vec{c} \in D_I | \vec{c} = (V_{\text{est}}, V_{\text{MRSP}}), V_{\text{est}} \neq 0 \wedge V_{\text{est}} \leq V_{\text{MRSP}}\} \tag{3.16}$$
$$X_2 = \{\vec{c} \in D_I | \vec{c} = (V_{\text{est}}, V_{\text{MRSP}}), V_{\text{est}} \neq 0 \wedge V_{\text{est}} > V_{\text{MRSP}} \tag{3.17}$$
$$\wedge V_{\text{est}} \leq V_{\text{MRSP}} + \mathsf{dV}_{\mathsf{warning}}(V_{\text{MRSP}})\} \tag{3.18}$$
$$X_3 = \{\vec{c} \in D_I | \vec{c} = (V_{\text{est}}, V_{\text{MRSP}}), V_{\text{est}} \neq 0 \wedge V_{\text{est}} > V_{\text{MRSP}} + \mathsf{dV}_{\mathsf{warning}}(V_{\text{MRSP}}) \tag{3.19}$$
$$\wedge V_{\text{est}} \leq V_{\text{MRSP}} + \mathsf{dV}_{\mathsf{sbi}}(V_{\text{MRSP}})\} \tag{3.20}$$
$$X_4 = \{\vec{c} \in D_I | \vec{c} = (V_{\text{est}}, V_{\text{MRSP}}), V_{\text{est}} \neq 0 \wedge V_{\text{est}} > V_{\text{MRSP}} + \mathsf{dV}_{\mathsf{sbi}}(V_{\text{MRSP}}) \tag{3.21}$$
$$\wedge V_{\text{est}} \leq V_{\text{MRSP}} + \mathsf{dV}_{\mathsf{ebi}}(V_{\text{MRSP}})\} \tag{3.22}$$
$$X_5 = \{\vec{c} \in D_I | \vec{c} = (V_{\text{est}}, V_{\text{MRSP}}), V_{\text{est}} \neq 0 \wedge V_{\text{est}} > V_{\text{MRSP}} + \mathsf{dV}_{\mathsf{ebi}}(V_{\text{MRSP}})\} \tag{3.23}$$

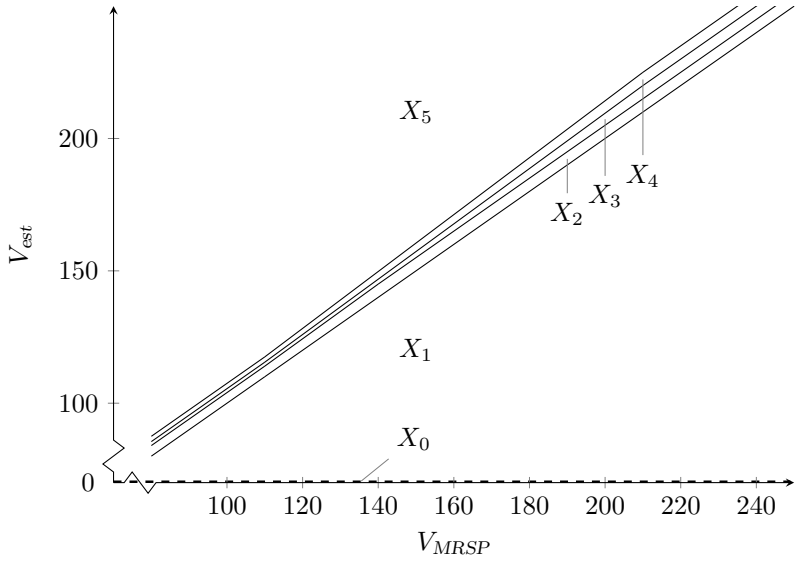


Figure 3.3: Coarsest IECP of the Ceiling Speed Monitor

3.2 A Route-Based Interlocking System Featuring Sequential Release

Rationale Section 4.2 presents an approach for MBT of route controllers. A route controller is a sub-system in an interlocking system which is responsible for the allocation, setting and release of exactly one route in a route-based interlocking system. The route controller has to adhere to the interlocking principles described in Section 2.1.2.2.

The behaviour of the route controller can be described by a SysML state machine. Section 4.2.4 presents a state-machine template that describes the behaviour of a route controller of an arbitrary route in a modern interlocking system including the feature of sequential release. Figure 4.4

shows this state-machine template. All route controllers of a specific route differ only in the operations that are used in the state-machine template. This allows for the generic behavioural description of the interlocking principles. For a specific route, the concrete behaviour can be instantiated. This will then be used for the test-case generation.

This work uses two concrete route controllers of two specific routes. The first route is a route from the railway network shown in Figure 2.1. From this network, we choose Route 7 as defined in the interlocking table in Table 2.1. This case study is denoted by Route 2011 (example network) in the remainder. This is the route from marker board mb20 to mb11. As a second case study, we choose a more complex real-world example: namely, Route 12a from the Danish train station in Lyngby. The network of this train station shown in Figure 3.4 is taken from [Vu15]. The interlocking table excerpt shown in Table 3.1 has been generated using the route-table generator from [Vu15]. We use Route 12a as our second interlocking system case study and call this route Route 12a (Lyngby) in the following.

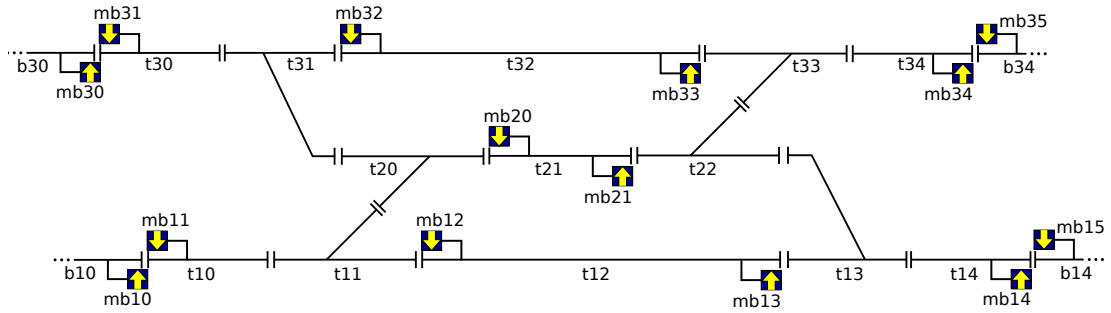


Figure 3.4: Railway Network of Lyngby Train Station in Denmark Taken from [Vu15]

id	src	dst	path	points	signals	conflicts
...						
12a	mb30	mb21	t30;t31;t20;t21	t11:p;t13:p; t20:p;t31:m; t33:p	mb20;mb31;mb32	1;2a;2b;5; 6a;6b;7a;7b; 8a;8b;9a;9b; 10a;10b;11; 12b;13;16a; 16b
...						

Table 3.1: Excerpt from the Interlocking Table of Lyngby Train Station

Both case studies have been used in previous mutation experiments published in [PHH16a]. Note that the state machine used in this work differs slightly from the models used in [PHH16a]. First, the state-machine template presented in Section 4.2.4 uses a more comprehensive approach that uses one state machine location OCCUPIED instead of several locations as used in [PHH16a]. Second, the check for safety violations has been improved to avoid spurious safety violations that might arise from the interaction of multiple route controllers.

DFSM Abstraction Although both cases studies, Route 2011 (example network) and Route 12a (Lyngby), use similar state machines, cf. Section 4.2.4, the state space of both models differs

because of different variable spaces and the different complexities of the respective routes.

The DFSM abstraction of Route 2011 (example network) consists of an input alphabet containing 31 IECs and of 8 SECs in minimised form. Compared to this, the coarsest IECP of Route 12a (Lyngby) is composed of 72 IECs and the coarsest SECP contains 17 SECs.

The difference in the size of the DFSM is mainly reasoned by the size of the routes: i.e., the number of path elements of the route. The higher the number of path elements, the more internal-state combinations in location OCCUPIED are possible.

3.3 Airbag Controller

Rationale A model from the automotive domain concludes our list of case studies. This model is clock-driven, as is the case for a variety of embedded systems. The airbag-controller model that is described by the state machine from Figure 3.5 models an embedded controller for a passenger airbag in a car. The airbag system is a crucial means to ensure passenger safety. This system needs to fulfil high availability constraints. It has to guarantee (up to a certain degree of confidence) that the airbag is activated when a crash situation is recognised by the controller. For the detection of a crash situation, acceleration sensors are used. However, accidental activation of the airbag (which might be caused by erroneous sensor data) has to be prevented, as erroneous activation of the airbag is in most cases very hazardous. Therefore, the system uses redundant sensor information to reduce the probability of unintended airbag activation by faulty sensor data. Additionally, the redundant sensor information allows for the detection of defective sensors, which is crucial to meeting the high-availability constraints of an airbag system.

Thus, the airbag controller uses two redundant acceleration sensors, indicated by the system inputs s1 and s2, to detect crash situations. Based on these inputs, the controller has to decide whether the airbag shall be fired or not (indicated by output fire), and it has to detect defect sensors. A defect of a sensor has to be indicated (by output defect); in this case, airbag firing shall be deactivated.

SysML State Machine The state machine describing the behaviour of the airbag controller is depicted in Figure 3.5. This state machine was first published in [HHP15]. The following description of the state machine has been taken with slight modifications from [HHP15]. The system reads the sensor values s1 and s2 cyclically on every rising and falling edge (input t). Both sensor values are checked for plausibility. The sensor values are considered plausible if the value of Sensor 1 (s1) does not exceed or drop below the value of Sensor 2 (s2) by more than five percent: i.e., $s1 \in [0.95 \cdot s2, 1.05 \cdot s2]$. If the sensor values are plausible and an acceleration greater than $3 \cdot g_0$[1] is measured in three consecutive cycles, the airbag is fired. This is done by setting output variable fire to one. If instead the sensor values are implausible, internal variable error_ctr is incremented. This variable holds the number of implausible measurements, and if it reaches a value equal to three, the output variable defect is set to one, causing a shutdown of the complete airbag system and activating the service lamp to indicate a sensor defect of the airbag. After at least three consecutive cycles with plausible sensor values, the internal variable error_ctr is reset.

[1] The acceleration is measured as a factor of g_0, where g_0 denotes the standard acceleration due to gravity.

DFSM Abstraction The minimal DFSM abstraction of the airbag controller contains 44 states. Figure 3.6 gives a rough idea of the DFSM.

The coarsest IECP of the airbag controller is as follows:

$$\mathcal{I}_c = \{X_0, X_1, X_2, X_3, X_4, X_5\} \tag{3.24}$$
$$X_0 = \{(\mathrm{s1}, \mathrm{s2}, \mathrm{t}) \in D_I | \mathrm{s1} > 3 \wedge \mathrm{s2} > 3 \wedge \mathrm{s1} \geq 0.95 \cdot \mathrm{s2} \wedge \mathrm{s1} \leq 1.05 \cdot \mathrm{s2} \wedge \mathrm{t} = 0\} \tag{3.25}$$
$$X_1 = \{(\mathrm{s1}, \mathrm{s2}, \mathrm{t}) \in D_I | \mathrm{s1} > 3 \wedge \mathrm{s2} > 3 \wedge \mathrm{s1} \geq 0.95 \cdot \mathrm{s2} \wedge \mathrm{s1} \leq 1.05 \cdot \mathrm{s2} \wedge \mathrm{t} = 1\} \tag{3.26}$$
$$X_2 = \{(\mathrm{s1}, \mathrm{s2}, \mathrm{t}) \in D_I | (\mathrm{s1} \leq 3 \vee \mathrm{s2} \leq 3) \wedge \mathrm{s1} \geq 0.95 \cdot \mathrm{s2} \wedge \mathrm{s1} \leq 1.05 \cdot \mathrm{s2} \wedge \mathrm{t} = 0\} \tag{3.27}$$
$$X_3 = \{(\mathrm{s1}, \mathrm{s2}, \mathrm{t}) \in D_I | (\mathrm{s1} \leq 3 \vee \mathrm{s2} \leq 3) \wedge \mathrm{s1} \geq 0.95 \cdot \mathrm{s2} \wedge \mathrm{s1} \leq 1.05 \cdot \mathrm{s2} \wedge \mathrm{t} = 1\} \tag{3.28}$$
$$X_4 = \{(\mathrm{s1}, \mathrm{s2}, \mathrm{t}) \in D_I | (\mathrm{s1} < 0.95 \cdot \mathrm{s2} \vee \mathrm{s1} > 1.05 \cdot \mathrm{s2}) \wedge \mathrm{t} = 0\} \tag{3.29}$$
$$X_5 = \{(\mathrm{s1}, \mathrm{s2}, \mathrm{t}) \in D_I | (\mathrm{s1} < 0.95 \cdot \mathrm{s2} \vee \mathrm{s1} > 1.05 \cdot \mathrm{s2}) \wedge \mathrm{t} = 1\}. \tag{3.30}$$

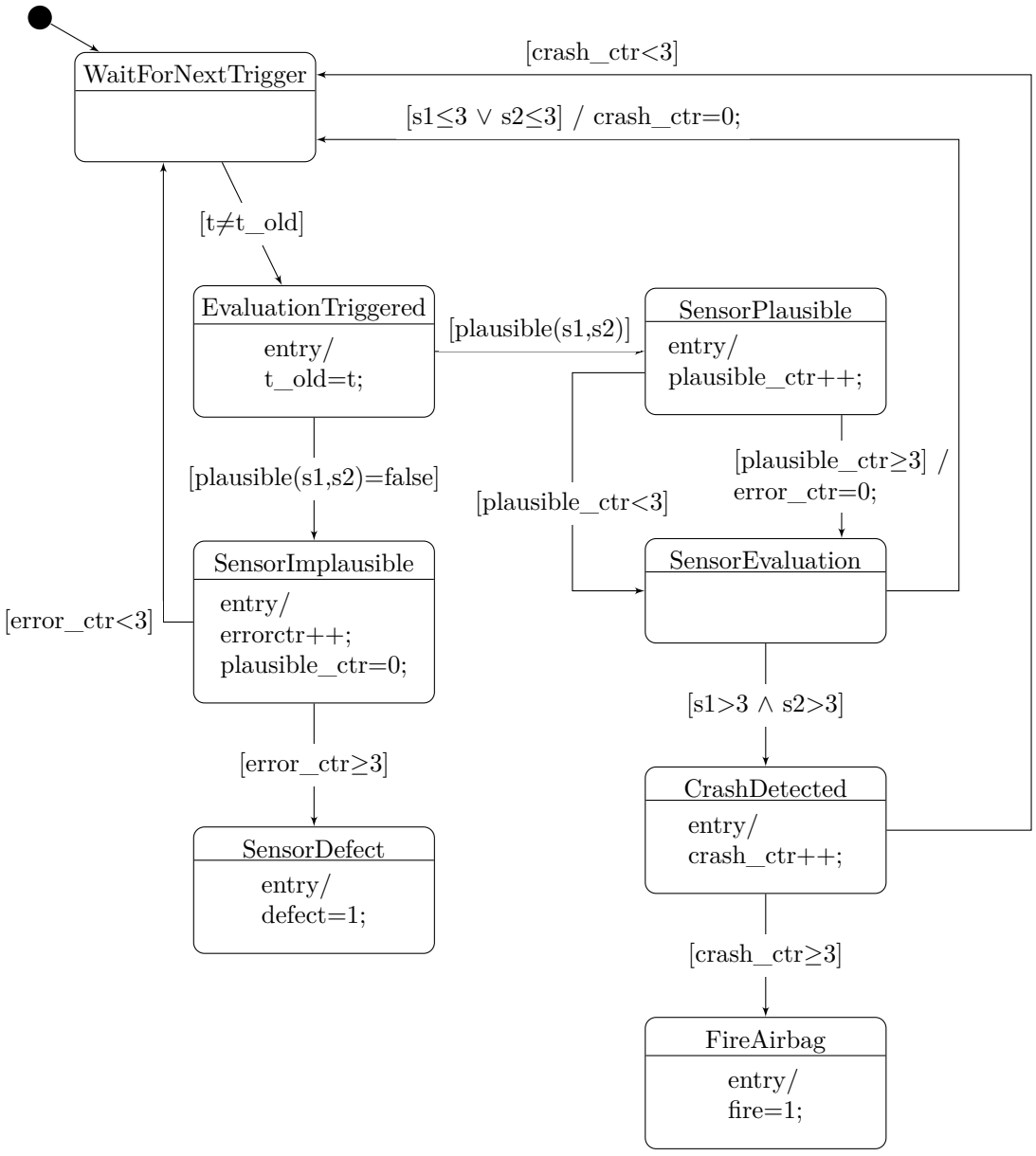


Figure 3.5: State Machine of the Airbag Controller.

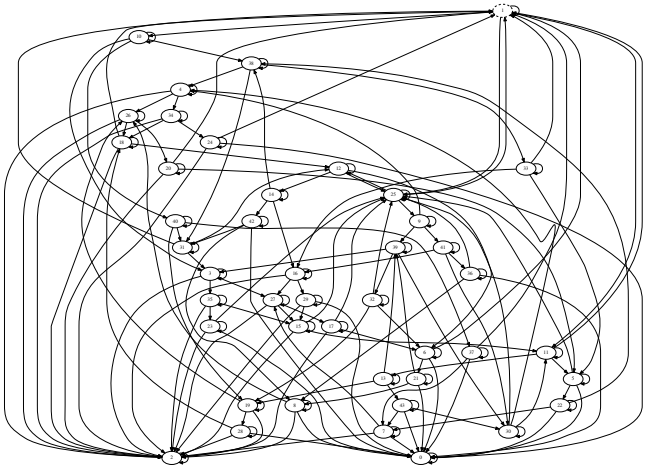


Figure 3.6: Sketch of the DFSM Abstraction of the Airbag Controller

4 Testing Methodology

4.1 Overview

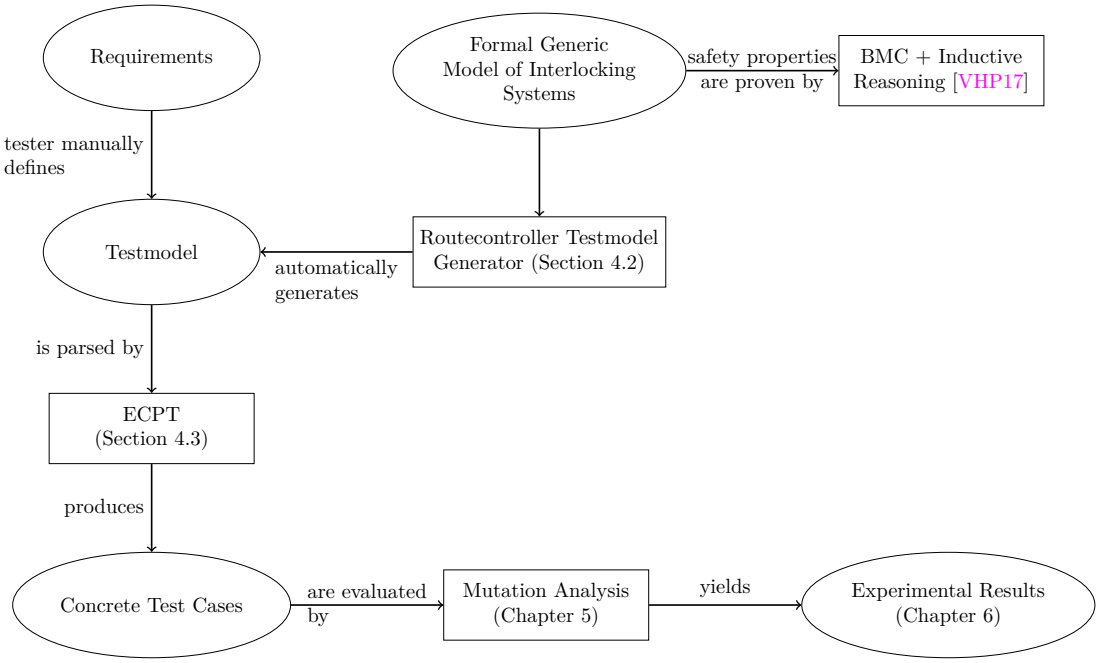


Figure 4.1: Illustration of the Testing Methodology

Figure 4.1 illustrates the context of our proposed testing methodology. For the verification of railway systems, we propose the use of MBT—especially the ECPT approach introduced in Section 2.5. The test model needed for our ECPT will usually be derived from requirements by a test engineer. However, for the special case of formal modelling of interlocking systems, we propose an approach that is based on a formal generic model of modern route-based interlocking systems. This generic model, which can be used to derive a concrete formal model of an interlocking system of a concrete railway network, is based on the work of [VHP17]. This formal model, originally used for the proof of safety properties, can be used to automatically derive a test model for HSI tests of route-controller components of the interlocking system. This will be described in detail in the next section. The result of this step is a concrete test model describing the behaviour of a concrete route controller as in the case studies of Route 2011 (example network) and Route 12a (Lyngby) previously presented in Chapter 3.

Regardless of how a test model is derived, whether from requirements or in case of interlocking

systems from a generic formal model, test-case generation is performed using our implementation of the ECPT approach. Section 4.3 details our implementation and extensions of the approach based on the fundamentals first laid out in [HP13]. The extensions are used to ensure that the approach is applicable to SUTs in the context of black-box tests where membership in a formal fault domain can neither be proven nor expected. To prove that our approach is able to detect faults with a high probability—especially in the context of HSI and system tests—we present a novel mutation-analysis approach in Chapter 5. The experimental results obtained by this approach are given in Chapter 6.

4.2 Formal Modelling of Interlocking Systems

[VHP17] presents an approach to the modelling and verification of modern route-based interlocking systems (including the feature of sequential release). In this work, interlocking systems are described by their *generic behaviour* (expressed in an DSL), which is independent of the concrete railway network under consideration. This generic behaviour formally describes the interlocking principles (cf. Section 2.1.2.2) that are used by concrete interlocking systems. The concrete railway network and interlocking tables are considered *configuration data*, which is described in another DSL. Combining generic behaviour and configuration data, the behaviour of a concrete interlocking system is instantiated, resulting in a formal description of concrete interlocking system behaviour. The resulting system can be verified using formal methods. The authors in [VHP17] perform an approach called k-induction that is based on bounded model checking. This approach allows for the verification of safety properties. The absence of safety hazards in the concrete formal model can be proven automatically.

The advantage of the approach proposed in [VHP17] is that, once a generic behavioural model is specified, it can be instantiated with arbitrary configuration data. This clearly reduces the modelling effort for the specification of interlocking systems. In our approach, we want to use this approach with a generic behavioural model of interlocking systems to automatically extract a formal model of route-controller behaviours from configuration data. This generated model of a route-controller behaviour can, in turn, be used for MBT.

4.2.1 Compositional Reasoning Applied to Interlocking Systems

As described in Section 2.1.2. an interlocking system is responsible for ensuring the safe operation of trains traversing the railway network over pre-defined routes. In Section 2.1.2.2, we described the operational principles for a single route in the interlocking system. Because every route in a route-based interlocking system performs the same generic behaviour, it seems natural that the implementation of an interlocking system is, among other things, comprised of separate components for each single route. Such a controlling component, which is responsible for a single route in an interlocking system, will be called *route-controller component* (or *route controller* in abbreviated form) in the remainder of this manuscript.

For the MBT of interlocking systems, it seems promising to first verify the correctness of each route controller in isolation. Given the correctness of the single route controllers, the correctness of the overall system is verified afterwards. This imposes a natural integration test level for interlocking systems.

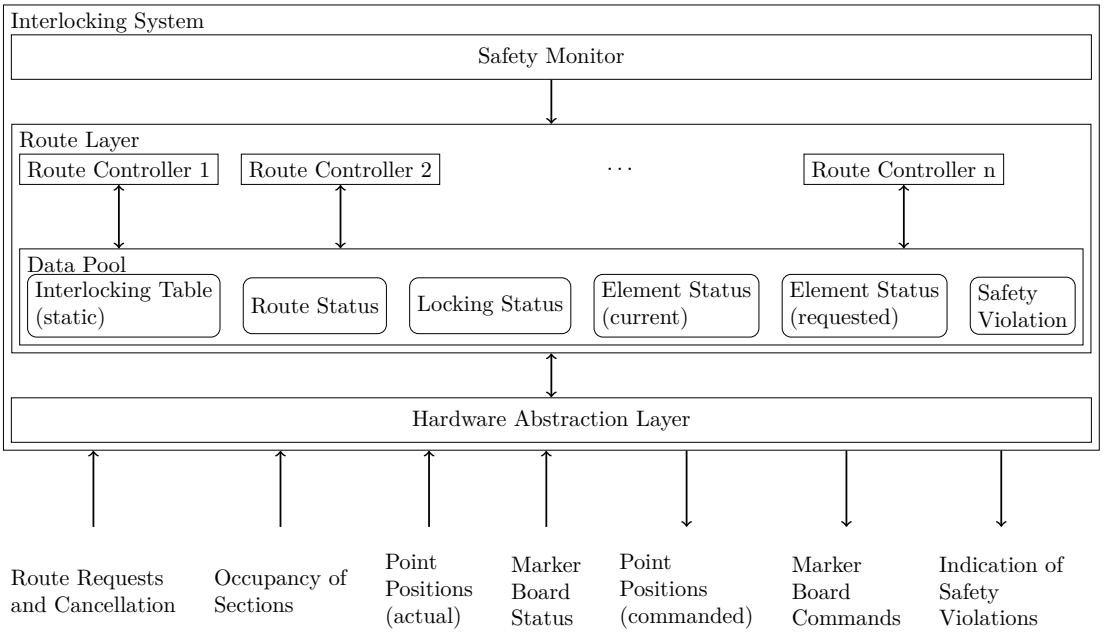


Figure 4.2: Interlocking System, Interfaces and Internal Structure

The technique of *compositional reasoning* is well known in model checking (see [BCC97]). It is one of the main techniques used to mitigate the state-explosion problem. In *compositional reasoning*, the system's components $C_1, \ldots, C_n$ are considered in isolation. From the properties of the sub-components, properties of the composite system can be inferred (automatically). If the specification of the composite system can be inferred completely from the specification of the components and how these components interact, the system is *compositional*. In this case, the correctness of the overall system can be verified by verifying all sub-components in isolation.

In this work, we assume that interlocking systems composed of route-controller components are compositional. Note that several design principles and technical measures are used to enforce compositionality. Nonetheless, compositionality is hard to prove. Therefore, standards for the verification of safety-critical systems [Eur01, RTC92, ECS09] usually require tests at different integration levels. In particular, the fully integrated system has to be verified by system tests. For the application of MBT to route controllers, we assume that the interlocking system architecture is similar to the architecture shown in Figure 4.2.

The interlocking system is composed of different route-controller components $1 \ldots n$. Every route-controller component is responsible for exactly one route, as specified in the interlocking table. All route-controller components share the same data from the data pool. The route-controller components together with the data pool form the *route layer*, which is responsible for the allocation of a requested route, prevention of simultaneous allocation of conflicting routes, sequential release and cancelling of routes. To do all of this, it requires several inputs from the system environment: route request and cancellation commands, occupancy states, point positions and marker-board states are inputs to the interlocking system. The system in turn outputs point-position commands and marker-board commands. In case some safety condition is violated, this fact is communicated through a special output indicating safety violations. Interlocking systems

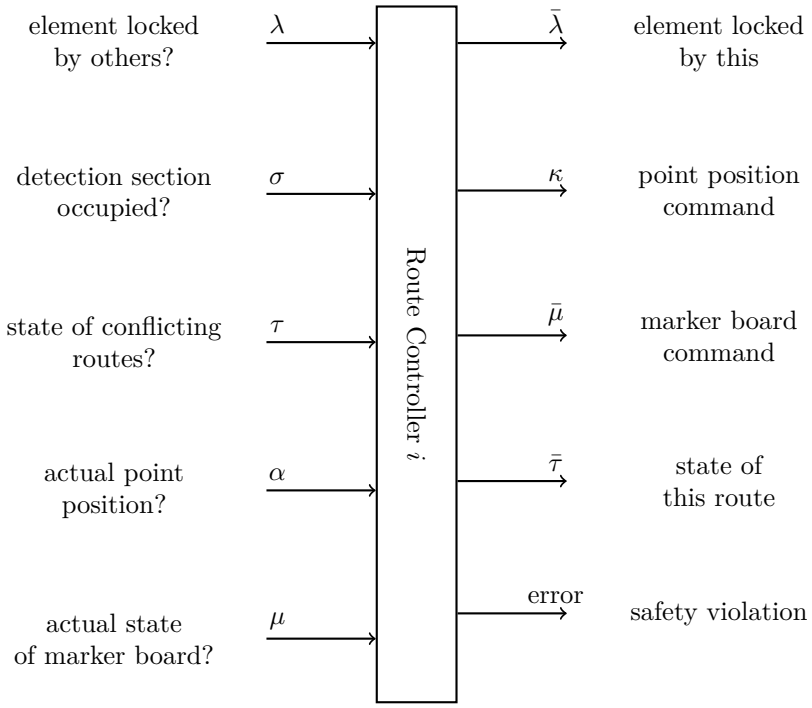


Figure 4.3: Route Controller Sub-component Interfaces

are highly distributed systems. Therefore, a special layer called *hardware abstraction layer (HAL)* is used to process the network interfaces of the system. The HAL receives route requests and cancellation commands, occupancy states and point and marker-board status updates. The received input is written into the data pool to be processed by the route-controller components. Every route-controller component processes route requests, cancellation commands and element states from the data pool and updates its route state and track-side element commands in the data pool. Following this, the HAL reads the data pool updates and sends point position and marker-board commands to its output interface. Every route controller supervises some local safety conditions; for example, it detects unintended switches of points on the route or illegal train movements. In this case, a safety violation is output and the system transitions to a fail-safe state (all marker boards are set to HALT).

Moreover, an additional safety monitor is needed, because some safety conditions cannot be supervised locally by a route controller. For example, train movements in elements currently not locked by any route have to be detected on a higher-level layer. Other safety conditions to be supervised on a system level include the following: Elements must not be locked by more than one route-controller component at a time, different route controllers must not give contradicting commands to the same element, points that are not locked must not be commanded to switch their positions, and marker boards must not be commanded to PASS unless the route controller of a route starting at the marker board is in the LOCKED state.

4.2.2 Route-Controller Component Interfaces

Figure 4.3 gives an overview of the input and output interfaces of route-controller components. The inputs to the route controller are defined by the functions λ, σ, τ, α and μ. The outputs of the route controller are described by the functions $\bar{\lambda}$, κ and $\bar{\mu}$. These outputs are complemented by an output variable for the state of the route-controller component $\bar{\tau}$ and output variable error indicating that a local safety condition of this route controller is violated. Note that we do not list all input I and output variables O in the remainder but instead use functions to describe the mapping of variables from I and O to their values. Thus, the functions described in the remainder of this section can be considered a set of input/output variables. The definitions of guard conditions and effects, which are given below, describe how these variables are evaluated or updated, respectively.

Let E denote the set of all path elements of the route under consideration. A path element $e \in E$ is by definition a detection section, which is a point or a linear section. Let $P_p \subset E$ denote the *path points*: i.e., the subset of path elements of type point. Let P_f denote the set of protecting points of the route providing flank protection. Path points and protecting points are collectively denoted $P = P_p \cup P_f$.

$\lambda : E \cup P_f \to \mathbb{B}$ is a function mapping all path elements and protecting points to a boolean value that is true iff the element is locked by any other route.

$\sigma : E \to \mathbb{B}$ is a function mapping the path elements to their occupancy status: a boolean value that is true iff the detection section is occupied.

$\tau : C \to \{\text{FREE,MARKED,ALLOCATING,LOCKED,OCCUPIED,ERROR}\}$ maps every conflicting route from the set of conflicting routes C to its current state.

$\alpha : P \to \{\text{PLUS}, \text{MINUS}\} \cup \{\perp\}$ maps every point to its current position and $\beta : P \to \{\text{PLUS}, \text{MINUS}\}$ maps every point to the position that is requested for the route according to the route table column *points*.

$\bar{\lambda} : E \mapsto \mathbb{B}$ is a function mapping path elements to true iff the path element is currently locked by *this* route.

$\kappa : P \to \{\text{PLUS}, \text{MINUS}\} \cup \{\perp\}$ maps a point p to a value from $\{\text{PLUS}, \text{MINUS}\}$ if this route currently commands a concrete position for p. $\kappa(p) = \perp$ indicates that this route currently does not command any position for point p.

M denotes the set of marker boards that are relevant to the route under consideration. This includes the protecting marker boards $S \subset M$ listed in column *signals* used for flank and front protection and the source (m_{src}) and destination signal (m_{dest}) of the route.

Input function $\mu : S \to \{\text{PASS}, \text{HALT}\} \cup \{\perp\}$ maps every protecting marker board $m \in S$ to its current status. Output function $\bar{\mu} : M \to \{\text{PASS}, \text{HALT}\} \cup \{\perp\}$ maps marker board m to a value from $\{\text{PASS}, \text{HALT}\}$ if this marker board is currently commanded a specific signal aspect by this route. $\bar{\mu}(m) = \perp$ indicates that this route currently does not command any signal aspect for m.

4.2.3 Generic Behaviour of Route Controllers

The generic behaviour of route controllers was informally introduced in Section 2.1.2.2. This generic behaviour can be described by a state-machine template, as depicted in Figure 4.4. The grey parts in Figure 4.4 are dependant on the concrete configuration data. The behaviour of these parts is explained below.

4.2.4 Extraction of Local Behavior

The predicate no_conflict is defined as follows:

$$\text{no\_conflict} \triangleq \bigwedge_{e \in E} (\lambda(e) = \text{false} \wedge \sigma(e) = \text{false}) \wedge \tag{4.1}$$

$$\bigwedge_{c \in C} (\tau(c) \notin \{\text{ALLOCATING}, \text{LOCKED}\}) \wedge \tag{4.2}$$

$$\bigwedge_{p_f \in P_f} (\lambda(p_f) = \text{false} \vee \alpha(p_f) = \beta(p_f)). \tag{4.3}$$

No conflict for the route exists if all path elements are vacant and not locked by any other route, if none of the conflicting routes is in state ALLOCATING or LOCKED and if all protecting points are either not locked (and can therefore be switched to the requested position) or the current position of these protecting points is already the requested position.

When entering the location ALLOCATING, the update functions lock_path_elements, command_points and command_mbs are called.

The effect of lock_path_elements (denoted by Σ(lock_path_elements)) is as follows:

$$\Sigma(\text{lock\_path\_elements}) : \bar{\lambda} := \bar{\lambda} \oplus \{e \mapsto \text{true} | e \in E\}. \tag{4.4}$$

$f \oplus g$ denotes the update of function f by partial function g: $f \oplus g(x)$ is $g(x)$ if $x \in \text{dom}(g)$ and $f(x)$ otherwise. The effect of lock_path_elements is that all path elements are locked by this route.

$$\Sigma(\text{command\_points}) : \kappa := \kappa \oplus \{p \mapsto \beta(p) | p \in P\} \tag{4.5}$$

The effect of command_points is that every point from the interlocking table column *points* for this route is commanded to its required position.

$$\Sigma(\text{command\_mbs}) : \bar{\mu} := \bar{\mu} \oplus \{m \mapsto \text{HALT} | m \in S\} \tag{4.6}$$

The effect of command_mbs is that all protecting signals are commanded to the signal aspect HALT.

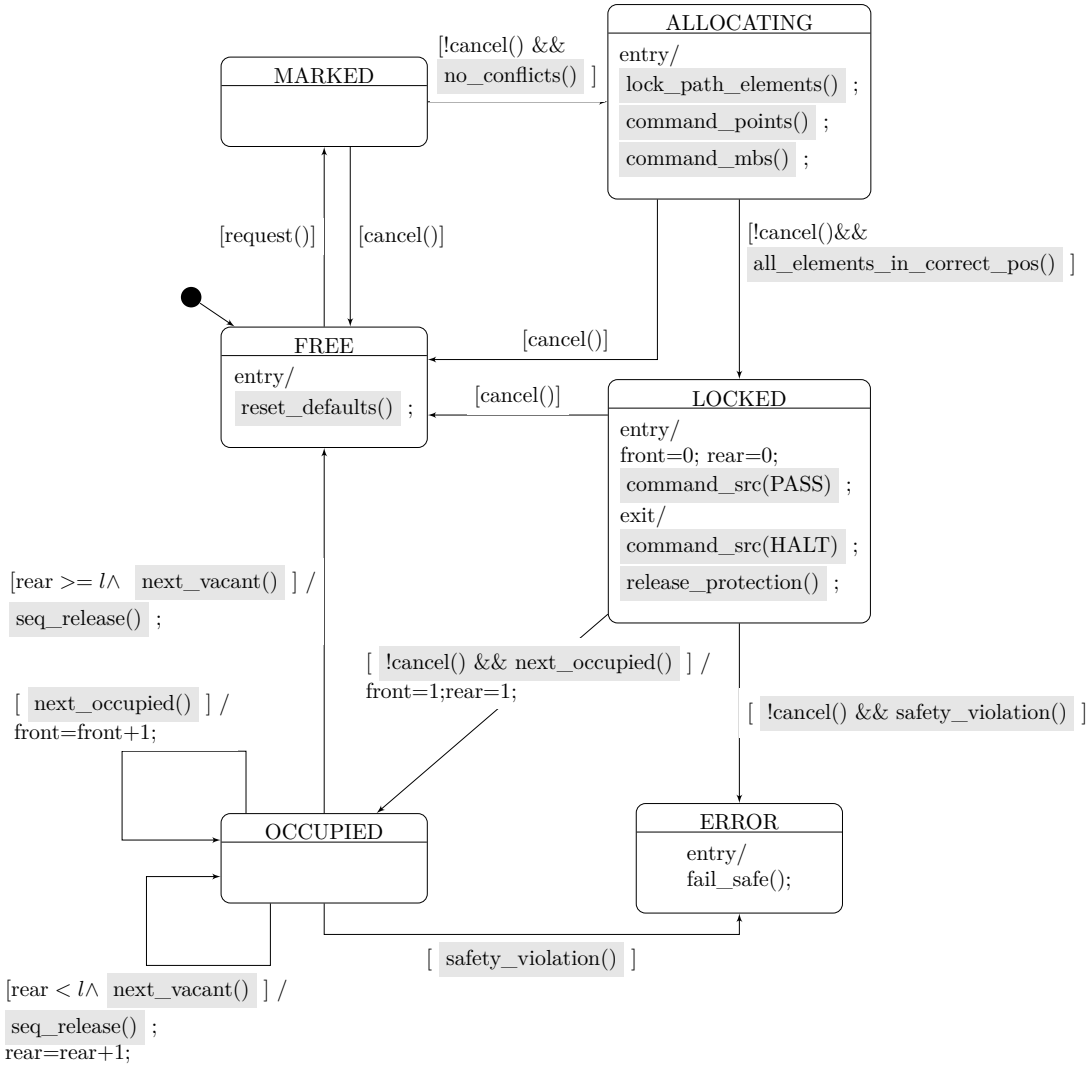


Figure 4.4: State Machine Template for Route Controllers

The predicate all_elements_in_correct_pos is defined as follows:

$$\text{all\_elements\_in\_correct\_pos} \triangleq \bigwedge_{p \in P} (\alpha(p) = \beta(p)) \wedge \tag{4.7}$$

$$\bigwedge_{m \in S} (\mu(m) = \text{HALT}). \tag{4.8}$$

all_elements_in_correct_pos is true if all points of table column *points* for the considered route are in the correct position and all protecting marker boards show the aspect HALT.

$$\Sigma(\text{command\_src}(a)) : \bar{\mu} := \bar{\mu} \oplus \{m_{\text{src}} \mapsto a\} \tag{4.9}$$

The effect of operation command_src for a given signal aspect $a \in \{\text{PASS}, \text{HALT}\}$ is that the source signal of this route is commanded to show the signal aspect a.

$$\Sigma(\text{release\_protection}) : \kappa := \kappa \oplus \{p_f \mapsto \bot | p_f \in P_f\}, \tag{4.10}$$

$$\bar{\mu} := \bar{\mu} \oplus \{m \mapsto \bot | m \in S\} \tag{4.11}$$

Operation release_protection suspends all commands to protecting signals and protecting points.

Location OCCUPIED models all legal train movements and the sequential release of freed elements. Again, the grey parts of the template are dependant on configuration data from the interlocking table. The following paragraphs describe these route-specific parts.

Let e_1, e_2, …, e_l be the path elements of the route in the order these elements are traversed on the route. In location OCCUPIED, the variable front shall contain the maximum i fulfilling $\sigma(e_i) = \mathit{true}$. That is, front describes the index of the path element the front of the train resides in. Note that an i with $\sigma(e_i) = \mathit{true}$ always exists as long as the route controller resides in location OCCUPIED. Accordingly, the variable rear shall hold the minimal index i fulfilling $\sigma(e_i) = \mathit{true}$. Hence, rear is the index of the path element containing the rear of the train or the index of the first element of the route if the train has not yet entered the route completely. If the front of train has not yet entered the route (location LOCKED), front = rear = 0. The value of variable front is updated as soon as next_occupied becomes true.

$$\text{next\_occupied} \triangleq (\text{front} < l \wedge \sigma(e_{\text{front}+1}) = \mathit{true}) \wedge \neg\text{safety\_violation} \tag{4.12}$$

Operation next_occupied can be defined as the check of whether the next path element ($e_{\text{front}+1}$) neighbouring the current path element addressed by front (e_{front}) is occupied. This check is only to be performed if no safety violation exists. The check for safety violations is described below.

$$\text{next\_vacant} \triangleq (\sigma(e_{\text{rear}}) = \text{false}) \wedge \neg\text{safety\_violation} \wedge \neg\text{next\_occupied} \tag{4.13}$$

Variable rear is updated as soon as the path element that has contained the trains rear so far becomes vacant and if no safety violation is present. The term ¬next_occupied is included to make the state machine deterministic.

$$\Sigma(\text{seq\_release}) : \begin{cases} \bar{\lambda} := \bar{\lambda} \oplus \{e_{\text{rear}} \mapsto \text{false}\}, \kappa := \kappa \oplus \{e_{\text{rear}} \mapsto \bot\} & \text{if } e_{\text{rear}} \in P \\ \bar{\lambda} := \bar{\lambda} \oplus \{e_{\text{rear}} \mapsto \text{false}\} & \text{else} \end{cases} \tag{4.14}$$

Operation seq_release implements the sequential release of elements that have just become vacant. Note that the call of seq_release and the update of rear are performed in an action of one transition. According to the UML semantics, actions are performed atomically. Thus, e_{rear} in the equation above is the path element which has just been freed. Operation seq_release causes the route controller to release the lock on e_{rear}; in addition, the position commands of released points are suspended.

$$\text{safety\_violation} \triangleq \bigvee_{i \in [\text{front}+2, l]} (\sigma(e_i) = \text{true}) \vee \tag{4.15}$$

$$\bigvee_{i \in [\text{rear}+1, \text{front}]} (\sigma(e_i) = \text{false}) \vee \tag{4.16}$$

$$(\text{front} = \text{rear} \wedge \text{rear} > 0 \wedge \text{rear} < l \wedge \sigma(e_{\text{rear}}) = \text{false}) \vee \tag{4.17}$$

$$\bigvee_{p \in P_p} (\bar{\lambda}(p) = \text{true} \wedge \alpha(p) \neq \beta(p)) \tag{4.18}$$

Safety violations due to unexpected train movements are noticed by operation safety_violation. This operation is based on assumptions of valid train movements. The assumptions are inspired by the "rubber-band model"[AT12] for the implicit modelling of train movements. It is assumed that no sudden jumps of the front or the rear of the train can happen, no holes in the train are allowed, and the train is not allowed to change its direction on the route. Equation 4.15 checks for sudden jumps of the front of the train, while Equation 4.16 checks for safety violations that may result from holes in the train, sudden jumps of the rear or a train rolling back. Equation 4.17 evaluates to true if the train vanishes. If the front and rear of the train reside in one detection section but not the last detection section, the vacancy status of this section is not allowed to change to vacant before the front has entered the next detection section. Furthermore, a safety violation can be triggered by unintended point switches (Equation 4.18). If a point of the route switches its position while it is still locked, this is considered a safety violation.

$$\Sigma(\text{reset\_defaults}) : \bar{\lambda} := \bar{\lambda} \oplus \{e \mapsto \text{false} | e \in E\}, \tag{4.19}$$

$$\kappa := \kappa \oplus \{p \mapsto \bot | p \in P\} \tag{4.20}$$

$$\bar{\mu} := \bar{\mu} \oplus \{m \mapsto \bot | m \in S \vee m \in \{m_{\text{src}}, m_{\text{dest}}\}\} \tag{4.21}$$

The effect of reset_defaults is that all path elements of the route are unlocked (released) and all commands to points and marker boards are suspended. When entering location FREE from location OCCUPIED the only change in $\bar{\lambda}$ is the locking state of the last path element. All other elements have been released before due to sequential release.

$$\Sigma(\text{fail\_safe}) : \text{error} := \text{true}, \bar{\mu} := \bar{\mu} \oplus \{m \mapsto \text{HALT} | m \in S \vee m \in \{m_{\text{src}}, m_{\text{dest}}\}\} \tag{4.22}$$

Operation fail_safe sets output variable error and commands every marker board to HALT.

Output variable $\bar{\tau}$ is implicitly updated to a value from {FREE,MARKED,ALLOCATING, LOCKED,OCCUPIED,ERROR} whenever the state machine location changes.

4.3 Equivalence Class Partition Testing

The testing approach we present and evaluate in this work is based on the ECPT approach introduced in Section 2.5. This approach, whose workflow is illustrated in Figure 4.5, starts from a given formal model of an SUT. In our case, this model is specified as a SysML model. The behavioural part of the model is defined by state machines. The semantics of these state machines, as introduced in Section 2.2.1, can be expressed by RIOSTSs. For our approach, we require the model and output variables to have a finite domain.

Starting from the intermediate model representation as a RIOSTS, we apply the DFSM abstraction. The final DFSM is used for the generation of abstract test cases on DFSM level. Therefore, the W or Wp-method is applied. From the resulting abstract test cases, concrete test cases are derived by selection of concrete input vectors from the IECs. Concrete inputs of an IEC X are calculated by solving the predicate Φ_X that describes X with a Satisfiability Modulo Theories (SMT) solver. Following the original approach, as proposed in [HP16a], one fixed value for each IEC is selected. However, first experiments have shown that this approach leads to relatively low fault-detection capabilities when applied to SUTs that are outside the fault domain of our approach. A solution to this would be an increase of the fault domain, to finally ensure that the SUT is in the fault domain. The fault domain can be increased by refining the IECP used for test-case generation. This measure is described in Section 4.3.1. Unfortunately, the refinement of an IECP results in additional test cases. Therefore, Section 4.3.2 presents an alternative approach that is based on a randomisation of the concrete input selection from IECs. By random selection of multiple values from an IEC instead of one fixed representative, the test strength is increased, as is demonstrated by our experiments (see Chapter 6). Combined with boundary-value selection (Section 4.3.3), the test strength can be further improved. This results in a heuristic, which is denoted by STRAT-3 in our experiment and which we propose as a best practice for the use of the ECPT approach. This heuristic allows for a significant increase of test strength without increasing the number of test cases or test steps. Section 4.3.4 discusses how to detect SUT errors that are not covered by our fault domain because of additional states in the SUT. While these can naturally be covered by increasing parameter m in the Wp/W-method, the exponential explosion of the number of test cases resulting from increasing m has to be avoided. Therefore, we propose heuristics that aim at the exploration of additional states by longer test sequences but without increasing the number of test cases.

The whole workflow, as illustrated in Figure 4.5, has been implemented by the author as part of RTT-MBT: the MBT component of the test automation tool RT-Tester developed by Verified Systems International GmbH[1]. The workflow is fully automated in our tool, and several command line options make it possible to configure test-case generation: e.g., use of W/Wp-method,

[1] `https://www.verified.de`

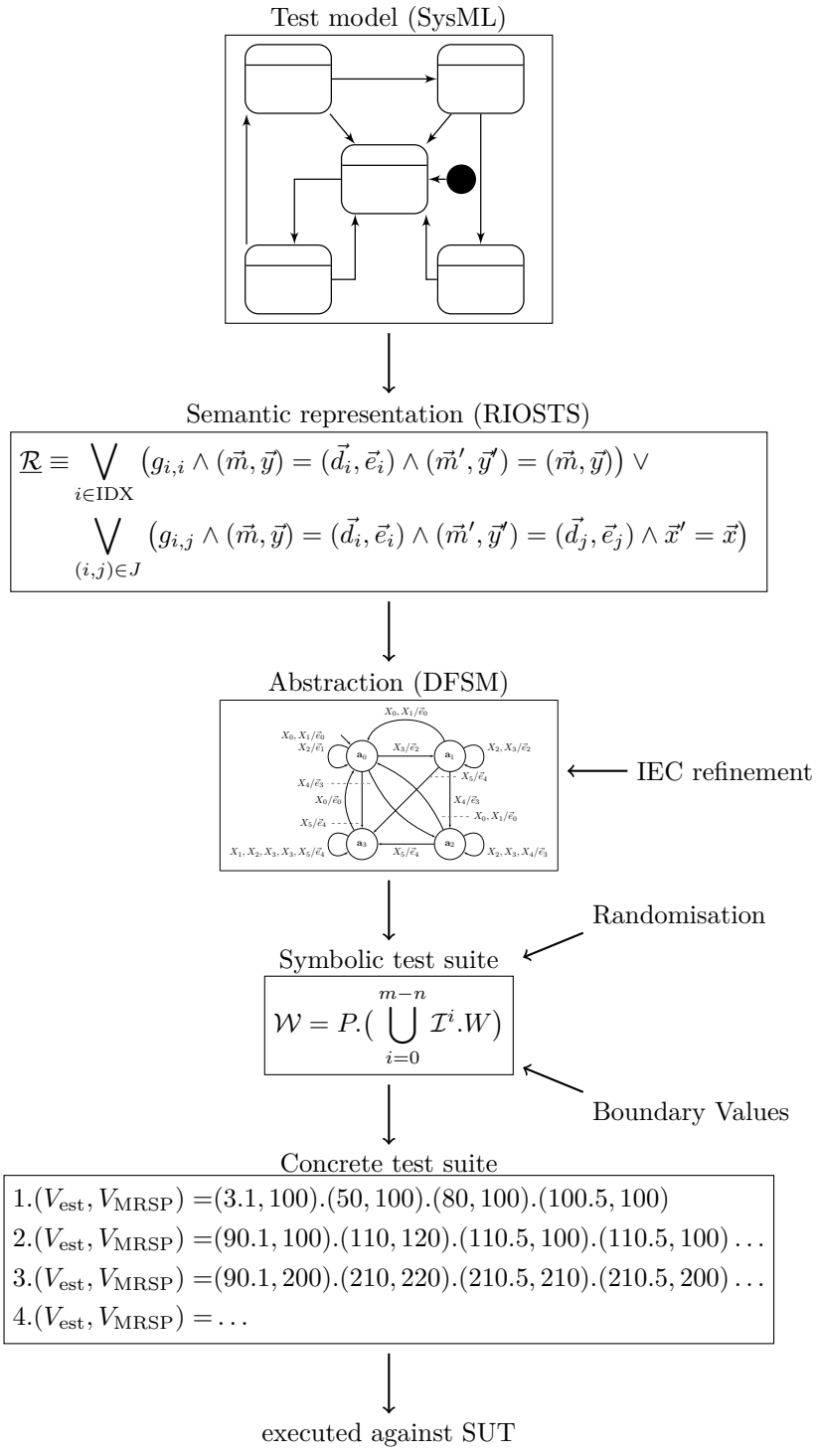


Figure 4.5: Workflow of the ECPT Approach

enable/disable use of boundary values, enable random test generation, specify target language (i.e., JUnit or SystemC tests). Some implementation considerations are detailed in Section 4.3.5. We think that these are necessary to guarantee that the test generation scales up for complex systems.

4.3.1 Refinement of Input Equivalence Classes

As shown in Section 2.5.3, the fault domain of our testing approach depends on the IECP that is used as an input alphabet for the DFSM. The coarsest IECP $\mathcal{I}_c$ surely results in a smaller fault domain than any refined IECP $\mathcal{I} \subseteq \mathcal{I}_c$. In the following, we present approaches to refine the IECP $\mathcal{I}_c$ to obtain a greater fault domain and in turn a better test strength. Note that every refinement of the IECP $\mathcal{I}$ used for test-case generation results in an increase in the number of test cases. This is because the transition cover used by the W/Wp-method and the number of subsequences used for the exploration of additional states is increased if the input alphabet Σ_I of a DFSM is increased.

4.3.1.1 Interval Bisection

An approach to refining the IECP is to use interval bisection. Every equivalence class partitioning can always be refined by systematically dividing the input domain. The bisection operator introduced in Section 4.3.2.3 can be used. This operator iteratively bisects D_I. Therefore, the input domain is considered a box: i.e., an interval vector. The bisection splits a box to two child boxes. Each resulting child box is bisected again in the next iteration. n iterations result in 2^n subboxes of D_I of equal size. These 2^n boxes partition D_I to a partitioning denoted by $D_I/_{2^n}$. To obtain a refined IECP $\mathcal{I}$ from the coarsest IECP $\mathcal{I}_c$ of the SUT using interval bisection, both partitions $D_I/_{2^n}$ and $\mathcal{I}_c$ have to be intersected. Every non-empty intersection of classes from $D_I/_{2^n}$ and $\mathcal{I}_c$ becomes a member of the IECP $\mathcal{I}$ we are looking for.

$$\mathcal{I} = \{X_b \cap X_i | X_b \in D_I/_{2^n}, X_i \in \mathcal{I}_c, X_b \cap X_i \neq \emptyset\} \tag{4.23}$$

Interval bisection offers a means to systematically refine an IECP. This way, an IECP can be refined without limit (in case of a finite D_I the bisection terminates as soon as all refined IECs are singletons); thus, the fault domain can be increased without limit. This ensures that the probability that the SUT IECP fulfills Equation 2.78 converges to 100 percent. Of course, this drastically increases the testing effort. The number of subboxes increases exponentially with every bisection, and the number of test cases generated by the W/Wp-method increases exponentially by the number of IECs to the power of the number of additional states a. Thus, the number of test cases in relation to the number of bisections n grows with $\mathcal{O}((2^n)^a)$.

4.3.1.2 Requirement-Based Refinement

Interval bisection is an approach that can be fully automated and that makes it possible to get arbitrarily close to a refined IECP that fulfills Equation 2.78. However, interval bisection is somehow blind and usually is infeasible because of the exponential complexity.

An alternative to IECP refinement is an approach that we call requirement-based refinement. Usually, the requirements of an SUT are more restrictive than the coarsest IECP that is generated from a test model. We therefore propose to refine the coarsest IECP $\mathcal{I}_c$ by using explicit case distinctions from the requirements. To this end, we suppose that it is possible to define predicates over input variables for these case distinctions $C = \{\varphi_1, \ldots, \varphi_n\}$.

A requirement-based refinement $\mathcal{I} \subseteq \mathcal{I}_c$ can then be calculated as follows:

$$\mathcal{I} = \{\{\vec{c} \in X | \bigwedge_{\varphi \in c} \vec{c} \models \varphi \wedge \bigwedge_{\varphi' \in C \setminus c} \vec{c} \not\models \varphi'\} | X \in \mathcal{I}_c, c \subset C\}. \tag{4.24}$$

Intuitively, all inputs from an IEC of $\mathcal{I}_c$ fulfilling the same predicates from C are collected in the same refined IEC.

Example 20. Consider, for example, the CSM case study (Section 3.1). The CSM requirements specification [UNI12] contains different requirements describing the condition for a transition to location WARNING. The current train speed V_{est} has to exceed the allowed speed V_{MRSP} at least by a V_{MRSP} dependent threshold $\mathsf{dV}_{\mathsf{warning}}(V_{\text{MRSP}})$. This threshold can be described by three different requirements (cf. Equation 3.1):

1. if $V_{\text{MRSP}} \leq 110$, the threshold is 4;
2. if $V_{\text{MRSP}} \in [110, 140]$, the threshold is $\frac{V_{\text{MRSP}}}{30} + \frac{1}{3}$; and
3. if $V_{\text{MRSP}} > 140$, the threshold is 5.

These three case distinctions are not reflected by the IECP $\mathcal{I}_c$ shown in Equation 3.14. There, all inputs causing a transition to location WARNING are condensed in the IEC X_3:

$$\begin{aligned} X_3 = &\{\vec{c} \in D_I | \vec{c} = (V_{\text{est}}, V_{\text{MRSP}}), V_{\text{est}} \neq 0 \wedge V_{\text{est}} > V_{\text{MRSP}} + \mathsf{dV}_{\mathsf{warning}}(V_{\text{MRSP}}) && (4.25)\\ &\wedge V_{\text{est}} \leq V_{\text{MRSP}} + \mathsf{dV}_{\mathsf{sbi}}(V_{\text{MRSP}})\}. && (4.26)\end{aligned}$$

Selection of one fixed representative from this class, e.g., $\vec{c} = (V_{\text{est}}, V_{\text{MRSP}}) = (104.2, 100)$ leads to an input that will not be able to detect errors in the implementation in all but one of the aforementioned requirements.

If we use $C = \{V_{\text{MRSP}} \leq 110, V_{\text{MRSP}} \leq 140\}$, we get three refined IECs from X_3:

$$\begin{aligned} X_{3_a} = &\{\vec{c} \in D_I | \vec{c} = (V_{\text{est}}, V_{\text{MRSP}}), V_{\text{est}} \neq 0 \wedge V_{\text{est}} > V_{\text{MRSP}} + \mathsf{dV}_{\mathsf{warning}}(V_{\text{MRSP}}) && (4.27)\\ &\wedge V_{\text{est}} \leq V_{\text{MRSP}} + \mathsf{dV}_{\mathsf{sbi}}(V_{\text{MRSP}}) \wedge V_{\text{MRSP}} \leq 110\} && (4.28)\\ X_{3_b} = &\{\vec{c} \in D_I | \vec{c} = (V_{\text{est}}, V_{\text{MRSP}}), V_{\text{est}} \neq 0 \wedge V_{\text{est}} > V_{\text{MRSP}} + \mathsf{dV}_{\mathsf{warning}}(V_{\text{MRSP}}) && (4.29)\\ &\wedge V_{\text{est}} \leq V_{\text{MRSP}} + \mathsf{dV}_{\mathsf{sbi}}(V_{\text{MRSP}}) \wedge V_{\text{MRSP}} > 110 \wedge V_{\text{MRSP}} \leq 140\} && (4.30)\\ X_{3_c} = &\{\vec{c} \in D_I | \vec{c} = (V_{\text{est}}, V_{\text{MRSP}}), V_{\text{est}} \neq 0 \wedge V_{\text{est}} > V_{\text{MRSP}} + \mathsf{dV}_{\mathsf{warning}}(V_{\text{MRSP}}) && (4.31)\\ &\wedge V_{\text{est}} \leq V_{\text{MRSP}} + \mathsf{dV}_{\mathsf{sbi}}(V_{\text{MRSP}}) \wedge V_{\text{MRSP}} > 140\}. && (4.32)\end{aligned}$$

The refined IECs are suitable to check all three case distinctions from the requirements related to transitions to the WARNING location.

Note that the requirement-based refinement most likely needs a kind of user interaction. The predicates $C = \{\varphi_1, \ldots, \varphi_n\}$ manually have to be extracted from the specification, as requirements are usually semi-formally described–most likely in verbatim text. Once the predicates are given, the refinement can be fully automated. We implemented this refinement in our ECPT approach and expect the predicates as a user-defined input.

4.3.2 Randomisation of Concrete Input Selection

The original ECPT as presented in [HP13] assumes a fixed representative from each IEC of the IECP used for test-case generation. This is based on hypotheses about the SUT—especially the hypothesis that the SUT is a member of the fault domain, as introduced in Section 2.5.3. This implies that the IECP used for test-case generation is adequate for testing: i.e., that all inputs of an IEC are equally suited to cause the test case to fail if applied to an SUT with an IECP that deviates from the IECP of the test model but fulfills Equation 2.78.

If the SUT is outside the fault domain, the use of a fixed representative from each IEC may be unfavorable. In the following paragraphs, we demonstrate that the use of multiple different input values from each IEC yields a higher fault-detection probability whenever the SUT is outside the fault domain because the SUT has an IECP that violates Equation 2.78.

Therefore, we combine our ECPT approach with random testing as follows: (1) We apply the I/O-equivalence abstraction to the RIOSTS to obtain a DFSM abstraction of the SUT. On this DFSM, we apply the W/Wp-method to obtain a finite set of symbolic test cases. These symbolic test cases are described by sequences of IECs, which are the input symbols of the DFSM. (2) Instead of using one fixed representative from each IEC to obtain concrete inputs (and in turn a concrete test suite), we select multiple concrete inputs from each IEC. Whenever an IEC X appears in the IEC sequence of a symbolic test case, a new random value that is a member of X is selected. This strategy is denoted STRAT-2 in our experiments in Chapter 6. Because each IEC is referenced in the symbolic test suite more than once, this results in a better coverage of the whole input domain than can be obtained using fixed representatives. In the remainder of this subsection, we demonstrate why the use of randomisation is justified and how the selection of multiple values can be implemented.

4.3.2.1 Justification for Randomisation

The randomisation of input selection aims at higher fault coverage for SUTs that are outside the fault domain. Assuming that a system violates Equation 2.78 (property 5 of Definition 21), the IECP used for test-case generation is not adequate for the test of the SUT.

For simplicity, we assume that the SUT shares the same SECP as the test model. We further assume that, for a single IEC $X \in \mathcal{I}_c$ of the coarsest IECP of the test model, a subset X_f of inputs exists. All inputs from this subset, and only inputs from this subset, provoke an error in the SUT (i.e., a transfer or output fault) when applied to any of the states from exactly one reachable SEC $\mathbf{a}_f \in \mathcal{A}_c$.

In this case, the probability $P_{D|\mathbf{a}_f \wedge X}$ to detect this error in a test step–denoted by event D, given that the test step starts in a state in $\mathbf{a}_f$ and uses a concrete input from X, denoted by event $\mathbf{a}_f \wedge X$–is given by

$$P_{D|\mathbf{a}_f \wedge X} = \frac{|X_f|}{|X|}. \tag{4.33}$$

By construction, the probability of detecting the error in a test step that does not start from a concrete state in $\mathbf{a}_f$ or does not use a concrete input from X, denoted by $\neg(\mathbf{a}_f \wedge X)$, is zero:

$$P_{D|\neg(\mathbf{a}_f \wedge X)} = 0. \tag{4.34}$$

For a strategy in which exactly one fixed representative is chosen for each IEC, the probability of detecting an error as constructed above is $P_{D|\mathbf{a}_f \wedge X} = \frac{|X_f|}{|X|}$. This is independent of the number of test cases and test steps.

If a random input is selected from each IEC X whenever an input of X is needed, the overall probability of detecting the error as constructed above is given by the following term:

$$1 - (1 - P_{D|\mathbf{a}_f \wedge X})^n, \tag{4.35}$$

where n is the number of test steps starting from a state in $\mathbf{a}_f$ and using a concrete input from X. Obviously, the fault-detection probability is greatly increased for large values of n. Note that n differs for all pairwise combinations of SECs and IECs. To guarantee a minimal n for all combinations, one could add extra test cases to ensure that, for each SEC/IEC combination, at least n test steps exist in the test suite. Given that an assumption of the minimal size of X_f is possible, it is possible, using Equation 4.35, to calculate the "confidence level" of the concrete input selection: i.e., the probability of detecting errors that result from mismatches in SUT and test model IECPs. Alternatively, a value for n can be calculated to fulfil a prescribed confidence level, which might depend on the system criticality. This n can in turn be used to extend the test suite as described above.

Note that the calculations above are based on some assumptions. First of all, the SUT is assumed to have the same SECP as the reference model. When relaxing this assumption, two cases have to be considered. In case the SECP of the implementation has more SECs than the number of maximal SECs m assumed for test-case generation, it must be expected that the error will not be detected. In case the number of SECs of the implementation is less or equal to m, an error might be expected to be detectable with a probability that is higher than the term shown in Equation 4.35–because additional inconsistencies mean additional errors and more errors are in general easier to detect. But this assumption is not always true, as we discuss in Chapter 7 in the context of high-order mutation testing. The combination of errors can result in situations in which multiple errors in combination are harder to detect than single errors in isolation would be. However, we expect that, in most cases, the randomisation of concrete input selection will result in a higher test strength.

4.3.2.2 Completeness Properties for Randomised Concrete Input Selection

Section 2.5.3 introduced the fault model $\mathcal{F}(\mathcal{S}, \leq, \mathcal{D}(\mathcal{S}, m, \mathcal{I}))$ for our approach. Recall that [HP16a] proves that the application of the W/Wp-method to the DFSM abstraction of the SUT–with expected behaviour captured by a RIOSTS $\mathcal{S}$–results in a test suite that is complete with respect to the fault model. However, the results in [HP16a] rely on the fact that the concrete test suite for $\mathcal{S}$ is obtained by selecting one representative for each IEC from $\mathcal{I}$. Unfortunately, the completeness property of the testing approach is not preserved if different random elements of an IEC are selected each time a candidate from this IEC is needed. This is based on the fact that the IECP $\mathcal{I}$ used for test-case generation and fulfilling the properties stated in Definition 21 is not necessarily an IECP of the SUT. However, the W/Wp-method relies on the application of input sequences from the characterisation set to identify the target states of input sequences applied to the system's initial state. However, different randomised concrete input sequences obtained from the same sequence of IECs may put the SUT into different target states if $\mathcal{I}$ is not an IECP of the SUT. Under specific conditions, this may lead to situations in which erroneous SUTs from the fault domain are not rejected by the generated test suite.

The completeness property of the testing approach can be preserved if the test suite is created as proposed in [PHH18]. Therefore, the set of Wp-method test cases is first applied with the choice of fixed representatives from each IEC; afterwards, the same set of test cases is applied by using randomised inputs from the IECs. Thus, in effect, two test suites are applied: one test suite with fixed representatives ensuring completeness and one test suite with randomised values to increase test strength for SUTs outside the fault domain. Obviously, this approach needs twice as many test cases.

For our experiments, we do intentionally not use this approach but simply apply the randomised Wp-method tests to keep the number of test cases constant.[2] However, this randomised test suite, while not preserving the completeness for $\mathcal{F}(\mathcal{S}, \leq, \mathcal{D}(\mathcal{S}, m, \mathcal{I}))$ as defined in Section 2.5.3, still guarantees completeness for a generalised fault model $\hat{\mathcal{F}}(\mathcal{S}, \leq, \mathcal{D}(\mathcal{S}, m, \mathcal{I}))$ obtained when Definition 21 is reformulated as follows:

Let $\mathcal{I}$ be the IECP used for test-case generation which needs to be an IECP of the test model $\mathcal{S}$. A RIOSTS $\mathcal{S}'$ is part of the fault domain if it fulfills properties 1-4 and 6 of Definition 21 and $\mathcal{I}$ is an IECP of $\mathcal{S}'$ as well.

The completeness of the randomised approach with respect to this generalised fault model has been proven in [HP16b].

4.3.2.3 Implementation of Randomisation

The randomisation of concrete input selection imposes a challenge on the test data generation. The main challenge is caused by the complexity of the SMT solver used to calculate concrete inputs from the IECs described by first-order logic predicates. Since SMT solvers rely on Boolean Satisfiability Problem (SAT) solvers which are proven to be NP-complete, these solvers in most cases belong to NP or even higher-complexity classes [Mon16]. Hence, the number of SMT instances to be solved during test data generation should be reduced. The randomisation counteracts the objective of reducing the number of SMT instances. Every concrete value has to be calculated by solving an SMT instance. Thus, the runtime of the test data generation largely

[2] In preparation of the experimental results published in [HHP17] we observed that the mutation score is hardly ever affected if the test cases are doubled using one fixed test suite and one randomised test suite.

depends on the number of concrete inputs that are calculated. In the following paragraphs, we discuss two different implementations for concrete input-data selection and afterwards present an alternative means to calculate concrete input data by interval analysis. This alternative approach offers the advantage that no SMT solver is needed; but we will show that this approach is not applicable to all input equivalence classes. All different selection approaches have been implemented in our implementation of the ECPT approach and are competitively used to calculate multiple concrete inputs of IECs as quickly as possible.

4.3.2.3.1 Negation of Existing Solutions A natural approach to generate different solutions for the same SMT instance–in our case for the same predicate Φ_X describing an IEC–is to re-run the SMT solver multiple times. On each execution, all assignments $\sigma_1, \ldots, \sigma_{i-1}$ that have been returned by the SMT solver so far have to be excluded from the search space. Thus, the i-th concrete input from IEC X can be calculated by solving the following SMT instance:

$$\Phi_X \wedge \bigwedge_{j=1}^{i-1} \neg\sigma_j \tag{4.36}$$

Using this approach, it is possible to generate all possible solutions of Φ_X. Note that this approach is analogous to the All-SAT problem. The obvious drawback of this method is that the size of the SMT instance grows linearly in the number of inputs that are generated. Because SMT is known to be NP-complete, the worst-case runtime of existing SMT solvers grows exponentially in the size of the SMT instance. Another flaw of this approach can arise the SMT solver returns "adjacent" solutions every time it is called. Randomisation aims at a good distribution of values. Hence, solutions that differ as much as possible are favoured over concrete input vectors that are adjacent. The experiments indicate that this problem is not apparent. Our SMT solver uses a conflict-driven SAT solver (see the next paragraphs). The nature of this SAT solver induces an acceptable distribution of values, because conflict clauses and propagation ensure that more than one bit is changed when excluding one total assignment.

4.3.2.3.2 Randomisation of the SAT solver As can be seen in the previous paragraph, the negation of existing solutions might evoke an exponential explosion of the runtime if many concrete inputs are needed.

We therefore implemented an alternative approach which is based on a randomisation of the SAT solver that is used. Our SMT solver SONOLAR uses the SAT solver MINISAT [ES03].

MINISAT is a conflict-driven SAT solver based on the famous DPLL algorithm from [DLL62]. A general explanation of the algorithm used by MINISAT is given in Algorithm 4. The algorithm is taken from [ES03], with slight adaptions regarding the notation. For more details on MINISAT, refer to [ES03].

MINISAT has been randomized by manipulating the behavior of Line 8 in Algorithm 4. The call to `DECIDE()` selects the next variable to be assigned and the value to be used for the assignment. This is the part of the SAT solver algorithm that performs the search of the state space. MINISAT and SAT solvers use heuristics to decide which variables and values to choose. These heuristics are important instruments which make SAT solving efficient. To generate different solutions,

Algorithm 4 Conflict-driven SAT solver

Input: $F(x_1, \ldots, x_n)$ as boolean function
Output: Assignment σ for $x_1, \ldots, x_n$, or UNSAT if not satisfiable

```
1:  function SOLVESAT
2:      while true do
3:          PROPAGATE( )            ▷ performs unit propagation, completing the assignment σ
4:          if not conflict then
5:              if all variables assigned in σ then
6:                  return σ                          ▷ F is satisfiable, return assignment
7:              else
8:                  DECIDE( )     ▷ extends assignment σ by selecting next variable and value
9:              end if
10:         else
11:             ANALYZE( )                  ▷ analyzes conflict and adds conflict clause
12:             if top-level conflict then
13:                 return UNSAT                                   ▷ F is unsatisfiable
14:             else
15:                 BACKTRACK( )          ▷ undo assignments until conflict clause is unit
16:             end if
17:         end if
18:     end while
19: end function
```

we replaced the heuristics by a Pseudo Random Number Generator (PRNG). The rest of the algorithm is unchanged.

This approach does not increase the size of the SAT instance for successive runs and thus does not suffer from the exponential growth of runtime due to larger problem instances. Otherwise, the circumvention of the heuristics can cause significant disadvantages for the overall runtime of many SAT instances. SAT solver branching heuristics are able to significantly reduce the runtime for a wide range of SAT instances, as has been shown in [Sil99, MMZ+01, LGPC16].

Replacing these heuristics by an PRNG might thus reduce the overall performance and make the advantages of a constant-sized SAT instance obsolete. We expect this strategy to be superior to the strategy proposed before, which is based on the negation of existing solutions, if a large number of concrete solutions is to be generated.

4.3.2.3.3 Interval Analysis To overcome issues resulting from the complexity of SMT solving, we implemented an alternative approach based on interval analysis. This approach uses the Set-Inverter-Via-Interval-Analysis (SIVIA) algorithm originally presented in [JW93a, JW93b]. This algorithm can be used to calculate inner-approximations of IECs. The approximations are represented as regular subpavings: a compact data structure to represent sets as a union of interval vectors. Given a regular subpaving, an PRNG can be used to efficiently calculate concrete values from these inner-approximations.

We will shortly introduce the notions and notations needed to understand the SIVIA algorithm. We use the notations from [JKDW01]. For a thorough introduction to interval analysis, we refer to [JKDW01].

$[x]$ is called *interval real* or merely *interval* if $[x]$ is a connected subset of $\mathbb{R}$. In this work, we consider only *closed intervals* notated as $[x] = [\underline{x}, \overline{x}]$. Closed intervals include their endpoints $\underline{x}$ and $\overline{x}$. $\underline{x}$ is called *lower bound* of $[x]$ and $\overline{x}$ is called *upper bound* of $[x]$. The *width* $w([x])$ of an interval $[x] = [\underline{x}, \overline{x}]$ is defined as $w([x]) \equiv \overline{x} - \underline{x}$.

An *interval real vector* $[\mathbf{x}]$ is a subset of $\mathbb{R}^n$ and can be defined as the Cartesian product of n *interval components*:

$$[\mathbf{x}] = [x_1] \times [x_2] \times \ldots \times [x_n]. \tag{4.37}$$

In the remainder of this work, an interval real vector will be called *box*. The width of a box is defined as the sum of each component's width.

The definitions above can be applied to the boolean domain as well. An *interval boolean* is a subset of $\mathbb{B}$. There are exactly four possible interval booleans $\mathbb{IB} = \{\emptyset, 0, 1, [0, 1]\}$.

Note that the intervals and boxes we introduced are a special case of a *set of wrappers*: $\mathbb{IX}$ is called a set of wrappers for $\mathbb{X}$ if $\mathbb{IX}$ is a set of subsets of $\mathbb{X}$, $\mathbb{IX}$ contains at least each singleton of $\mathbb{X}$ and $\mathbb{IX}$ is closed by intersection.

The *smallest wrapper* $[\mathbb{X}']$ of $\mathbb{X}' \subset \mathbb{X}$ is the smallest element of the set of wrappers $\mathbb{IX}$ that contains $[\mathbb{X}']$. Applied to intervals, the smallest wrapper of a subset $\mathbb{X}'$ is the smallest closed interval that contains all elements of $\mathbb{X}'$. For example, consider the set $\mathbb{X}' = \{4.2, 5.6, 7.8\}$ as a subset of $\mathbb{R}$. The smallest wrapper for $\mathbb{X}'$ is $[4.2, 7.8]$.

Every binary operator $\diamond : \mathbb{X} \times \mathbb{Y} \to \mathbb{Z}$ with sets of wrappers $\mathbb{IX}$, $\mathbb{IY}$ and $\mathbb{IZ}$ can be extended to sets of wrappers:

$$\mathbb{X}_1[\diamond]\mathbb{Y}_1 \equiv [\{x_1 \diamond x_2 | x_1 \in \mathbb{X}_1, y_1 \in \mathbb{Y}_1\}], \tag{4.38}$$

with $\mathbb{X}_1 \in \mathbb{IX}$ and $\mathbb{Y}_1 \in \mathbb{IY}$.

The same applies to arbitrary functions $f : \mathbb{X} \to \mathbb{Y}$:

$$[f](\mathbb{X}_1) \equiv [\{f(x) | x \in \mathbb{X}_1\}]. \tag{4.39}$$

Note that the wrapped versions of binary operators and functions always introduce pessimism:

$$\mathbb{X}_1[\diamond]\mathbb{Y}_1 \supset \mathbb{X}_1 \diamond \mathbb{Y}_1 \tag{4.40}$$

$$[f](\mathbb{X}_1) \supset f(\mathbb{X}_1). \tag{4.41}$$

The wrapped image of a binary operator or a function is a superset of the *direct image* of the operator ($\mathbb{X}_1 \diamond \mathbb{Y}_1 = \{x_1 \diamond x_2 | x_1 \in \mathbb{X}_1, y_1 \in \mathbb{Y}_1\}$) or function ($f(\mathbb{X}_1) = \{f(x) | x \in \mathbb{X}_1\}$), respectively. The pessimism is caused by the wrapping and dependency effects.

Equation 4.38 gives rise to an extension of the four arithmetic operators $+$, $-$, $\cdot$, $/$ for intervals. This allows for *interval computation*. For example, $[1.2, 3.0][+][0.5, 4.2]$ yields $[1.7, 7.2]$. As a consequence, every arithmetical expression can be computed at intervals, although the pessimism introduced by the wrapping and dependency effects over-approximates the set of solutions.

A *test* is a function $\mathbf{t} : \mathbb{R}^n \to \mathbb{B}$. An *inclusion test* $[\mathbf{t}]$ for $\mathbf{t}$ has to fulfil the following properties:

$$[\mathbf{t}]([\mathbf{x}]) = 1 \Rightarrow \forall \mathbf{x} \in [\mathbf{x}] : \mathbf{t}(\mathbf{x}) = 1 \tag{4.42}$$

$$[\mathbf{t}]([\mathbf{x}]) = 0 \Rightarrow \forall \mathbf{x} \in [\mathbf{x}] : \mathbf{t}(\mathbf{x}) = 0. \tag{4.43}$$

Inclusion tests can be used to characterize sets. For a set $\mathbb{X}$, the test $\mathbf{t}$ is defined as $\mathbf{t}(x) \Leftrightarrow (x \in \mathbb{X})$. If $\mathbb{X}$ is a subset of $\mathbb{R}$, the inclusion test $[\mathbf{t}]$ can be obtained by the use of the wrapped versions of binary operators.

A *subpaving* of a box $[\mathbf{x}] \subset \mathbb{R}^n$ is a union of non-overlapping subboxes of $[\mathbf{x}]$. A subpaving is called finite if it contains finitely many subboxes. Subpavings offer a means for the description of arbitrary compact sets of $\mathbb{R}^n$. Finite subpavings can be used to over-approximate any compact set $\mathbb{X}$ in $\mathbb{R}^n$: i.e., $\mathbb{X} \subset \overline{\mathbb{X}}$. Furthermore, the approximation $\overline{\mathbb{X}}$ can be chosen to be as close to $\mathbb{X}$ as desired. Close, in this case, is defined using the distance function m_{inf} for compact sets, as proposed in [JW93b]. For every *full* compact set (i.e., a compact set that equals the closure of its interior), an inner-approximation $\underline{\mathbb{X}} \subset \mathbb{X}$ can be defined as well–again as close to $\mathbb{X}$ as desired.

Regular subpavings are a special kind of subpavings. A regular subpaving can be obtained by bisection and selection. Bisection is the process of dividing a box $[\mathbf{x}]$ into its left child (denoted $L[\mathbf{x}]$) and right child (denoted $R[\mathbf{x}]$). The left and right child of a box $[\mathbf{x}] = [x_1] \times \ldots \times [x_j] \times \ldots \times [x_n]$ are obtained by bisecting the first interval component $[x_j] = [\underline{x_j}, \overline{x_j}]$ with maximum width. The left child of $[\mathbf{x}]$ is $L[\mathbf{x}] = [x_1] \times \ldots \times [\underline{x_j}, (\underline{x_j} + \overline{x_j})/2] \times \ldots \times [x_n]$ and the right child is $R[\mathbf{x}] = [x_1] \times \ldots \times [(\underline{x_j} + \overline{x_j})/2, \overline{x_j}] \times \ldots \times [x_n]$

Algorithm 5 shows the SIVIA algorithm as presented in [JKDW01]. This algorithm calculates an inner and outer-approximation of a set $\mathbb{X}$, which is characterized by a test function $\mathbf{t}$, such that $\underline{\mathbb{X}} \subset \mathbb{X} \subset \overline{\mathbb{X}}$. This algorithm gets as input the test function $\mathbf{t}$, a box $[\mathbf{x}]$ and an ϵ that determines the precision of the SIVIA algorithm. $[\mathbf{x}]$ is initially set to a value that is guaranteed to contain the set $\mathbb{X}$ to be characterised. This parameter can be considered the search space of the SIVIA algorithm. Recursively, this box $[\mathbf{x}]$ is refined by bisection. The bisection is performed until one of the following conditions hold: (1) $[\mathbf{x}]$ is completely outside $\mathbb{X}$. In this case, the recursion terminates with no further action. (2) $[\mathbf{x}]$ is completely inside $\mathbb{X}$. In this case, the recursion terminates and $[\mathbf{x}]$ is added to the subpavings $\underline{\mathbb{X}}$ and $\overline{\mathbb{X}}$. (3) The size of $[\mathbf{x}]$ is less than ϵ. In this case, the recursion terminates and $[\mathbf{x}]$ is added to the outer-approximation since some members of $[\mathbf{x}]$ are members of $\mathbb{X}$. ϵ is a parameter that determines how close the inner and outer approximations are to $\mathbb{X}$ and on the other hand how many recursions are needed before SIVIA terminates.

For concrete input selection, we choose a test function $\mathbf{t}_X : D_I \to \mathbb{B}$ for every IEC X defined as a mapping from concrete inputs $\vec{c} \in D_I$ to a boolean value indicating whether $\vec{c}$ is member of the IEC X:

$$\mathbf{t}_X(\vec{c}) = \begin{cases} 1 & \text{if } \vec{c} \in X \\ 0 & \text{else.} \end{cases}$$

Algorithm 5 SIVIA with inclusion test, cf. [JKDW01]

Input: $\mathbf{t}, [\mathbf{x}], \epsilon$
Inout: $\underline{\mathbb{X}}, \overline{\mathbb{X}}$

```
1:  function SIVIA
2:      if [t]([x]) = 0 then                    ▷ [x] is completely outside 𝕏: [x] ∩ 𝕏 = ∅
3:          return
4:      end if
5:      if [t]([x]) = 1 then                    ▷ [x] is completely inside 𝕏: [x] ⊂ 𝕏
6:          𝕏̲ := 𝕏̲ ∪ [x]
7:          𝕏̄ := 𝕏̄ ∪ [x]
8:          return
9:      end if                                  ▷ [x] is undetermined: [x] ∩ 𝕏 ≠ ∅ ∧ [x] ⊄ 𝕏
10:     if w([x]) < ε then
11:         𝕏̄ := 𝕏̄ ∪ [x]
12:         return
13:     end if
14:     SIVIA(t, L[x], ε, 𝕏̲, 𝕏̄)
15:     SIVIA(t, R[x], ε, 𝕏̲, 𝕏̄)
16: end function
```

$\mathbf{t}_X$ can be calculated for a given concrete input $\vec{c}$ given that IEC X is described by a first-order logic predicate Φ_X over input variables I. First, every occurrence of an input variable in Φ_X is replaced by the value, as given by $\vec{c}$. Then the resulting predicate $\Phi_X[\vec{c}(v)/v \in I]$ is an expression of constant values that can be evaluated. If the expression evaluates to true, $\vec{c}$ is a member of X.

The wrapped inclusion test $[\mathbf{t}]$ of $\mathbf{t}$ can be calculated by use of interval computation. Every occurrence of an arithmetical operator $\diamond$ in the predicate Φ_X is replaced by the wrapped version $[\diamond]$ using interval computation. Every occurrence of a boolean operator is replaced by the interval counterpart, as shown below:

$$[B_1][\wedge][B_2] \equiv [\{b_1 \wedge b_2 | b_1 \in [B_1], b_2 \in [B_2]\}] \tag{4.44}$$
$$[B_1][\vee][B_2] \equiv [\{b_1 \vee b_2 | b_1 \in [B_1], b_2 \in [B_2]\}] \tag{4.45}$$
$$[\neg][B_1] \equiv \{1 - b_1 | b_1 \in [B_1]\}. \tag{4.46}$$

Note that all definitions used above for intervals and boxes are based on real numbers from $\mathbb{R}$. The properties of intervals are preserved when natural numbers or integral data types like *int*, *char*, ... are used. The input domain of the SUTs considered in this work can be considered a "mixed" box: Each component of the interval vector is from the domain D_x of the respective input variable x. For integer variables, the interval computation can be performed using integer arithmetics.

The SIVIA algorithm shown above with the wrapped test function for an IEC X that is obtained by the use of interval computation for the predicate Φ_X can be used to calculate regular subpavings that describe the inner and outer approximation of X. The initial box used for the SIVIA algorithm is the complete input domain D_I of the SUT.

The authors in [JKDW01] give an upper-bound for the number of bisections that are performed by SIVIA:

$$\left(\frac{w([\mathbf{x}])}{\epsilon} + 1\right)^n. \tag{4.47}$$

Hence, the complexity of the SIVIA approach depends exponentially on the number of variables n. For concrete input-data selection, this means the number of input variables of the system. Furthermore, the worst-case complexity is dependent on $\frac{w([\mathbf{x}])}{\epsilon}$. Assuming a fixed number of input variables n, there remains a high polynomial complexity class. While $w([\mathbf{x}])$ cannot be changed, the choice of ϵ can reduce the computational costs of the SIVIA approach. On the other hand, increasing ϵ might result in a loss of precision leading to an inner-approximation which misses parts that are necessary for higher fault coverage.

Note, however, that the upper-bound shown in Equation 4.47 is just a worst-case estimate. In many cases, the exponential complexity might not become apparent. Additionally, once the subpavings are generated using the SIVIA algorithm, concrete inputs can be selected using PRNGs. To this end, boxes from the subpaving are randomly chosen. For a chosen box, a concrete input can be calculated by choosing random values from the interval of each component of the box. The runtime of this input selection is linear in the number of input variables. Thus, the computational overhead of calculating a large number of concrete inputs is relaxed if the calculation of an inner-approximation is possible within time constraints. The overall runtime $T_{\text{CIS}}(m)$ needed for selecting m different input values using SIVIA is the sum of the runtime for SIVIA plus the runtime of each input selection using a PRNG:

$$T_{\text{CIS}}(m) = T_{\text{SIVIA}} + m \cdot T_{\text{PRNG}}. \tag{4.48}$$

In contrast, the overall runtime $T'_{\text{CIS}}(m)$ needed by the SMT solver for selecting m different concrete values can be estimated by the following:

$$T'_{\text{CIS}}(m) = m \cdot T_{\text{SMT}}. \tag{4.49}$$

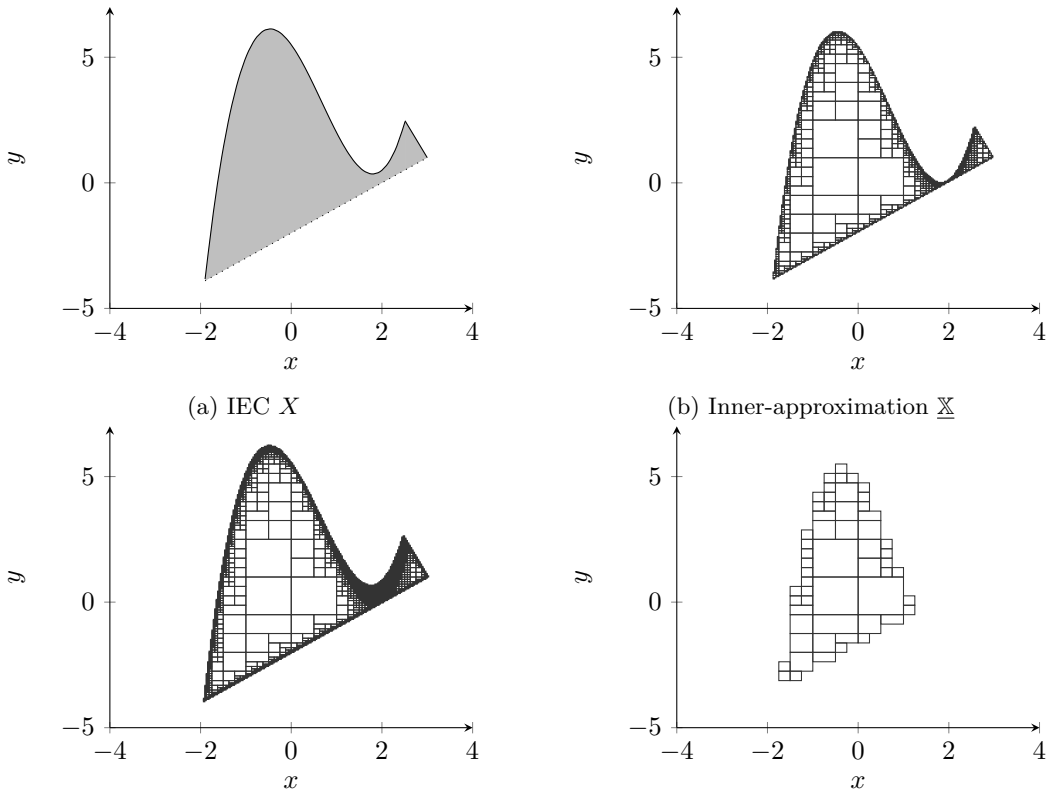
This estimate assumes a constant time T_{SMT} for every SMT solver call. In practice, this will not be the case; as discussed in the paragraphs above, the runtime will most likely be worse. Assuming that input selection from a sub-paving using a PRNG is much faster than an SMT solver call for all but the most trivial cases ($T_{\text{PRNG}} \ll T_{\text{SMT}}$), there will always be a value m for which the SIVIA approach is faster than an SMT approach: $T_{\text{CIS}}(m) < T'_{\text{CIS}}(m)$.

Example 21. Consider an IEC X defined by predicate Φ_X:

$$\Phi_X \equiv y \leq x^3 - 2 \cdot x^2 - 2.5 \cdot x + 5.5 \wedge y > x - 2 \wedge y \leq -3 \cdot x + 10.$$

Figure 4.6 visualises the SIVIA algorithm on X. Figure 4.6a shows the set that is described by Φ_X. Figure 4.6b displays the subpaving that is generated using the SIVIA algorithm with an ϵ of

0.05. The resulting inner-approximation consists of 1059 boxes. The outer-approximation shown in Figure 4.6c contains 2666 boxes. Note that the inner-approximation that is generated using $\epsilon = 0.5$ and shown in Figure 4.6d contains merely 56 boxes. While the runtime of the SIVIA algorithm is much lower in this case, the lack of precision results in an approximation that does not cover the area around the local minimum of the polynomial of degree three. All members of X with a value for x that is greater than 1.25 are not covered by the subpaving.



(a) IEC X

(b) Inner-approximation $\underline{\mathbb{X}}$

(c) Outer-approximation $\overline{\mathbb{X}}$

(d) Inner-approximation $\underline{\mathbb{X}}$ with $\epsilon = 0.5$

Figure 4.6: Visualisation of the SIVIA Algorithm

4.3.3 Boundary-Value Selection

4.3.3.1 Integrating Boundary-value Selection to the ECPT Approach

Boundary-value tests are a well-known testing heuristic and also mandated by safety-related standards [RTC92, Eur01, ECS09]. This is based on the fact that extreme values of equivalence classes are known to reveal more errors (at least errors of a certain type) than interior values of equivalence classes. It is trivial to see that errors resulting from the wrong use of relational operators (e.g., confusion of $\leq$ with $<$) will only be detected by a value that lies on the boundary that is defined by such a wrongly implemented inequality. [Rei97] provides evidence for the superiority of boundary-value analysis.

Therefore, we propose to augment the equivalence class approach by boundary-value testing. This can be achieved by two means: by IECP refinement, or by guiding the concrete input selection. IECP refinement can be used to distinguish between an equivalence class' interior and a boundary. The idea is to refine each IEC X by two IECs $\partial(X)$ and $\mathfrak{I}(X)$, $\partial(X)$ containing the boundary or *frontier* of X and $\mathfrak{I}(X)$ containing the *interior* of X. The frontier of X is intuitively defined as the collection of all members that have at least one neighbouring value that is outside X. The other possibility is to combine boundary-value selection with the randomisation introduced above. We combine both approaches as follows: Whenever a value from an IEC X is needed in a symbolic test case, select a value from the boundary of the IEC with a probability of 50 percent and from the interior of the IEC again with a probability of 50 percent. The selection from the interior and frontier is randomised as presented above. This ensures a reasonable distribution of random values from the IEC. Selecting values from the boundary would miss only certain faults that are not detectable on the boundary of an IEC.[3] However, selecting arbitrary values from an IEC will most likely result in many values from the interior of an IEC and only a few values from the boundary, as the frontier of an IEC is in most cases smaller and thus less likely to be covered. This strategy–i.e., the random selection of values from the boundary and from the interior of an IEC with a probability of 50 percent each–is denoted STRAT-3 in our experiments in Chapter 6.

We believe that this approach is, in most cases, favourable over the IECP refinement because IECP refinement would result in a doubling of the number of IECs. Considering stricter boundary-coverage criteria, as given in [KLPU04], the number of IECs would grow by an even higher factor. Instead, the boundary selection in combination with the randomisation does not affect either the number of IECs or the size of the resulting test suites.

Following [KLPU04], boundary values can be formalised for continuous domains. Let $X \subset \mathbb{R}^n$. A value $A = (a_1, \ldots, a_n) \in X$ is a boundary value if for every $\epsilon > 0$ the ball $B_\epsilon(A)$ of radius ϵ and center A contains at least one point of $\mathbb{R}^n \setminus X$:

$$\partial(X) = \{A \in X | \forall \epsilon > 0, \exists B \in B_\epsilon(A) : B \in \mathbb{R}^n \setminus X\}. \tag{4.50}$$

In case of discrete domains (i.e., $X \subset \mathbb{Z}^n$), [KLPU04] defines the *discrete neighbourhood* of a value $A = (a_1, \ldots, a_n) \in X$ as $V(A) = \{A, (a_1 + 1, \ldots, a_n), \ldots, (a_1, \ldots, a_n + 1)\}$. A value $A \in X$ is a boundary value if at least one value of its discrete neighbourhood $V(A)$ is outside of X:

$$\partial(X) = \{A \in X | \exists B \in V(A) : B \in \mathbb{Z}^n \setminus X\}. \tag{4.51}$$

The interior $\mathfrak{I}(X)$ of X is then defined as $\mathfrak{I}(X) = X \setminus \partial(X)$.

Note that the definition of the discrete neighbourhood is not only defined for integers but for variables of the floating-point data types `float` and `double`, according to IEEE 754. In this case, the discrete neighbour of a floating-point variable is the next higher or lower value that can be represented by data type `float` or `double`. Thus, the definition for boundary values given in Equation 4.51 can be generalised to `float` and `double`. Furthermore, this definition can be extended to boolean variables, given that true and false can be interpreted as 1 and 0, respectively (i.e., $\mathbb{B} = \{0, 1\} \subset \mathbb{Z}$).

[3] Consider for example the confusion of relational operator $\leq$ by $\geq$. This error will not be revealed by a boundary value.

4.3.3.2 Implementation of Boundary-value Selection

For the implementation of boundary-value selection from an IEC X, we are especially interested in inequalities. Note that these inequalities are not necessarily linear inequalities, as the use of an SMT solver allows for arbitrary complex predicates. We assume that our IEC X is defined by a first-order logic predicate Φ_X that contains subformulae of the form $\phi \triangleq f(x_1, \ldots, x_n) \leq 0$, where $f(x_1, \ldots, x_n)$ is a formula over variables $x_1, \ldots, x_n$. Furthermore, assume that this inequality is not implied by other predicates: i.e., that at least one input $\vec{c} \in X$ exists that does not fulfil ϕ: $\vec{c} \models \Phi_X \wedge \vec{c} \not\models \phi$ or, in other words, that Φ defines a segment of the frontier of X. It is obvious to see that values that lie on this segment (i.e., inputs that fulfil $\partial f(x_1, \ldots, x_n) = 0$) are boundary values. Formula $f(x_1, \ldots, x_n) = 0$, which we derived from ϕ by replacing $\leq$ by $=$, is denoted by $\partial f(x_1, \ldots, x_n)$ or $\partial\phi$ in the remainder. Thus, $\partial\phi$ is a sufficient condition for $\vec{c}$ to be a boundary value of ϕ. The same applies for the analogous case in which ϕ has the form $f(x_1, \ldots, x_n) \geq 0$. In this case, $\partial\phi$ is defined as shown in Equation 4.52.

$$\begin{aligned} \partial f(x_1, \ldots, x_n) \leq 0 &\triangleq f(x_1, \ldots, x_n) = 0 \\ \partial f(x_1, \ldots, x_n) \geq 0 &\triangleq f(x_1, \ldots, x_n) = 0 \end{aligned} \tag{4.52}$$

For the discrete case in which the variables are of integer or floating-point type, we can specify the sufficient condition $\partial\phi$ for a boundary value of inequalities involving $<$ and $>$, as shown in Equation 4.53.

$$\begin{aligned} \partial f(x_1, \ldots, x_n) < 0 &\triangleq f(x_1, \ldots, x_n) = sd(0) \\ \partial f(x_1, \ldots, x_n) > 0 &\triangleq f(x_1, \ldots, x_n) = si(0) \end{aligned} \tag{4.53}$$

$sd(x)$ denotes the smallest decrement (i.e., next lower representable value), and $si(x)$ denotes the smallest increment (i.e., the next higher representable value). In case of an integer variable x, $sd(x)$ is given by $x - 1$ if x is greater than the smallest representable value of the concrete integer type. For a floating-point variable, this value is the next lower representable floating point value, as specified by IEEE 754. The smallest increment $si(x)$ is defined analogously.

For the continuous case, i.e., $x_1, \ldots, x_n \in \mathbb{R}$, there is no boundary value for a formula of the form $f(x_1, \ldots, x_n) < 0$ because there will always a smaller ϵ for which the ball $B_\epsilon(\vec{c})$ will only contain values that fulfil $f(x_1, \ldots, x_n) < 0$.

To summarise the observations above, given an IEC X described by a predicate $\Phi_X = \varphi \wedge \bigwedge_{i=1}^{n} \phi_i$, where the ϕ_i are formulae of one of the forms introduced above and φ is of arbitrary form, the boundary values can be selected by solving the predicate $\partial\Phi_X$ as defined by Equation 4.54 with $\partial\phi_i$ as defined above.

$$\partial\Phi_X \triangleq \Phi_X \wedge (\bigvee_{i=1}^{n} \partial\phi_i) \tag{4.54}$$

Note that, for discrete variable domains, Equation 4.54 may have no solution. However, this fact does not mean that no boundary values exist. Consider, for example, an inequality $\phi \triangleq x^2 - 26 \leq 0$ over integer variable x. $\partial\phi \triangleq x^2 - 26 = 0$ has no integer solution. However, constraint programming/optimisation techniques could be used to obtain the boundary value $x = 5$ in this

special case.[4] Thus, our approach using an SMT solver and Equation 4.54 does not guarantee that we will find all boundary values; but the solutions found are guaranteed to be boundary values. Note that, for our case studies, we were able to generate boundary values for all IECs. In many cases, it is possible to get boundary values by reformulation of predicates–such as by changing the above predicate to $x^2 - 25 \leq 0$, which is semantically equivalent for the integer domain.

To randomly select boundary values by solving Equation 4.54, interval selection cannot be used because of the "narrow nature" of boundaries. However, for the selection of interior values, interval analysis is in many cases still applicable.

Example 22. Consider the IEC X described by the predicate $\Phi_X \equiv y \leq x^3 - 2 \cdot x^2 - 2.5 \cdot x + 5.5 \wedge y - x > -2 \wedge y \leq -3 \cdot x + 10$, as introduced in Example 21. The following conditions for boundary values can be identified:

$$\begin{aligned} \partial y \leq x^3 - 2 \cdot x^2 - 2.5 \cdot x + 5.5 &\equiv y = x^3 - 2 \cdot x^2 - 2.5 \cdot x + 5.5 \\ \partial y - x > -2 &\equiv y - x = si(-2) \\ \partial y \leq -3 \cdot x + 10 &\equiv y = -3 \cdot x + 10. \end{aligned}$$

Thus, boundary values can be obtained by solving the following:

$$\begin{aligned} \Phi_X \wedge (y = x^3 - 2 \cdot x^2 - 2.5 \cdot x + 5.5 \vee \\ y - x = si(-2) \vee \\ y = -3 \cdot x + 10). \end{aligned}$$

Examples for solutions (in the form of (x, y)) are $(1, 2)$, $(0.5, -2.4999998)$ and $(2.75, 1.75)$.

The considerations above can reasonably be used for the calculation of boundary values for parts of the IEC frontier, where integer or floating values are involved. However, for boolean input variables, the definitions from above are of limited use. Therefore, we follow the boundary-value definition given in [PHH16b] to calculate boundary values for predicates including boolean variables:

$$\partial x \triangleq x \tag{4.55}$$
$$\partial \neg x \triangleq \neg x \tag{4.56}$$
$$\partial \neg \varphi \triangleq \partial NNF(\neg \varphi) \tag{4.57}$$
$$\partial(\varphi \wedge \phi) \triangleq \partial(\varphi) \wedge \partial(\phi) \tag{4.58}$$
$$\partial(\varphi \vee \phi) \triangleq (\partial(\varphi) \wedge \neg\phi) \vee (\neg\varphi \wedge \partial(\phi)). \tag{4.59}$$

For formulae containing a boolean variable x, an assignment of $x = \text{true}$ is a boundary value, since a change of the value of x will result in the formula evaluating to false (Equation 4.55). The

[4] However, this boundary value will not detect any defects related to the confusion of $<$ and $\leq$ in $x^2 - 26 \leq 0$. In fact replacing $\leq$ by $<$ or 26 by any integer constant from [25,35] yiels I/O-equivalent mutants.

same argument applies to $\neg x$ (Equation 4.56). Formulae of the form $\neg\varphi$ have to be reformulated to the negation normal form, denoted by $NNF(\neg\varphi)$ (Equation 4.57). Then the two last rules (Equation 4.58,Equation 4.59) can be used to obtain boundary values from formulae containing $\wedge$ and $\vee$. The intuition behind Equation 4.59 is that for a boundary value $\vec{c}$ of $\varphi \vee \phi$, according to Equation 4.51, the evaluation of $\varphi \vee \phi$ has to change to false if one component of the boundary value $\vec{c}$ is changed. This is the case for all values $\vec{c}$ that either do not fulfil ϕ and are a boundary value for φ or that do not fulfil φ and are a boundary value of ϕ.

Example 23. Consider the IEC defined by predicate $\Phi_X \equiv a \wedge (b \vee c)$ over boolean variables a, b, c. Boundary values for this IEC are calculated as follows:

$$
\begin{aligned}
\partial(a \wedge (b \vee c)) &\equiv \partial a \wedge \partial(b \vee c) && \text{Equation 4.58}\\
&\equiv a \wedge \partial(b \vee c) && \text{Equation 4.55}\\
&\equiv a \wedge ((\partial(b) \wedge \neg c) \vee (\neg b \wedge \partial(c))) && \text{Equation 4.59}\\
&\equiv a \wedge ((b \wedge \neg c) \vee (\neg b \wedge c)) && \text{Equation 4.55}
\end{aligned}
$$

Note from the example above, that the final boundary predicate is the Modified Condition/Decision Coverage (MC/DC) condition of the original predicate. Hence, the MC/DC condition of a predicate that involves boolean variables only can be considered the boundary value condition.

To summarise, we use Equation 4.53 and Equation 4.52 for boundary-value selection for integer and floating-point data-type guard conditions. For guard conditions involving boolean variables only, we use the MC/DC conditions of the respective guard conditions.

[KLPU04] not only formalises the definition of boundary values but also defines boundary-coverage criteria. [KLPU04] introduces a hierarchy of coverage criteria ranging from the *One-Boundary* criterion (at least one boundary value is selected) to the *All-Boundaries* criterion, which states that every boundary value is chosen. Intermediate criteria like *All-Edges* (at least one boundary value from each edge of the IEC) or *Multi-Dimensional* (every variable takes its minimum and maximum value) lie in between. Our approach to calculate boundary values by solving Equation 4.54 can best be considered a *Some-Boundary* approach, as we neither guarantee that all boundary values are selected (this would be infeasible in case of infinite input domains) nor that every edge or even every vertex is covered. Such guarantees would require an IECP refinement which drastically increases the number of IECs and in turn the overall test effort. However, the combination with randomisation will result in multiple different boundary values being selected.

4.3.4 Extension of Test Cases for the Heuristic Exploration of Additional States

The heuristics presented above are efficient for tackling the problem of a mismatch of the IECP of the test model and the SUT. The second issue that has an influence on the test strength is the problem of additional states. As seen in Section 2.5.3, the fault domain is dependent on the upper bound of the number of states the SUT is assumed to have. Increasing m apparently results in a larger fault domain and thus theoretically increases the number of faulty systems that are rejected by the generated test suite. On the other side the exponential growth on the number of generated test cases makes the increase of m intractable for most problems. Therefore, we present

a heuristic to increase the test strength of our IECP approach in the presence of additional states in the implementation.

The general idea behind our heuristic is to extend the existing set of test cases by random suffixes. Contrary to the approach of increasing variable m for application of the W/Wp-method, this does not introduce extra test cases. Instead, existing test cases are extended. In case of system and HSI tests, the number of test cases is the most important cost metric. In many cases, a restart of the SUT is very costly compared to a single test step. Therefore, the extension of test cases will in general not introduce significant extra costs. Still, the extended test cases might reveal errors in the SUT that can only be uncovered if a test case of a certain length is executed.

Assume that $a = m - n$ is the number of additional states, where n is the number of states in the minimal DFSM abstraction of our reference model and m is the number of states the minimal DFSM representation of our SUT is assumed to have.

All test cases generated by the Wp-method (or the W-method), denoted by $\mathcal{W}_p$, are extended by a sequence of inputs of length a, resulting in a set of test cases, denoted by $\mathcal{W}_p^*$. The resulting test suite obviously has a better (or at least the same) test strength as the original one, because all test cases in $\mathcal{W}_p$ are prefixes of test cases in $\mathcal{W}_p^*$. Thus, $\mathcal{W}_p^*$ uncovers at least all faults that are revealed by $\mathcal{W}_p$.

4.3.4.1 Heuristics for Suffix Generation

The generation of the test case suffixes can be performed in different ways.

1. Perform a random walk of length a through the DFSM starting at the state reached by the prefix from $\mathcal{W}_p$
2. As 1, but ensure that every vertex of the DFSM is reached equally often as the final state after application of the complete input sequence (i.e., the prefix from $\mathcal{W}_p$ and the randomly generated suffix of length a) to the initial system state.
3. As 1, but ensure that every edge of the DFSM is selected equally often as final edge in the path resulting from application of the complete input sequence to the initial system state.

The heuristics presented above perform a random search for additional states. After application of the randomly generated suffix, an additional faulty state in the implementation might be reached. This state could still show correct outputs: i.e., the same outputs as state q_e that is expected according to the test model's DFSM. Therefore, it is necessary to identify this state. Thus, the prefix from $\mathcal{W}_p$ extended by the randomly generated suffix must be further extended by an element from the characterisation set CS or from the state-identification set W_e. Note that the state-identification set of the final state and the characterisation set will in general contain more than one input sequence that is necessary to identify a state. To keep the number of test cases unchanged, one input sequence is chosen randomly.

The random generation of paths through our SUT, or the DFSM representation of the SUT, can be reduced to the problem of random generation of combinatorial structures. [DGG04] presents a generic approach to generate random paths from any graphical representation: for example, graphs, DFSMs, STSs. The main focus of this work is to perform the random generation uniformly over all existing paths, additionally guided by coverage criteria. Our random generation of suffixes is based on the approach presented in [DGG04] and is presented below. First, we

recall the approach from [DGG04] and then state how our three heuristics listed above can be realised.

4.3.4.2 Uniform Random Generation of Paths in a Graph

Given a directed graph $G = (V, E)$, with vertices V and edges that are labelled with elements $l \in L$: $E \subseteq V \times L \times V$, let $\mathcal{P}_n$ denote all paths of length n that start at the start vertex v_s and end in the end vertex v_e. A path P of length n is a sequence of alternating vertices and edges $P = v_0.e_0.v_1.\ldots.v_{n-1}.e_{n-1}.v_n$, such that v_i and v_{i+1} are connected by an edge $e_i = (v_i, l, v_{i+1}) \in E$ for $i \in \{0, \ldots, n-1\}$. Note that paths are sometimes restricted to paths where each vertex occurs at most once. These paths are called *simple paths*. In our case, the paths to be generated randomly are not constrained to be simple paths.

The aim of uniform random generation of paths in a graph is that every path from the set of paths of length n, denoted by $\mathcal{P}_n$, is generated with the same probability. To achieve this, the algorithm shown in Algorithm 6 is performed.

Algorithm 6 Uniform Random Generation of a Path in a Graph

Input: $G = (V, E)$ as a directed graph
Input: v_s as the vertex to start at
Input: v_e as the vertex where the path ends
Input: n as the requested length of the path
Output: a path $P = v_s.e_0.v_1.\ldots.v_{n-1}.e_{n-1}.v_e$ of length n

```
1:  function generateRandomPath
2:      m := n
3:      v := v_s
4:      P = v_s
5:      while m > 0 do
6:          A := {(v, l, v_i)|(v, l, v_i) ∈ E}
7:          randomly select e from A with probability for e = (v, l, v_i) defined by f_{v_i}(m−1)/f_v(m)
8:          P := P.e.v_i
9:          v := v_i
10:         m := m − 1
11:     end while
12:     return P
13: end function
```

The algorithm starts at the start vertex, v_s. Initially, v_s becomes the current vertex (variable v), and the path is successively extended with the next randomly selected edge that starts from the current vertex v. Then the target vertex of this edge becomes the current vertex v, and the next edge is chosen. The loop terminates after n iterations. The algorithm has to guarantee that all paths of length n to end vertex v_e can be generated and that each possible path of length n is generated with the same probability. This is realized by a special distribution: An edge $e = (v, l, v_i)$ starting at the current vertex v is selected with probability $f_{v_i}(m-1)/f_v(m)$. $f_v(m)$ is the number of paths of length m that start from vertex v and end at the end vertex, v_e. Thus, $f_{v_i}(m-1)/f_v(m)$ is the ratio of the number of paths of length m starting with $v.e_i.v_i$ and ending in v_e to the overall number of paths of length m starting in v and ending in v_e. Therefore, the probability of choosing an edge is proportional to the number of paths that are still possible

after this edge has been chosen. In particular, the probability for an edge leading to a vertex from which no path to v_e exists is zero. It is obvious that this distribution guarantees that, (1) all and only paths of length n from v_s to v_e can be generated, and (2) that each possible path is generated with the same probability.

$f_v(m)$ can be calculated recursively:

$$f_v(0) = \begin{cases} 1 & \text{if } v = v_e \\ 0 & \text{else} \end{cases} \tag{4.60}$$

$$f_v(m) = \sum_{(v,l,v') \in E} f_{v'}(m-1) \qquad \text{for } m > 0. \tag{4.61}$$

Algorithm 6 makes extensive use of $f_v(m)$. Therefore, it is beneficial to use the algorithmic concept of *dynamic programming*. Every $f_v(m)$ is calculated at most once for each value of $v \in V$ and $m \in \{0, \ldots, n\}$. Once the value for $f_v(m)$ is calculated, this value is stored in a look-up table. This value can be reused in later calculations that require it. [FZC94] shows that the approach can be implemented very efficiently. The number of operations for the generation of the look-up table is in $\mathcal{O}(n \cdot |V|^2)$ in the worst case, and the number of operations for random path generation is in $\mathcal{O}(n \log n)$ in the worst case. In the worst case, the memory needed for the look-up table is in $\mathcal{O}(n \cdot |V|)$.

4.3.4.3 Implementation of Suffix Heuristics

Recall the three proposed heuristics listed in Section 4.3.4.1. The first heuristics aims at a purely random generation of paths through the DFSM of the SUT. In this case, no final state exists; consequently, there is no v_e. In this case, Algorithm 6 is unnecessary, because the DFSMs considered in this work are completely specified. There are no vertices in the DFSM without outgoing edges. Therefore, every vertex allows for a next edge to be selected (and the out degree is exactly the same for every vertex). In this special case, the random generation of paths through the DFSM graph can be performed by randomly selecting a (i.e., the number of additional states the SUT is assumed to have) edge labels. Note that each edge label in our DFSM abstraction represents an IEC. Hence, the uniformly generated random path through the DFSM of length a with an arbitrary target state represents a random sequence of IECs.

However, heuristics 2 and 3 aim at visiting a specific state or a specific transition to ensure that every state or every edge has been visited equally often. A uniform generation of all paths through the graph guarantees that all possible paths of a specific length are generated uniformly; but this does not guarantee that specific states or edges of the graph are covered equally well. States (or edges) that are hard to be covered, because of a low number of paths that go through this state could be missed by Heuristic 1. This problem has to be overcome by Heuristic 2. This can be achieved for Heuristic 2 as shown in Algorithm 7.

Each test case from $\mathcal{W}_p$ is to be extended by a suffix. After application of the input trace $t \in \mathcal{W}_p$, a state q_s is reached. This state becomes the start vertex for random path generation. The end vertex for the path generation is chosen randomly from the set of all least "visited" states. In this case, a state q is said to be visited if it is the target state after application of the test input sequence t extended by the randomly generated suffix. Initially, all $q \in Q$ are visited zero times.

Algorithm 7 Suffix Generation for Heuristic 2

for each $t \in \mathcal{W}_p$ **do**
  $q_s :=$ state after application of t to $\mathfrak{q}$
  randomly select q_e from the least visited states
  suf :=generateRandomPath(M, q_s, q_e,a)
  extend t by sequence of inputs in suf
  increment number of visits for q_e
end for

Once start and end vertices are chosen, `generateRandomPath` as defined in Algorithm 6 can be used to generate a random path of length a.

For the implementation of Heuristic 3, this algorithm is slightly adapted, as shown in Algorithm 8.

Algorithm 8 Suffix Generation for Heuristic 3

for each $t \in \mathcal{W}_p$ **do**
  $q_s :=$ state after application of t to $\mathfrak{q}$
  randomly select $e = (q, \mathfrak{x}, \mathfrak{y}, q')$ from the least visited edges
  suf :=generateRandomPath(M, q_s, q,$a-1$)
  extend t by sequence of inputs in suf and $\mathfrak{x}$
  increment number of visits for e
end for

In this case, the algorithm keeps track of the visits of edges of the DFSM. A least-visited edge $e = (q, \mathfrak{x}, \mathfrak{y}, q')$ is chosen, and then `generateRandomPath` is used to generate a random path to the source state of e with length $a-1$. This path is extended by the input symbol of e, resulting in a random path that visits e in the last step.

As mentioned earlier, the complexity of uniform random path generation is in $\mathcal{O}(n \cdot |V|^2)$ for the look-up table generation and in $\mathcal{O}(n \log n)$ for the path generation, given a fixed graph and a fixed end vertex. Applied to our case, n is determined by the number of additional states a the implementation is assumed to have. $|V|$ is determined by the number of states in the minimised DFSM representation of the SUT.

The end vertex is not fixed in our implementation of heuristics 2 and 3. Therefore, multiple look-up tables are needed: namely, $|V|$ look-up tables have to be generated, since every vertex of the DFSM graph can be chosen as end vertex for the path generation. Exactly $|\mathcal{W}_p|$ random suffixes have to be generated. This yields an overall runtime complexity in $\mathcal{O}(a \cdot |V|^3)$ for the generation of the look-up tables and an overall runtime complexity in $\mathcal{O}(|\mathcal{W}_p| \cdot a \cdot \log a)$ for the generation of all random suffixes.

4.3.5 Implementation-Efficiency Considerations

In this subsection, we will give some insights into our implementation of the ECPT approach. We will detail some efficiency considerations. We think that these are necessary to guarantee that the test-case generation scales up for complex systems. The main computational effort arises from the reachability analysis that has to be performed to determine the reachable SECs, for the calculation of the IECs and for the calculation of concrete inputs from IECs. The latter has

already been described in Section 4.3.2.3. Another potential scalability problem arises from the use of the W/Wp-method. The number of test cases is significantly affected by the sizes of the characterisation and state-identification sets. Therefore, we present an algorithm to determine minimal versions of these sets.

4.3.5.1 Reachability Analysis

In this section, we present an algorithm for the system's reachability analysis. The goal of this algorithm is to identify all reachability MO-pairs which are needed to define the system's transition relation in INF (cf. Section 2.5.2.6). Thus, we propose an algorithm that performs a reachability analysis and captures all reachable MO-classes $\mathcal{A}_{\text{MO}} = \{\mathbf{a}_1, \ldots, \mathbf{a}_n\}$ together with the conditions for transitions between state classes $\mathbf{a}_i, \mathbf{a}_j$ given as a first-order logic predicate, $\alpha_{i,j}$.

In [HP16a], an algorithm is presented that calculates the INF from a transition relation of arbitrary form. However, the approach presented there relies on two steps: (1) all possible MO-classes, even unreachable MO-classes, have to be enumerated; and (2) the calculation of a Disjunctive Normal Form (DNF) representation from an arbitrary form of the global transition relation $\mathcal{R}$ is necessary. This DNF representation can always be calculated, but in general, this DNF representation will be of exponential complexity.

For the efficient calculation of an INF, we propose an alternative. For our approach, we assume that it is possible to symbolically extract, given a current state s, the conditions for all possible successor states s'.[5] If a global transition relation R of arbitrary form is given, this is always possible. Thus, our assumption is not restrictive compared to the algorithm proposed in [HP16a]. This symbolic extraction can, for example, be performed by using an abstract model representation of a SysML model and the state-machine semantics. In case of a single state machine, the conditions can be extracted considering only the current location the state machine resides in. All outgoing transitions from this location define one condition, $g_{i,j}$. We believe that, for other formal semantics, the extraction of conditions for future evolution, given a fixed state, should be possible as well. For imperative programming languages like C, the Hoare logic [Hoa69] could be used to give another example. In any case, such an extraction should always be at least as easy as the definition of a global transition relation.

The set of concrete value combinations for variables in $M \cup O$ induces a partitioning of the system state space: $\mathcal{B} = \{\mathbf{b}_0, \ldots, \mathbf{b}_m\}$ with $\mathbf{b}_i = \{\vec{m} \mapsto \vec{d}_i, \vec{y} \mapsto \vec{e}_i\}$ for all $i = 0, \ldots, m$. Note that $\mathcal{B}$ is not an SECP as defined in Section 2.5.2.1. An SECP is defined over the set of reachable quiescent states. The state classes $\mathbf{b}_i$ from $\mathcal{B}$ may include state classes that might not be reachable or contain solely transient states. $\mathcal{B}$ must be recursively enumerable but not necessarily finite, although the variable data types and value bounds used in the concrete modelling formalism (e.g., SysML models) usually restrict the set of possible value combinations. The finiteness is no strict requirement, because we do not enumerate all value combinations; instead, we use only the reachable combinations. Thus, in our algorithm, we merely have to assume that the MO-partitioning $\mathcal{A}_{\text{MO}}$ is finite.

If we are able, to extract the conditions to transition from a state s with fixed concrete values for variables in $M \cup O$, the transition relation can conceptually be described by a predicate:

[5] These successor states must not necessarily be quiescent.

$$\mathcal{R} \triangleq \bigvee_{\mathbf{b}_i, \mathbf{b}_j \in \mathcal{B}} \vec{m} = \vec{d}_i \wedge \vec{y} = \vec{e}_i \wedge g_{i,j} \wedge \vec{m}' = \vec{d}_j \wedge \vec{y}' = \vec{e}_j. \tag{4.62}$$

This predicate may be of infinite size, depending on the finiteness of $\mathcal{B}$. This predicate is not explicitly instantiated. Instead, we only extract the terms $g_{i,j}$ from the concrete modelling formalism, whenever needed by our algorithm. The predicate $g_{i,j}$ describes the condition for a RIOSTS to transit from a state class $\mathbf{b}_i \in \mathcal{B}$ to a state class $\mathbf{b}_j$. The source and target states from these classes may or may not be quiescent. Quiescent states are exactly those states in $\mathbf{b}_i$ that fulfil $g_{i,i}$.

The algorithm shown in Algorithm 9 starts with the initial state, s_0. The state class of this state $s_0|_{M \cup O}$ is added to the set of reached states R and to the work list W. Then the algorithm performs a search, as long as the work list is not empty. In every loop iteration, the state classes from W are examined. For each state class, all quiescent successor state classes are calculated using function `successor`. This can be performed in parallel. To this end, let $\|_{x \in X}\ f(x)$ denote the parallel execution of function f for every x in X. If f returns a set of elements from Y (i.e., $f(x) \subset Y$), the expression $\|_{x \in X}\ f(x)$ returns the union of all results: i.e., $\|_{x \in X}\ f(x) = \bigcup_{x \in X} f(x)$. Hence, Line 6 illustrates that the calculation of quiescent successor state is performed in parallel for each state class from the work list W. Consequently, all the reachable MO-pairs that are calculated are added to the result set P. If a new state class $\mathbf{a}_j$ has been found, this class is added to R and W. Finally, P is returned by function `reachability`.

The search for quiescent successor states is performed by the recursive function `successor`. This function is called with the first parameter $\mathbf{a}_i$, representing the starting point of the search. Because multiple intermediate states might exist before a quiescent state is reached, second parameter $\mathbf{b}_c$ represents a state class containing the current intermediate state. $\beta_{i,c}$ describes the condition for a transition from a quiescent state in $\mathbf{a}_i$ to members from $\mathbf{b}_c$. Some of the members of $\mathbf{b}_c$ might be quiescent. Therefore, it is checked whether the predicate $\beta_{i,c} \wedge g_{c,c}$ is solvable. To this end, $\beta_{i,c}$ is added as a constraint to the SMT solver. It is guaranteed, by the way in which `successor` is called, that $\beta_{i,c}$ has a solution. Additionally, $g_{c,c}$ is added as an assumption. Assumptions are additional constraints that the solver has to fulfil. After a call to `solverSolve` these assumptions are dropped, while the constraints are retained. This allows incremental use of SMT solvers. Conflict clauses that are learnt from previous calls to `solverSolve` can be reused in later calls, often resulting in improved runtime performances.

If the SMT solver finds a solution, the predicate $\beta_{i,c} \wedge g_{c,c}$ describes the condition for a transition from states in $\mathbf{a}_i$ to quiescent states in $\mathbf{b}_c$. Thus, $(\mathbf{a}_i, \mathbf{b}_c)$ is a new reachability MO-pair with $\alpha_{i,c} = \beta_{i,c} \wedge g_{c,c}$. While some (or none) of the members in $\mathbf{b}_c$ may be quiescent, others may be transient. Therefore, all conditions $g_{c,j}$ from the transition relation (extracted from the concrete system description) are considered. For each condition, `successor` is called recursively using $\mathbf{b}_j$ as second argument and $\beta_{i,c} \wedge g_{c,j}$ as third argument. As Line 26 indicates, the body of the for-loop can be executed in parallel[6], which makes the whole algorithm highly parallel.

Algorithm 9 is a very efficient and highly parallel algorithm that uses two facts:

[6] In a concrete implementation like in C++ common data structures in the body of the for loop have to be protected from data races. E.g. the update of R has to be locked by some synchronisation means like mutexes and the SMT solver has to be duplicated among the execution threads.

Algorithm 9 Enhanced MO-Reachability Analysis

Input: $\mathcal{R}$ with $g_{i,j}$ calculated on the fly
Output: $\{((\mathbf{a}_i, \mathbf{a}_j), \alpha_{i,j})\}$ all reachability MO-pairs together with predicate $\alpha_{i,j}$

```
1: function REACHABILITY
2:     P := ∅                                  ▷ P is the result set, i.e., the set of MO-pairs and predicates
3:     R := {s_0|_{M∪O}}                                          ▷ R is the set of reached MO-classes
4:     W := {s_0|_{M∪O}}                                          ▷ W is the set of unconsidered MO-classes
5:     while W not empty do
6:         E :=‖_{a_i∈W} SUCCESSOR(a_i, a_i, true) ▷ Call successor for each unconsidered MO-class
7:         W := ∅
8:         for each ((a_i, a_j), α_{i,j}) ∈ E do
9:             P := P ∪ {((a_i, a_j), α_{i,j})}
10:            if a_j ∉ R then
11:                R := R ∪ {a_j}
12:                W := W ∪ {a_j}
13:            end if
14:        end for
15:    end while
16:    return P
17: end function
18:
```

Input: $\mathbf{a}_i$ starting MO-class for successor analysis
Input: $\mathbf{b}_c$ state class containing intermediate states
Input: $\beta_{i,c}$ predicate describing the condition for a transition from $\mathbf{a}_i$ to $\mathbf{b}_c$
Output: $\{((\mathbf{a}_i, \mathbf{a}_j), \alpha_{i,j})\}$ all reachability MO-pairs that start at $\mathbf{a}_i$ and go through $\mathbf{b}_c$

```
19: function SUCCESSOR
20:     P := ∅                                                          ▷ P is the result set
21:     solverAddConstraint(β_{i,c})
22:     solverAssume(g_{c,c})
23:     if solverSolve( ) then
24:         P := P ∪ {((a_i, b_c), β_{i,c} ∧ g_{c,c})}
25:     end if
26:     for ‖ each g_{c,j} from R do
27:         solverAssume(g_{c,j})
28:         if solverSolve( ) then
29:             P := P∪successor(a_i,b_j,β_{i,c} ∧ g_{c,j})
30:         end if
31:     end for
32: end function
```

1. No global transition relation is needed. Instead, during reachability analysis, conditions for a transition from the current state can be extracted from the concrete description means. Usually, these conditions are magnitudes smaller than the global transition relation. This in turn accelerates the SMT solver calls.
2. The calculation of successor states can be performed in parallel, resulting in good runtimes on multi-processor platforms.

Example 24. Consider the model from case study Route 12a (Lyngby). The route controller has 18 variables in $M \cup O$. The domain of these variables reaches from the boolean domain to the integer interval $[0, 5]$. This leads to an overall number of possible value combinations for variables $M \cup O$ of $283,435,200$. Additionally, many of the complex guard conditions used in the state machine describing the behaviour of our route controller use boolean operators $\wedge$ and $\vee$. This will result in an exponential explosion of the transition-relation predicate when transformed to a DNF representation in a naive way. Thus, an enumeration of all possible even unreachable value combinations (state classes) and the construction of a DNF representation is far from realisable for the Route 12a (Lyngby) case study.

The reachability analysis shown in Algorithm 9 reveals that merely 28 MO-states are reachable. In our reachability analysis, only a very small subset of conditions must be considered. The SMT instances that have to be solved by the SMT solver are drastically smaller than they are for an approach that considers global transition relations. Because the runtime of a typical SMT solver depends on the size of the SMT instance, this will usually result in a significant improvement of runtime.

4.3.5.2 Calculation of Input Equivalence Classes

Algorithm 10 describes an efficient way to calculate an initial IECP (cf. Section 2.5.2.7). This is performed by finding all possible functions $f : \mathrm{IDX} \to \mathrm{IDX}$, for which the predicate $\Phi_f \triangleq \bigwedge_{i \in \mathrm{IDX}} \alpha_{i,f(i)}$ is solvable.

The overall number of possible functions from IDX to IDX is $|\mathrm{IDX}|^{|\mathrm{IDX}|}$. Thus, an enumeration of all possible functions is infeasible for all but the most trivial of systems. We therefore propose to recursively construct functions f and to use an incremental SMT solver and parallel execution for an efficient calculation of all solvable predicates of the form Φ_{f_i}. The recursive construction of a function f can be aborted as soon as a constructed partial function $f' : \mathrm{IDX} \not\to \mathrm{IDX}$ results in an infeasible $\Phi_{f'}$.

The function `calcIECP` shown in Algorithm 10 merely calls the recursive function `recCalcIECP` with the default arguments. The recursive function `recCalcIECP` is used to recursively enumerate all possible functions f. This is done by mapping all indices from $\mathrm{IDX} = \{1, \ldots, n\}$ starting at one to all possible values from the image IDX. The second parameter of `recCalcIECP` denotes the current index from the domain, which shall be assigned in the current recursive function call of `recCalcIECP`. A value $i > n$ indicates that a complete (and solvable) mapping f has been constructed. In this case, the recursion terminates and the third parameter Φ, which describes the predicate that is constructed simultaneous to f, is returned as a new predicate Φ_f describing one IEC X_f. If parameter $i \leq n$, f is still a partial mapping and Φ is a predicate describing a superset of a potential IEC X_f. Note that it is guaranteed by the way `recCalcIECP` is called that Φ has at least one solution. In this case, we have to recursively consider all possible mappings f that can be built by assigning i with all possible values from the image set IDX. All assignments of the form $i \mapsto j$ are checked for plausibility by adding $\alpha_{i,j}$ as an assumption to the SMT solver.

Algorithm 10 Calculation of Input Equivalence Classes

Input: $\mathcal{R}$ in INF
Output: $\{\Phi_{f_1}, \ldots, \Phi_{f_m}\}$ all solvable predicates Φ_{f_i} that describe IECs of the form X_{f_i}

```
1: function calcIECP
2:     return recCalcIECP(R,1,true)
3: end function
4:
```

Input: $\mathcal{R}$ in INF
Input: $i \in$ IDX index of current MO-class to consider, with IDX $= \{1, \ldots, n\}$
Input: Φ predicate describing a solvable superset of the potential IEC
Output: $\{\Phi_{f_1}, \ldots, \Phi_{f_l}\}$ solvable predicates Φ_{f_i} that describe IECs of the form X_{f_i}

```
5: function recCalcIECP
6:     if i > n then
7:         return {Φ}
8:     end if
9:     R := ∅
10:    solverAddConstraint(Φ)
11:    for ‖ each j ∈ IDX do
12:        solverAddAssumption(α_{i,j})
13:        if solverSolve( ) then
14:            R := R∪ recCalcIECP(R, i + 1, Φ ∧ α_{i,j})
15:        end if
16:    end for
17:    return R
18: end function
```

If $\Phi \wedge \alpha_{i,j}$ is solvable, the recursion is continued; otherwise, it is aborted because assignment $i \mapsto j$ led to a partial mapping f' whose predicate $\Phi_{f'}$ has no solution. Again, our algorithm can use parallel execution. The for-loop can be executed in parallel for all possible assignments, as indicated by operator $\|$ in Line 11.

4.3.5.3 Calculation of Minimal Characterisation and State-identification Sets

The number of test cases generated by the W/Wp-method depends largely on the size of the characterisation and state-identification sets. Unfortunately, the characterisation set as calculated by Algorithm 2 is not guaranteed to be minimal.

The problem of finding a minimal characterisation set can be reduced to the *minimal-hitting-set problem*. The *hitting-set problem* is the problem of finding a solution $W \subset U$ (called the hitting set) which is a minimal subset of elements from the universe $U = \{1, \ldots, n\}$, such that for a family of subsets of U: $\{U_1, \ldots, U_m\}$ with $U_i \subset U$ for all $i = 1, \ldots, m$, all subsets contain at least one element of W:

$$\forall U_i \in \{U_1, \ldots, U_m\} : U_i \cap W \neq \emptyset. \tag{4.63}$$

In the remainder of this section, we abbreviate the minimal-hitting-set instance $\{U_1, \ldots, U_m\}$, as introduced above, as $\{U_i\}$.

The *minimal hitting set* is a hitting set W for which no subset exists which is itself a hitting set. The minimal-hitting-set problem is known to be NP-complete [Kar72].

The hitting-set problem can be solved by a SAT solver. To this end, we formulate the hitting-set problem as a SAT instance. Let b_u denote a boolean variable representing element u from the universe $U = \{1, \ldots, n\}$. The predicate to be solved Φ can be defined as follows:

$$\Phi \triangleq \bigwedge_{i \in \{1,\ldots,m\}} \left(\bigvee_{u \in U_i} b_u \right). \tag{4.64}$$

The minimal hitting set can be found by using an SMT solver that is able to find a solution under additional optimisation constraints. Thus, the problem can be formulated as follows:

$$\texttt{MHS}(\{U_i\}) \equiv \texttt{solve(}\Phi\texttt{) and minimise(} \sum_{u \in U} b_u \texttt{)}. \tag{4.65}$$

We use the SMT solver Z3 [BPF15] to solve this SMT instance with the optimisation goal $\texttt{minimise(}\sum_{u \in U} b_u\texttt{)}$. $\sum_{u \in U} b_u$ has to be understood as an integer addition of boolean variables: i.e., $\sum_{u \in U} b_u$ is the number of boolean variables assigned true.

The calculation of a minimal state-identification set for DFSM state q_i can be performed using Algorithm 11. First of all, a minimal-hitting-set instance $\{U_i\}$ is constructed. To this end, a subset U_j from the universe, which is the characterisation set CS in this case, is constructed for every other DFSM state q_j. U_j contains all input traces from CS that distinguish q_i and q_j: I.e.,

Algorithm 11 Algorithm for the Calculation of a Minimal State-identification Set W_i

Input: $M = (Q, \mathfrak{q}, \Sigma_I, \Sigma_O, \delta, \omega)$ a minimal DFSM
Input: CS characterisation set of M
Input: q_i the state of M to be identified
Output: W_i minimal subset of CS identifying q_i
  function STATEIDENTIFICATIONSETMINIMAL
    $\{U_i\} := \{U_j | q_j \in Q, q_j \neq q_i, U_j = \{\sigma \in \mathrm{CS} | \omega^*(q_i, \sigma) \neq \omega^*(q_j, \sigma)\}\}$
    $\mathrm{W}_i :=$`MHS`$(\{U_i\})$
    return W_i
  end function

different output traces are generated when the input trace is applied to both states. From each of these U_j, at least one input trace has to be selected to distinguish q_i from every other state. Thus, the minimal-hitting-set algorithm (Equation 4.65) can be used.

Algorithm 12 Algorithm for the Calculation of a Minimal Characterisation Set CS

Input: $M = (Q, \mathfrak{q}, \Sigma_I, \Sigma_O, \delta, \omega)$ a DFSM
Input: $\{\mathcal{C}_1, \ldots, \mathcal{C}_k\}$ the P_k tables of M
Output: CS minimal characterisation set of M
  function CHARACTERISATIONSETMINIMAL
    $\Sigma := \{\sigma_{i,j} | q_i, q_j \in Q, q_i \not\sim q_j, \sigma_{i,j} \in \mathrm{DISTINGUISHINGSEQUENCE}^*(M, \{\mathcal{C}_1, \ldots, \mathcal{C}_k\}, q_i, q_j)\}$
    $\{U_i\} := \{U_{i,j} | q_i, q_j \in Q, q_i \not\sim q_j, U_{i,j} = \{\sigma \in \Sigma | \omega^*(q_i, \sigma) \neq \omega^*(q_j, \sigma)\}\}$
    CS :=`MHS`$(\{U_1, \ldots, U_m\})$
    return CS
  end function

The calculation of a minimal characterisation set can be performed quite similarly. Our approach is shown in Algorithm 12. First of all, we define our universe to get a minimal hitting set from as the set Σ of all possible distinguishing sequences of minimal length. This set can be calculated by considering all pairs of distinguishable states q_i, q_j and collecting the distinguishing sequences of minimal length. Therefore, we use a slightly modified version `distinguishingSequence*` of Algorithm 3. Algorithm 3 calculates only one possible distinguishing sequence of minimal length. This calculation is indeterministic because of the use of the `elem`() operator in Algorithm 3. Let `distinguishingSequence*` be the version of Algorithm 3 that returns the set of all possible distinguishing sequences that result from all possible choices for the `elem`() operator.

The minimal-hitting-set instance is now constructed with all subsets $U_{i,j}$ for distinguishable states q_i, q_j. $U_{i,j}$ contains all input traces from Σ that produce different outputs when applied to q_i and q_j. Note that this may include traces that are not returned by `distinguishingSequence*` when called for q_i and q_j: i.e., traces which have traces from the result set of `distinguishingSequence*` as prefix. In this way, we ensure that prefixes in the minimal characterisation set CS are dropped, as the minimal-hitting-set solution will not contain these prefixes.

5 A Novel Approach to Mutation Analysis for HW/SW Integration Tests

This chapter introduces our novel approach for test-strength evaluation of HSI tests. First, we briefly depict the state-of-the-art of mutation analysis and then motivate and present our new mutation-analysis method based on SystemC mutations.

5.1 Mutation Analysis State-of-the-Art

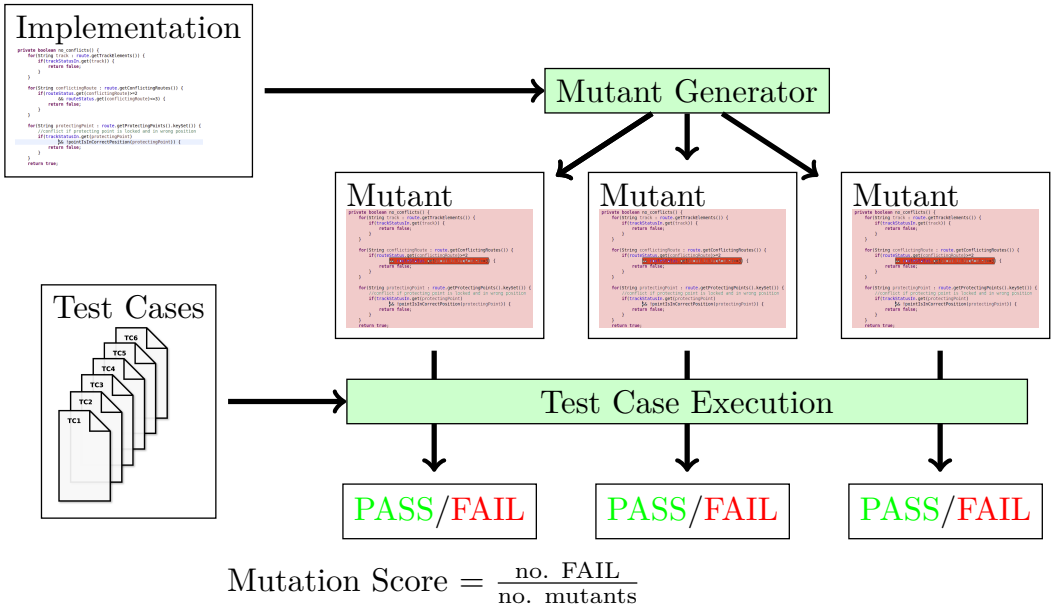


Figure 5.1: Overview of the Mutation Analysis Process

Different approaches to *mutation analysis* and *mutation testing* exist. By *mutation analysis*, we understand the process of generating mutants to evaluate the mutation score of a test suite. An overview of this process is depicted in Figure 5.1. Given an implementation of the SUT which passes the test suites under consideration, faulty versions of the SUT implementation are generated using a *mutant generator*. Usually, the *mutant generator* is a tool that systematically injects faults into the implementation. An implementation with injected faults is called *mutant*. Usually, the fault injection is performed using *mutation operators*. A *mutation operator* is an operator that introduces syntactic changes to the implementation. These operators are defined by the syntactic elements (locations) they can be applied to and by the change that is introduced to

the elements they are applicable to. In general, a mutant generator can apply mutation operators in multiple locations of the implementation. This results in a huge number of possible mutants. A mutant is called a *first-order mutant* if the mutant is obtained by the application of one mutation operator in exactly one location. Most mutant generators produce first-order mutants only. In contrast to this, a *high-order mutant* is a mutant that is obtained by the application of mutation operators in more than one location of the implementation.

Note that the term implementation does not restrict mutation analysis to typical implementation languages. An implementation can be considered a model of the SUT behaviour, and every model can be considered an implementation of the SUT. Thus, mutation analysis is applicable to modelling languages like UML/SysML, DFSMs but also to typical SW-implementation languages like Java or C/C++.

The evaluation approach we call *mutation analysis* in this work must not be confused with an approach widely known as *mutation testing*. By *mutation testing*, we mean the use of mutants to (automatically) generate test cases. In mutation testing, an implementation/model of the SUT is mutated using mutation operators, as introduced above. For the resulting mutant, one or more test cases are generated that are able to *kill* this mutant. A mutant is said to be killed if the application of at least one test case of a test suite fails for the mutant. Otherwise, the mutant is said to be *alive*.

In the next subsections, we categorise and give an overview of state-of-the-art mutation-analysis approaches. Depending on the model/implementation language used, the mutation types can be categorised as model mutations, HW mutations and SW mutations. The following subsections roughly present the overall idea of the different mutation types. A detailed discussion of related work in this area is given in Section 7.4.

5.1.1 Model Mutations

5.1.1.1 DFSM Model Mutations

For the domain of DFSMs, a very simple fault model exists. As mentioned in Section 2.4.2, there are two types of faults: *transfer* and *output* faults. These faults relate to the simple fault operators of changing the target state of a transition or changing the output of a transition. Additionally, a DFSM can be manipulated by deleting a state or introducing extra states. In this case, the transition and output functions of the DFSM have to be manipulated as well. Hence, state deletion/introduction of extra states can be considered a variant of high-order mutation. This fault model has widely been used [Cho78, FvBK+91] to compare testing methods. This model, however, applies only at the level of DFSM. In our evaluation, we do not use this fault-model as the primary evaluation means, as the resulting mutants are of limited use. Our method is proven to be complete for erroneous implementations with at most m states. Hence, the mutation score is 100% for erroneous implementations with at most m states in their DFSM abstraction. However, in Section 6.3, we use DFSM mutations for the evaluation of the suffix heuristics to explore additional states. There, we intentionally introduce additional states to ensure that the SUT is outside the fault domain and then investigate the mutation score considering the different suffix heuristics.

5.1.1.2 UML Model Mutations

Another model mutation approach is implemented in the tool, MoMuT::UML [KST+15]. The authors of [KST+15] use UML-specific mutation operators to generate mutants from UML models. Instead of mutation analysis, MoMuT::UML follows the mutation-testing approach. Based on the generated mutants test cases are selected to kill these mutants. The mutation operators of [KST+15] are applicable to our UML/SysML state machines as well. In preliminary work, we used these mutation operators to conduct our experiments. However, we observed that the mutations tend to introduce erroneous behaviour that is too easy to detect. RT achieved very high mutation scores of up to 85%. This demonstrates that, at least for our case studies, UML mutation operators are of limited use. We are interested in mutations that are, (1) hard-enough to be detected, and (2) valid surrogates for real faults to be expected in an implementation of the SUT. The latter requirement led to doubts that model mutations are an appropriate evaluation means for our testing methodology. We believe that model mutations are a valid means to automatically generate test goals from a test model. However, the use of mutation operators for test strength evaluation of HSI tests should always consider mutations that are to be expected in real implementations. Since UML/SysML is an abstract description means, we believe that the exclusive application of model mutation operators will miss many of the typical faults to be expected in real-world scenarios. The roots of these faults will first emerge on a more concrete level: e.g., at the HW-description or SW source-code levels.

5.1.2 SW-Mutation Analysis

SW source code can be a target of mutation analysis. Source code is by definition a means of concrete formal description for the behaviour of (a part of) a system. This formal description can very easily be manipulated. Many approaches exist, ranging from the earliest mutation-analysis experiments [DLS78] to mutation operators for modern, high-level OO languages like Java [MOK05]. The main advantage of SW-mutation analysis is that the mutated program is not an abstract model of the expected SUT, but is most likely the test target itself. The faults to be discovered by a SW testing approach are most likely introduced in the implementation phase and manifest themselves as faults in the source code of the SUT. Some evidence exists to show that faults introduced using SW mutation operators relate to real faults [ABL05, JJI+14].

5.1.3 HW-Mutation Analysis

The validation and verification of integrated circuits is a major research area. In this domain, *fault injection*, a technique that injects faults on the bit level of an integrated circuit, is widely applied. These faults represent HW faults that can be caused by hardware defects introduced by design errors or in the manufacturing process. Fault injection is strongly related to mutation testing and can be understood as the HW-related counterpart of SW-mutation analysis.

In fault injection, the injected faults can be categorised as *stuck-at faults* and transient faults. *Stuck-at faults* are assumed to be permanent, whereas *transient-fault* models introduce faults that happen at one point in time. It is assumed that these faults only appear temporarily and disappear in a consecutive clock cycle. Transient faults usually model faults that are triggered by external events (e.g., radiation). The focus of this fault model is mainly to evaluate the fault

tolerance of a system. Fault tolerance is out of the scope of this work; therefore, we consider only permanent errors, using the stuck-at fault model.[1]

5.2 HW/SW-Mutation Analysis

5.2.1 The Need for a Formal Fault Model of Typical HW/SW Integration Errors

The previous section presented mutation-analysis approaches that are based on syntactically seeded faults (mutation operators). Faults can be seeded at different levels of abstraction and in different parts of the system. As shown in [ABL05, JJI+14], software mutations are good surrogates for real SW-related faults. The mutation score achieved by SW-mutation analysis is therefore a good measure of the test strength of a SW test approach. However, the validity of these results is limited to the experimental evaluation of software tests. There is no evidence that the results gained by SW-mutation experiments are sound for the evaluation of HSI or system tests. Apart from this, HW-related faults that may be introduced in the manufacturing process or by bad design can be evaluated using fault injection and the stuck-at fault model, for example.

An integrated HW/SW system will surely contain SW-related and HW-related errors. Additionally, it must be expected that, during HSI additional faults can be introduced to the integrated system that do not only result from errors in software or hardware. These errors may result from mismatches in the hardware and software designs. Unfortunately, there is no formal fault model to the best of our knowledge that focuses on typical HSI faults.

To overcome the aforementioned problem, we propose a fault model to mimic typical HSI faults. This fault model is based on mutation operators applied to system specifications described in the SystemC design language. This fault model is realised by a mutant generation tool to allow for automation of mutation analysis for HW/SW integration errors.

5.2.2 Requirements for a Mutant Generation Tool of Typical HW/SW Integration Errors

A mutant generator aiming at mutation analysis of typical HW/SW integration errors should fulfil certain requirements. In the following, some of the requirements that we consider to be useful are listed. A mutant generator for mutation analysis aiming at the evaluation of HSI tests should do the following:

- cover software-related errors,
- cover hardware-related errors,
- cover errors resulting from mismatches in the software and hardware design,
- cover errors resulting from the faulty integration of software and hardware modules (e.g., wrong configuration data, misconfigured communication of modules),

[1] We assume that the fault tolerance of the SUT is evaluated and validated using other means than our MBT approach.

- cover errors resulting from special characteristics of the target platform (e.g., operating system, hardware specification, byte order),
- be applicable to every valid SystemC model,
- be easy to use,
- generate a large number of mutants, and
- generate syntactically correct mutations.

Note that we intentionally ignore the class of runtime errors for our evaluation approach. Runtime errors usually result from incorrect code (like invalid memory addresses for read-and-write operations, violation of array boundaries, race conditions) which may cause erroneous behaviour. However, runtime errors are not always detectable; in the worst case, runtime errors are observable only by chance. The nature of runtime errors makes it hard to detect this class of defects by functional tests. In fact, the absence of runtime errors is usually referred to as a non-functional requirement of a system. It is a well-known fact that runtime errors are best detected by abstract interpretation [KF16]. Therefore, runtime errors are out of the scope of our evaluation approach. We assume that abstract interpretation is used before HSI tests are applied; therefore, the SUT is supposed to be free of runtime errors.

5.2.3 A SystemC Mutation Tool

The generation of SystemC mutants is performed in a systematic and automated manner. To this end, a tool based on the clang LibTooling library[2] has been implemented. This tool is a back end to the well-known clang compiler. The clang compiler is used to parse C/C++ code and generates an abstract syntax tree (AST). Based on this AST mutation, operators are implemented: Syntactic patterns are matched and replaced to generate single-fault mutations. The next paragraphs list and group all mutation operators that were used for our experiments. The descriptions were taken from our previous publication [HHP17] with slight modifications to fit this work.

Ordinary mutation operators Mutation operators are commonly used in SW mutation tools [MOK05, Jus14]. These mutation operators include the replacement of logical, arithmetical and relational operators and the replacement of conditions in while and if statements with the constants true and false.[3] These operators are used to mimic software-related errors that may still be present in the HSI phase. Though many of the software-related errors are expected to be uncovered by unit testing, these errors must still be considered for HSI testing because the detection of all software errors by unit testing cannot be guaranteed. Furthermore, it is possible that software-related errors are introduced during the HSI phase, because the software code implementing some of the interfaces between modules might be erroneous itself. In the remainder of this work, we refer to these classical software-mutation operators as *ordinary mutation operators*.

Since the focus of this work is to demonstrate the applicability of our test method to HW/SW integration tests, we augmented the ordinary mutation operators with mutations that we would expect in typical HW/SW integration scenarios.

[2] see `http://clang.llvm.org/docs/LibTooling.html`

[3] Besides these "traditional" mutation operators [MOK05] also supports object-oriented (OO) mutation operators. These operators have been neglected in this work, because the different implementations of our case studies did not use the object oriented concepts that are targeted by OO mutation operators.

Byte-order mutation A potential error in distributed systems communicating over network protocols results from the wrong interpretation of data. Typically, data is sent in big-endian (network) byte order, while the data is then interpreted in little-endian (host) byte order on the local machine. To capture potential flaws resulting from this mismatch, we introduce a mutation operator that replaces every assignment of integer variables with an assignment using a right-hand side expression with an inverted byte order.

Listing 3 Example of Byte Order Mutation

```
int32_t x;//little endian, width 32 bit
x=13;
//assignment will be replaced by
x=htobe32(13);
// htobe32(n) transforms n in host byte order
// to the same integral number represented
// in 32 bit big endian byte order
```

Precision mutation Every floating-point operation is performed with a fixed precision. A mismatch in precision between the specification and the actual implementation might result in unexpected behaviour.[4] To mimic errors resulting from wrong precision, we use a mutation operator that replaces constant floating-point literals with another constant floating-point literal of different precision. In C and C++, this subtle error can easily be introduced.

Example 25. For example, the following expression used in a condition of an if-statement is a candidate for the proposed mutation operator.

Listing 4 Example of Precision Mutation

```
float x,y;
...
if(x*0.1f<y) { //literal will be replaced: if(x*0.1<y) {
   ...
}
```

The above example demonstrates that a subtle change in the floating-point literal can introduce non-conforming behaviour. In C/C++, the operations are automatically performed with the least required precision. Therefore, the expression `x*0.1f<y` is evaluated with single precision (float type), while the expression `x*0.1<y` is evaluated with double precision (double type). Because there is no exact binary representation of the real number 0.1, the evaluation of both conditions will differ for some values of `x` and `y`.

Sensitivity mutation Another source of errors is related to concurrency and timing. In SystemC, modules have a sensitivity list: i.e., a list of signals. Whenever the value of a signal in the module's sensitivity list changes, the method of the module is executed. An easy way to introduce timing and concurrency errors is to manipulate the sensitivity list of modules. Therefore, a simple mutation operator removes signals from a module's sensitivity list. This results in update methods being called in the wrong order or not at all.

[4] We consider a mismatch in precision a real error, while in real world examples a mismatch only resulting from different precisions might be negligible.

Example 26. Consider, for example, the excerpt of the SystemC implementation of the airbag controller shown in Listing 5. In this case, the sensitivity mutation would prevent the firing of the airbag, because the update method of the `crash_indication` sub-module would not be called. This sub-module is responsible for firing the airbag as soon as the crash counter variable (`crash_ctr`) exceeds the threshold (`CRASH_THRESHOLD`).

Listing 5 Example of Sensitivity Mutation

```
SC_MODULE(crash_indication) {
   sc_in<int> crash_ctr;
   sc_in<int> defect;
   sc_out<int> fire;

   SC_CTOR(crash_indication) {
      SC_METHOD(update);
      sensitive<<crash_ctr;
      //the previous line will be removed by the
      //sensitivity mutation operator
   }

   void update() {
      if(fire || defect) {
         return;
      }
      if(crash_ctr>=CRASH_THRESHOLD) {
         fire=1;
      }
   }

};
```

Stuck-at fault mutation Typically, hardware-related faults are modelled using fault models on a bit level. Stuck-at faults model errors in which one signal is constantly stuck at the logical value one or zero. These faults can be caused by hardware defects introduced in the manufacturing process but also by design errors. Therefore, these potential design errors are in the scope of HW/SW integration tests.

The stuck-at mutation is implemented by overriding the *read* method of SystemC signals. For each stuck-at mutation, a single bit of a single signal is constantly overridden with 1 or 0 whenever the signal's value is read.

Switch-ports mutation The last mutation operator we consider is an operator that interchanges the connection of signals to ports. In SystemC, a module can have ports for modelling the module's inputs and outputs. Signals are connected to the ports, supplying the value that can be read or written by a module through its port.

The switch-ports mutation swaps the connection of two signals to ports of the same type and thus effectively swaps the variables that a module reads from or writes to. This type of fault mimics a possibly wrong configuration of the integrated system, where a module reads from a wrong memory location. The mutation operator captures a wrong wiring of input ports and two modules that communicate with inconsistent network protocol definitions.

Example 27. For example, for the CSM, the two system inputs `V_est` and `V_mrsp` will be switched by the mutation operator leading to wrong control decisions in the mutated SUT.

Listing 6 Example of Switch-ports Mutation

```
//the system under test
ceiling_speed_monitor sut("ceiling_speed_monitor");

//signals for the system's inputs
fi_signal<float> V_est;
fi_signal<float> V_mrsp;

//signals for the system's outputs
fi_signal<signed int> DMICmd;
fi_signal<bool> DMIdisplaySBI;
fi_signal<signed int> TICmd;

int sc_main(int argc, char* argv[]) {

   //wrongly connected input signals
   sut.V_est(V_mrsp);
   sut.V_mrsp(V_est);

   //connect output signals to output ports
   sut.DMICmd(DMICmd);
   sut.DMIdisplaySBI(DMIdisplaySBI);
   sut.TICmd(TICmd);
   //run the test suite
   int result = Catch::Session().run(argc,argv);
   return result;
}
```

5.2.3.1 Implementation Details of the SystemC Mutation Tool

The mutation tool uses clang's LibTooling framework. LibTooling is a C++ library provided by clang to give developers access clang's internal infrastructure.

LibTooling offers, among other functionality, parsing of C/C++ programs; access to the abstract syntax tree (AST); tracing of AST nodes to source-code locations; and iteration through the AST by visitors, matchers, and source-to-source transformations.

Our SystemC mutation tool makes excessive use of *matchers*. *Matchers* are predicates that describe patterns of the AST. Every occurrence of such a pattern in an AST that matches is subject to a mutation operator. The mutation is then implemented in a callback function that gets called for every match of a matcher.

Listing 7 and Listing 8 are examples of the matchers used in the SystemC mutant generator. These matchers make it possible to specify patterns to be matched by predefined C macros. These matchers can be combined to allow the specification of arbitrary complex patterns that cover every possible AST structure of parsed C/C++ programs. As the reader has probably observed, the matcher used to find integer assignments in a program to be mutated is quite complex. This is mainly due to the complexity of the AST of C++ programs. An integer assignment in C++ can either be a binary operator of an ordinary variable or a class-member variable, or an overloaded operation call. This is especially relevant for the use of signals and

ports in SystemC models. While assignments in this case look like ordinary binary operations, they are overloaded operation calls of the base classes `sc_port` or `sc_signal`.

Listing 7 Overview of Binary Operator Matchers

```
StatementMatcher relOpMatcher =
      binaryOperator(
            anyOf(
                  hasOperatorName("<"),
                  hasOperatorName("<="),
                  hasOperatorName(">"),
                  hasOperatorName(">=")
            )
      ).bind("binop");

StatementMatcher aritOpMatcher =
      binaryOperator(
                  anyOf(
                        hasOperatorName("+"),
                        hasOperatorName("-"),
                        hasOperatorName("*"),
                        hasOperatorName("/"),
                        hasOperatorName("%")
                  )
            ).bind("binop");

StatementMatcher logicalOpMatcher =
      binaryOperator(
                  anyOf(
                        hasOperatorName("&&"),
                        hasOperatorName("||")
                  )
            ).bind("binop");

StatementMatcher binOpMatcher = anyOf(relOpMatcher, aritOpMatcher,
    logicalOpMatcher);
```

The implementation of our SystemC mutant generator together with the SystemC models is available under `www.mbt-benchmarks.org`.

Listing 8 Complex Example of Matchers Used for the Byte-order Mutation

```
StatementMatcher intAssignmentMatcher =
      binaryOperator(
            hasOperatorName("="),
            hasLHS(
                  anyOf(
                     declRefExpr(
                        to(
                           varDecl(
                              hasType(isInteger())
                           ).bind("varDecl"))),
                     memberExpr(
                        member(
                           hasType(
                              (TypeMatcher) anyOf(
                                 isInteger(),
                                 referenceType(pointee(isInteger()))))))).bind("
                                     member")))).bind("intAssignment");

StatementMatcher intAssignmentOverloadedMatcher =
      cxxOperatorCallExpr(
         hasOverloadedOperatorName("="),
         hasArgument(0,
            anyOf(
               memberExpr(
                  member(
                     hasType(
                        cxxRecordDecl(
                           (internal::Matcher<CXXRecordDecl>) anyOf(
                              isSameOrDerivedFrom("sc_port"),
                              isSameOrDerivedFrom("sc_signal"))))).bind("
                                  member"),
               declRefExpr(to(varDecl(
                  hasType(
                     cxxRecordDecl(
                           (internal::Matcher<CXXRecordDecl>) anyOf(
                              isSameOrDerivedFrom("sc_port"),
                              isSameOrDerivedFrom("sc_signal"))))).bind("
                                  varDecl"))))).bind("intAssignmentOverloaded"
                                  );
```

6 Experimental Evaluation

The experimental evaluation of our ECPT approach with its extensions and heuristics presented in Section 4.3 is described in this chapter. The evaluation is based on previous work that has been published in [BHH+14, BHP+14, HHP15, PHH16a, HHP17]. Our first experiments [HHP15] were based on Java software mutations and considered the CSM and airbag-controller case studies. In that work, we compared different approaches for the test-case generation, including the use of refined IECPs for boundary-value testing. The results obtained in [HHP15] indicate that the use of a refined IECP results in an increase of test strength that is relatively expensive. Therefore, we consider our randomised strategy, which includes boundary values in 50 percent of the cases, favourable, because the number of test cases is not increased by this approach.

In [PHH16a], we investigated the test strength of the ECPT approach for both interlocking case studies, again using Java software mutations. To confirm the good results in the scope of HSI testing, we proposed our novel evaluation approach in [HHP17] and applied this approach for experiments considering the CSM and airbag controller case studies.

Therefore, the experimental evaluation described in this chapter is mainly based on the experiments published in [HHP17] where we used the SystemC mutant generator presented previously. We took parts of the experimental description and threats to validity from [HHP17] and rephrased it to fit this work. Furthermore, the experimental evaluation presented here extends the experiments from [HHP17] in the following way:

1. The interlocking case studies are considered in this work: The experiments are extended to the route-controller case studies Route 2011 (example network) and Route 12a (Lyngby). Experiments for these case studies have been published in [PHH16a]. These experiments were based on Java SW mutations.
2. The strategies for additional states exploration are first presented and evaluated in this work. We compare the three proposed suffix heuristics using DFSM mutations in Section 6.3.
3. An optimised implementation has been used, resulting in fewer test cases using the Wp-method and minimal state-identification sets and a minimal characterisation set, as described in Section 4.3.5.3.

6.1 Experimental Setup

6.1.1 Compared Strategies

For our experimental evaluation, we are interested in a comparison of different testing strategies. We want to gain empirical evidence for the superiority of the ECPT approach in general and for the heuristics that we proposed in Chapter 4 in particular. We use RT as a minimal-strength benchmark to be surpassed by our ECPT approach.

The following list gives an overview of all testing strategies to be evaluated in our experiments:

STRAT-1 The original complete ECPT strategy, as introduced in Section 2.5, which uses the coarsest IECP with fixed representatives for every IEC.

STRAT-2 An extension of STRAT-1 by random concrete input selections from each IEC, whenever the IEC is referenced in a symbolic test case.

STRAT-3 An extension of STRAT-1 using random selection and boundary-value selection. Randomly select concrete inputs from an IEC used in symbolic test cases, such that 50% of the inputs come from the interior of the IEC and the rest come from the boundary of the IEC.

STRAT-RND We use RT as a minimal benchmark. We expect our ECPT approach to at least reach the same mutation score as conventional random testing. This strategy uses randomly generated sequences of random input vectors. For a fair comparison we generate an RT test suite of the same shape as every ECPT strategy that we apply. Same shape means that for every ECPT test suite composed of n test cases of lengths $l_1, \ldots, l_n$, where the i-th test cases has length l_i, we generate a similar RT test suite with n test cases where the i-th test case is a sequence of exactly l_i input vectors.

STRAT-n$_r$ Complementary to the way in which concrete input selection is performed, the coarsest IECP can be refined using requirement-based refinement (cf. Section 4.3.1.2). Therefore, STRAT-n$_r$ denotes the set of strategies that is obtained by using STRAT-1, STRAT-2 or STRAT-3 with a requirement-based IECP refinement. This strategy has been used exclusively for the CSM case study because the existing requirements regarding different speed intervals lend themselves for a natural refinement of the IECP.

We apply the Wp-method with the assumption that the number of states in the DFSM abstraction of the implementation is less or equal to the number of states in the test model, thereby to keep the number of test cases at a minimum. Hence, no additional states in the implementation are assumed for the experiments in the following section. Section 6.3 presents the experimental evaluation of our heuristics, addressing the additional states problem.

6.1.2 Conduction of Experiments

For the experimental evaluation, we created correct SystemC implementations from each model of the case studies. The implementation was performed by hand in a straightforward way. Next, mutants were automatically generated from each implementation using our SystemC mutation tool. For each case study, we applied the subset of the SystemC mutation operators that was reasonably applicable. In particular, this means that the precision mutation operator was applied to the CSM and airbag controller only, since these case studies use floating-point calculations. The sensitivity mutation was applied to the airbag-controller case study to internal signals and events only. The other case study implementations used a programming paradigm in which all input variables were cyclically polled, which makes the sensitivity-mutation operator inapplicable to these implementations. All other mutation operators, as presented in Chapter 5, were fully applied to all case studies.

Note that the mutants generated by our mutation tool may accidentally be I/O-equivalent. Therefore, in an additional step, the generated mutants have been investigated manually and all I/O-equivalent mutants were discarded for the experiments. This process could be accelerated by first running all test strategies against the mutants. All mutants that were not killed by any of the

test suites were manually investigated. For the manual investigation, each remaining mutant was compared to the original implementation and we checked to see the mutant was I/Oequivalent or not. In some cases, this decision was trivial. For example, many similar mutations were I/O-equivalent because the range of valid input/output values restricts the number of possible stuck-at mutations. A stuck-at-zero mutation of the most significant bit of a variable that never gets a value that is high enough for this bit to be set is trivially I/O-equivalent. For the CSM, for example, 38 mutants, which were not killed by any of the test strategies, remained after the first test run. Of these, 29 I/O-equivalent mutants could be identified. These were discarded from the experiments.

Afterwards, the derived test cases were executed against the remaining mutants to measure the mutation score of the test suite: i.e., the ratio of mutants that were "killed" by the test suite to the total number of non-I/O-equivalent mutants.

Since some of the strategies depend on the utilisation of random values for the concrete value selection, each of their test suites were generated and executed against the mutants by using ten different random seeds. Therefore, the experimental results show the mean number of killed mutants together with the standard deviation.

6.2 Experimental Results

Table 6.1: Results for the Ceiling Speed Monitor. For each strategy, 93 test case were generated.

Mutation Type	Strategy	Mutation Score ECPT Tests			Mutation Score RT		
byte-order	STRAT-1_r	10.0 (0.0)/10	=	100.0 %	7.1 (2.1)/10	=	71.0 %
	STRAT-2_r	10.0 (0.0)/10	=	100.0 %	7.1 (2.1)/10	=	71.0 %
	STRAT-3_r	10.0 (0.0)/10	=	100.0 %	7.1 (2.1)/10	=	71.0 %
ordinary	STRAT-1_r	69.0 (0.0)/91	=	75.8 %	47.7 (8.1)/91	=	52.4 %
	STRAT-2_r	83.2 (1.3)/91	=	91.4 %	47.7 (8.1)/91	=	52.4 %
	STRAT-3_r	86.7 (0.8)/91	=	95.3 %	47.7 (8.1)/91	=	52.4 %
precision	STRAT-1_r	0.0 (0.0)/5	=	0.0 %	0.0 (0.0)/5	=	0.0 %
	STRAT-2_r	0.0 (0.0)/5	=	0.0 %	0.0 (0.0)/5	=	0.0 %
	STRAT-3_r	0.0 (0.0)/5	=	0.0 %	0.0 (0.0)/5	=	0.0 %
stuck-at	STRAT-1_r	46.0 (0.0)/50	=	92.0 %	30.8 (1.5)/50	=	61.6 %
	STRAT-2_r	46.1 (1.3)/50	=	92.2 %	30.8 (1.5)/50	=	61.6 %
	STRAT-3_r	46.7 (1.3)/50	=	93.4 %	30.8 (1.5)/50	=	61.6 %
switch-ports	STRAT-1_r	2.0 (0.0)/2	=	100.0 %	2.0 (0.0)/2	=	100.0 %
	STRAT-2_r	2.0 (0.0)/2	=	100.0 %	2.0 (0.0)/2	=	100.0 %
	STRAT-3_r	2.0 (0.0)/2	=	100.0 %	2.0 (0.0)/2	=	100.0 %
all	STRAT-1_r	127.0 (0.0)/158	=	80.4 %	87.6 (10.8)/158	=	55.4 %
	STRAT-2_r	141.3 (1.8)/158	=	89.4 %	87.6 (10.8)/158	=	55.4 %
	STRAT-3_r	145.4 (1.4)/158	=	92.0 %	87.6 (10.8)/158	=	55.4 %

Table 6.1 shows the mutation score for all test strategies compared to a random test suite of the same size. The mutation score is shown in column three and four, in the form $k(\sigma)/t = p\%$, where k indicates the mean number of mutants that were killed by applying the test strategy ten times with different random seeds. t indicates the total number of non-I/O-equivalent mutants, and p indicates the ratio of k/t in percent. σ denotes the standard deviation of k. The results are grouped for the different types of mutation operators and the last row shows the overall result consisting of all mutations.

At a first a glance, two observation can be made. First, it is obvious that the ECPT approach outperforms random testing for most mutation-operator types and never exhibits lesser test strength. Second, from all compared strategies STRAT-3_r received the best overall mutation score and is able to significantly increase the test strength of the ECPT approach when comparing the overall results of STRAT-1_r and STRAT-3_r. The superiority of STRAT-3_r is mainly caused by the high mutation score for the class of ordinary mutations. In particular, the replacements of relational operators–namely, the replacement of $>$ by $\geq$ and $<$ by $\leq$ and the reverse–can only be revealed by boundary-value tests.

The results for the class of 'ordinary' mutation operators shown here confirm the observations made in [HHP15] for a Java implementation of the Ceiling Speed Monitor. In contrast to the Java mutation experiments presented there, we used our randomisation heuristics STRAT-3_r in this work, which achieved a mutation score of 95.3% for ordinary mutations. We think that this score can be considered effective, so that a further increase of test cases (for example, by refining the input equivalence partitioning) is not necessary. It should be noted, however, that [HHP15, Table 2] shows that 100% fault coverage can be achieved for this case study and the ordinary mutation-operator class when refining the input partitioning and applying the ECPT approach with 610 test cases.

The performance of the ECPT tests is very high for mutations of the stuck-at fault model. All faults that were caused by wrongly connected signals (switch-ports mutation) and byte-order mutations were uncovered by ECPT tests, though it has to be mentioned that random tests were able to reveal all switch-ports mutations for this case study as well.

All compared strategies performed equally weakly for the precision mutations, though the low number of these mutations might weaken the validity of these results. The precision mutants had a structure similar to the following mutant shown in Listing 9.

Listing 9 Example of a Precision Mutation

```
float dV_sbi(float v) {
 if (v <= 110) {
  return 5.5f;
 } else if (v <= 210) {
  return 0.045f * v
    + 0.55f;
 } else {
  return 10.0f;
 }
}
```

```
float dV_sbi(float v) {
 if (v <= 110) {
  return 5.5f;
 } else if (v <= 210) {
  return 0.045 * v
    + 0.55f;
 } else {
  return 10.0f;
 }
}
```

The behaviour of the mutant (right-hand side) differs from the original implementation in the way that the original expression `0.045f * v + 0.55f` is calculated with single precision, while the expression in the mutant is calculated in double precision. The I/O-behaviour is only affected in cases in which the narrowed double-precision value of the overall expression differs from the

original result completely calculated in single precision. Unfortunately, this condition is not necessarily fulfilled when calculating arbitrary or even boundary values. The results of the precision mutations indicate that a more sophisticated approach is needed to reveal such subtle errors. Such an approach needs IEEE 754 conformant floating-point arithmetic. Since we use the SONOLAR SMT solver, our approach could be refined as part of future work. This would be beneficial for the concrete input calculation for SUTs, where precision errors are considered critical.

Table 6.2: Results for the Airbag Controller. For each strategy, 722 test case were generated.

Mutation Type	Strategy	Mutation Score ECPT Tests			Mutation Score RT		
byte-order	STRAT-1	3.0 (0.0)/3	=	100.0 %	2.0 (0.0)/3	=	66.7 %
	STRAT-2	3.0 (0.0)/3	=	100.0 %	2.0 (0.0)/3	=	66.7 %
	STRAT-3	3.0 (0.0)/3	=	100.0 %	2.0 (0.0)/3	=	66.7 %
ordinary	STRAT-1	60.0 (0.0)/67	=	89.6 %	34.4 (0.5)/67	=	51.3 %
	STRAT-2	64.0 (0.0)/67	=	95.5 %	34.4 (0.5)/67	=	51.3 %
	STRAT-3	66.5 (0.5)/67	=	99.3 %	34.4 (0.5)/67	=	51.3 %
precision	STRAT-1	0.0 (0.0)/2	=	0.0 %	0.0 (0.0)/2	=	0.0 %
	STRAT-2	0.0 (0.0)/2	=	0.0 %	0.0 (0.0)/2	=	0.0 %
	STRAT-3	1.0 (0.0)/2	=	50.0 %	0.0 (0.0)/2	=	0.0 %
sensitivity	STRAT-1	10.0 (0.0)/10	=	100.0 %	6.6 (0.5)/10	=	66.0 %
	STRAT-2	10.0 (0.0)/10	=	100.0 %	6.6 (0.5)/10	=	66.0 %
	STRAT-3	10.0 (0.0)/10	=	100.0 %	6.6 (0.5)/10	=	66.0 %
stuck-at	STRAT-1	36.0 (0.0)/59	=	61.0 %	42.9 (1.0)/59	=	72.7 %
	STRAT-2	56.0 (0.0)/59	=	94.9 %	42.9 (1.0)/59	=	72.7 %
	STRAT-3	58.1 (0.5)/59	=	98.5 %	42.9 (1.0)/59	=	72.7 %
switch-ports	STRAT-1	4.0 (0.0)/4	=	100.0 %	4.0 (0.0)/4	=	100.0 %
	STRAT-2	4.0 (0.0)/4	=	100.0 %	4.0 (0.0)/4	=	100.0 %
	STRAT-3	4.0 (0.0)/4	=	100.0 %	4.0 (0.0)/4	=	100.0 %
all	STRAT-1	113.0 (0.0)/145	=	77.9 %	89.9 (1.2)/145	=	62.0 %
	STRAT-2	137.0 (0.0)/145	=	94.5 %	89.9 (1.2)/145	=	62.0 %
	STRAT-3	142.6 (0.8)/145	=	98.3 %	89.9 (1.2)/145	=	62.0 %

The higher complexity of the airbag controller compared to the CSM increased the required test effort. 722 test cases were generated for each of the strategies–STRAT-1,-2,-3–and for the compared random-testing strategy. Note that the number of test cases differs from the results presented in [HHP17] because of the use of minimal state-identification sets and a minimal characterisation set for the Wp-method.

The experimental results for the airbag controller shown in Table 6.2 confirm the observations seen before. The ECPT testing approach has a test strength that is significantly higher than the test strength of naive random testing. Again test strategy STRAT-3 achieved the best mutation score caused by the superiority over STRAT-1,-2 for ordinary mutations but for stuck-at faults as well. The results for the airbag controller show that the mutation score for stuck-at faults largely depends on the number of different input vectors. STRAT-1 (61%) performed significantly worse

than STRAT-2 (94.9%). The selection of boundary values improved this mutation score to a total of (98.5%).

Compared to the CSM, the sensitivity-mutation operator was applicable to this case study. All sensitivity mutations were uncovered by the three compared ECPT strategies. In contrast to this, the mutation score of RT for this mutation operator indicates that random tests do not provide a sufficient level of confidence that errors due to missed signal changes and events are actually detected; therefore, a more sophisticated approach, like our ECPT approach, is needed for such errors.

Table 6.3: Results for the Route 2011 (example network). For each strategy, 477 test case were generated.

Mutation Type	Strategy	Mutation Score ECPT Tests			Mutation Score RT		
byte-order	STRAT-1	2.0 (0.0)/3	=	66.7 %	0.0 (0.0)/3	=	0.0 %
	STRAT-2	2.0 (0.0)/3	=	66.7 %	0.0 (0.0)/3	=	0.0 %
	STRAT-3	2.1 (0.3)/3	=	69.7 %	0.0 (0.0)/3	=	0.0 %
ordinary	STRAT-1	146.0 (0.0)/150	=	97.3 %	77.8 (9.0)/150	=	51.9 %
	STRAT-2	146.4 (1.1)/150	=	97.6 %	77.8 (9.0)/150	=	51.9 %
	STRAT-3	146.2 (0.8)/150	=	97.5 %	77.8 (9.0)/150	=	51.9 %
stuck-at	STRAT-1	148.1 (0.3)/173	=	85.6 %	163.0 (4.3)/173	=	94.2 %
	STRAT-2	150.9 (4.8)/173	=	87.2 %	163.0 (4.3)/173	=	94.2 %
	STRAT-3	154.4 (3.3)/173	=	89.2 %	163.0 (4.3)/173	=	94.2 %
switch-ports	STRAT-1	93.0 (0.0)/94	=	98.9 %	88.3 (1.2)/94	=	93.9 %
	STRAT-2	93.0 (0.0)/94	=	98.9 %	88.3 (1.2)/94	=	93.9 %
	STRAT-3	93.0 (0.0)/94	=	98.9 %	88.3 (1.2)/94	=	93.9 %
all	STRAT-1	389.1 (0.3)/420	=	92.6 %	329.1 (8.9)/420	=	78.4 %
	STRAT-2	392.3 (5.4)/420	=	93.4 %	329.1 (8.9)/420	=	78.4 %
	STRAT-3	395.6 (3.6)/420	=	94.2 %	329.1 (8.9)/420	=	78.4 %

Table 6.3 and Table 6.4 show the results of the interlocking case studies Route 2011 (example network) and Route 12a (Lyngby), respectively. The overall results confirm the superiority of the ECPT approach compared to RT. This is especially true for the Route 12a (Lyngby) case study. While the ECPT strategies obtained an overall mutation score of 88 to 90 percent, RT merely achieved a mutation score of 38 percent for the most complex route controller used in our experiments. For Route 2011 (example network), the same is true. The reader may find that RT achieved a much higher mutation score (around 78 percent) for Route 2011 (example network). This discrepancy of the overall RT mutation score of both interlocking case studies can mainly be explained by differences in the complexity of both case studies. We can observe that a lower complexity of the model results in a higher coverage of the overall system behaviour by RT. For the complex case study, Route 12a (Lyngby), RT input sequences were very likely not to cover parts of the system behaviour that can only be exercised after longer sequences of dedicated inputs. It is, for example, very unlikely that a random input sequence will exercise the whole sequence of route allocation, locking, occupancy and sequential release. Therefore, the low mutation scores of RT for the complex Route 12a (Lyngby) were to be expected. Thus, we

Table 6.4: Results for the Route 12a (Lyngby). For each strategy, 2222 test case were generated.

Mutation Type	Strategy	Mutation Score ECPT Tests			Mutation Score RT		
byte-order	STRAT-1	5.0 (0.0)/5	=	100.0 %	3.0 (0.0)/5	=	60.0 %
	STRAT-2	5.0 (0.0)/5	=	100.0 %	3.0 (0.0)/5	=	60.0 %
	STRAT-3	5.0 (0.0)/5	=	100.0 %	3.0 (0.0)/5	=	60.0 %
ordinary	STRAT-1	220.0 (0.0)/224	=	98.2 %	49.6 (2.8)/224	=	22.1 %
	STRAT-2	213.6 (7.6)/224	=	95.4 %	49.6 (2.8)/224	=	22.1 %
	STRAT-3	216.7 (6.4)/224	=	96.7 %	49.6 (2.8)/224	=	22.1 %
stuck-at	STRAT-1	281.0 (0.0)/421	=	66.7 %	110.0 (0.0)/421	=	26.1 %
	STRAT-2	286.7 (4.8)/421	=	68.1 %	110.0 (0.0)/421	=	26.1 %
	STRAT-3	297.9 (8.3)/421	=	70.8 %	110.0 (0.0)/421	=	26.1 %
switch-ports	STRAT-1	665.0 (0.0)/675	=	98.5 %	345.0 (0.0)/675	=	51.1 %
	STRAT-2	670.0 (1.0)/675	=	99.3 %	345.0 (0.0)/675	=	51.1 %
	STRAT-3	671.9 (1.4)/675	=	99.5 %	345.0 (0.0)/675	=	51.1 %
all	STRAT-1	1171.0 (0.0)/1325	=	88.4 %	507.6 (2.8)/1325	=	38.3 %
	STRAT-2	1175.3 (8.2)/1325	=	88.7 %	507.6 (2.8)/1325	=	38.3 %
	STRAT-3	1191.5 (13.4)/1325	=	89.9 %	507.6 (2.8)/1325	=	38.3 %

conclude that the superiority of our ECPT (at least over RT) will be even more apparent for SUTs of higher complexity.

The results of the ECPT approach confirm that ECPT is indeed suitable to reveal almost every ordinary mutation and nearly all switch-ports mutations. Considering the ordinary mutation operators, for both case studies, all ECPT strategies achieved mutation scores > 95 percent. Surprisingly, for Route 12a (Lyngby), STRAT-2 and STRAT-3 obtained lower mutation scores than STRAT-1 on average. We assume that this is caused by the fact that, for this case study, the negative effect of randomisation was reflected in the experimental results. As explained in Section 4.3.2, the random selection of concrete inputs from IECs may lead to situations in which the completeness property with respect to the fault model, as introduced in Section 2.5.3, is not preserved. However, this negative effect could be encountered by first running the test suite with a fixed representative and afterwards applying a randomised test suite according to STRAT-2 or STRAT-3. We repeated our experiments using this strategy and observed that a combination of the test suite with fixed IEC representatives with STRAT-2 and STRAT-3 achieved a mutation score of 100 and 98.7 percent, respectively, for the ordinary mutation operators. However, in this case, the number of test cases was increased to 4444 test cases. For SUTs of high criticality, this approach might be favourable, while in other cases the testing effort might be critical and justify the use of the randomised strategies STRAT-2 or STRAT-3 only.

For both route-controller case studies, ECPT could demonstrate its suitability to reveal switch-ports mutations. For the Route 12a (Lyngby) case study, the ECPT approach was able to reveal > 98 percent of the 675 switch-ports mutations. This result is remarkable since the former case studies and the respective SystemC mutations resulted in a low number of switch-ports mutations due to the lower number of system inputs and outputs. For Route 12a (Lyngby), it could be demonstrated that in case of a high number of input variables–which increases the risk of integration errors due to erroneous connections or misconfigurations that are mimicked by the

switch-ports mutation–our ECPT offers a reasonably high level of test strength. Comparing this result to the mutation score of around 51 percent achieved by RT, the results indicate that erroneous communications and interconnections are difficult to discover simply by stimulating the system inputs by random values whenever the complexity and number of system variables becomes high enough. We conclude that the high test strength of ECPT for switch-ports mutations is not caused by a combination of chance and the large number of test cases but can be considered a substantial strength of our ECPT approach.

Considering the stuck-at mutations, the experimental results highlight two facts worth mentioning. First, RT shows a surprisingly high mutation score of 94 percent for Route 2011 (example network) and a very low mutation score of 26 percent for Route 12a (Lyngby). Again, this discrepancy is to be expected because of the differing complexities of both case studies. Second, the mutation scores for stuck-at mutations are not as satisfying as the mutation scores observed for the other case studies so far. For Route 12a (Lyngby), the ECPT strategies yield mutation scores of 67 to 71 percent. Many of the mutations were not revealed by the test suite because of some "redundancy" in the input variables. Consider, for example, a stuck-at mutation in one of the input variables representing the route state of a conflicting route. This mutation may result in the input variable never evaluating to ALLOCATING or LOCKED from the perspective of the route controller. In this case, the error can only be observed if this exact input variable is the only variable set to ALLOCATING or LOCKED in a test step. In the IECs calculated by our ECPT approach, however, there is no distinction between input vectors in which all conflicting routes or only one conflicting route is set to ALLOCATING or LOCKED. The same argument applies for other input variables, like the state of flank-protection elements. We expect these mutations to be revealed anyway during HSI testing of the complete interlocking system in which every route controller is tested in isolation using the ECPT approach. Most of the uncovered stuck-at mutations are related to internal system-state variables like the conflicting route-status variables. The state of a conflicting route is written by the route controller of this conflicting route. If a functional error of this route controller leads to a situation in which a single bit in this state variable is constantly stuck at zero or one, the respective ECPT HSI tests of this route controller will definitely reveal such an error, since simple output faults are captured by the fault model of our approach.

6.3 Deterministic Finite-State Machine Experiments for Additional States Mutations

To investigate the test strength of the heuristics for exploration of additional states, we performed dedicated experiments using DFSM mutations. The use of SystemC mutations was insufficient for this, because the first-order mutants rarely resulted in additional states in an implementation. The use of DFSM mutations makes it possible to introduce additional states directly to the DFSM abstraction. The resulting mutants will be outside the fault domain because of the additional states,[1] while the IECP of the mutants will be equal to the IECP used for testing. Therefore, the results presented in the remainder of this study clearly focus on the evaluation of the heuristics presented in Section 4.3.4.

[1] Recall that in the previous experiments we applied the Wp-method under the assumption that the number of states in the implementation was equal to the number of states in the specification ($m = n$). Thus, every DFSM mutation with at least one additional state is outside the fault domain.

The mutants were created as follows. In a first step, additional equivalent states were added to the DFSM by duplication of existing states and changing the target of existing transitions in some cases from the original state to the equivalent duplicated state. Afterwards, a single DFSM fault (i.e., a transfer or an output fault) was seeded. The resulting DFSM was checked to be non-I/O-equivalent and to contain more states in its minimised DFSM than the original DFSM.

For our experiments, 2000 mutants were generated in each case with up to 10 additional states. The concrete number of additional states generated by the mutation algorithm varied, but our observations indicate that the actual number of additional states was distributed quite uniformly. For the test-case generation, we used the three heuristics presented in Section 4.3.4, with the assumption that the SUT does at most contain 10 additional states.

In the following tables, SUF-R denotes the heuristics performing a random walk of length 10 through the DFSM. SUF-S denotes the heuristics that guarantees that every state is visited equally often. SUF-E denotes the heuristics that ensures that every edge is taken equally often. Again, we generated a random test suite to compare the ECPT test suites to a test suite with the same number of test cases with same length as the test cases generated by ECPT–including the suffix of length 10 for STRAT-R, STRAT-S and STRAT-E.

Table 6.5: Results for Random Suffix Heuristics for the Ceiling Speed Monitor Considering DFSM Mutants with Additional States. For each strategy, 93 test case were generated.

Strategy	Mutation Score ECPT Tests			Mutation Score RT		
STRAT-1_r	463.0 (0.0)/2000	=	23.1 %	4.6 (7.9)/2000	=	0.2 %
SUF-R	1361.2 (46.1)/2000	=	68.1 %	63.7 (86.2)/2000	=	3.2 %
SUF-S	1359.4 (97.7)/2000	=	68.0 %	63.7 (86.2)/2000	=	3.2 %
SUF-E	1439.5 (76.4)/2000	=	72.0 %	63.7 (86.2)/2000	=	3.2 %

Table 6.5 shows that, for the CSM case study, about 70 percent out of the 2000 mutants could be detected using the suffix heuristics. Without the use of the suffix heuristics, merely 23 percent of the mutants could be revealed, which is not surprising, given the fact that the mutants were outside the fault domain because of their additional states. The improvement to a mutation score of up to 72 percent by use of STRAT-E is remarkable given the fact that the number of tests was not increased and the search for the mutation was randomised and only guided in a way ensuring that the original DFSM was uniformly covered.

Comparing the three heuristics, SUF-E shows the best results–discovering up to 80 mutants more than the other heuristics. SUF-R and SUF-S show quite similar results. The comparison to RT points out that the naive choice of arbitrary input values to generate test sequences was not sufficient to reveal errors resulting from additional states in the implementation. RT merely achieved a mutation score of 3.2 percent when applying random test sequences of the same length as our ECPT strategies with suffix heuristics. This very low mutation score results from the fact that a random choice of input values from the input domain using a uniform distribution of values is very likely to select values from X_1 and X_5 (cf. Figure 3.3) while the other IECs are very unlikely to be selected because of their narrow shape.

Table 6.6 confirms the observation that SUF-E performs best for the airbag controller. In this case, about 98 percent of the 2000 mutants were rejected by the test suite using strategy SUF-E.

Table 6.6: Results for Random Suffix Heuristics for the Airbag Controller Considering DFSM Mutants with Additional States. For each strategy, 722 test case were generated.

Strategy	Mutation Score ECPT Tests			Mutation Score RT		
STRAT-1	1666.0 (0.0)/2000	=	83.3 %	233.1 (32.8)/2000	=	11.7 %
SUF-R	1942.8 (11.1)/2000	=	97.1 %	305.0 (63.2)/2000	=	15.2 %
SUF-S	1962.0 (10.5)/2000	=	98.1 %	305.0 (63.2)/2000	=	15.2 %
SUF-E	1966.6 (11.2)/2000	=	98.3 %	305.0 (63.2)/2000	=	15.2 %

Note that the value of STRAT-1 indicates that more than 83 percent of the mutants were killed without any suffix heuristics, although these mutants were not covered by the fault domain. Therefore, the improvement of the suffix heuristics is not as obvious as for the CSM. Still, the final mutation score is remarkable.

Comparing the results with RT, it is apparent that RT, with a mutation score of about 15 percent, is inadequate to deal with the additional-states mutants from these experiments. Compared to the CSM case study, the RT mutation score did not suffer from the large inequality of IEC sizes; therefore, RT yielded slightly better, but still unacceptably low, mutation scores.

Table 6.7: Results for Random Suffix Heuristics for the Route 2011 (example network) Considering DFSM Mutants with Additional States. For each strategy, 477 test case were generated.

Strategy	Mutation Score ECPT Tests			Mutation Score RT		
STRAT-1	869.0 (0.0)/2000	=	43.5 %	573.5 (12.0)/2000	=	28.7 %
SUF-R	1617.2 (24.7)/2000	=	80.9 %	665.7 (16.0)/2000	=	33.3 %
SUF-S	1683.4 (16.6)/2000	=	84.2 %	665.7 (16.0)/2000	=	33.3 %
SUF-E	1690.1 (18.5)/2000	=	84.5 %	665.7 (16.0)/2000	=	33.3 %

Table 6.7 and Table 6.8 show the results for the interlocking case studies Route 2011 (example network) and Route 12a (Lyngby), respectively. Again strategy SUF-E performed best and revealed on average 85 percent of the mutations. SUF-S was in both case a little less effective, and SUF-R showed the worst results of the three compared suffix strategies. Still, all suffix strategies were able to double the mutation score for the additional-states mutations considered in these experiments when compared to STRAT-1.

The surprisingly high mutations score that RT achieved for the Route 2011 (example network) model can be explained by appealing to the relatively large number of test cases applied to a DFSM with merely eight states.

The high variance in the mutation scores is probably correlated with the difference in DFSM characteristics. In particular, the number of states of the minimised DFSM abstraction seems to be correlated with the mutation scores achieved by the ECPT heuristics. Observe that the

Table 6.8: Results for Random Suffix Heuristics for the Route Controller for Route 12a (Lyngby) Considering DFSM Mutants with Additional States. For each strategy, 2222 test case were generated.

Strategy	Mutation Score ECPT Tests			Mutation Score RT		
STRAT-1	822.0 (0.0)/2000	=	41.1 %	16.5 (2.1)/2000	=	0.8 %
SUF-R	1595.4 (61.5)/2000	=	79.8 %	19.0 (2.0)/2000	=	0.9 %
SUF-S	1684.3 (10.7)/2000	=	84.2 %	19.0 (2.0)/2000	=	0.9 %
SUF-E	1716.5 (22.3)/2000	=	85.8 %	19.0 (2.0)/2000	=	0.9 %

number of states for CSM, Route 2011 (example network), Route 12a (Lyngby) and the airbag controller are 4, 8, 17 and 44, respectively. The mutation scores achieved by ECPT in combination with suffix heuristics yields the same order. For CSM, the lowest mutation score is achieved, while we observe the highest mutation score for the Route 12a (Lyngby) model. Note that the seeding of 10 additional states has a higher impact for models with lower numbers of states. The recurrence diameter of the resulting mutant is increased in any case by a value in the range of one and ten. Obviously, the relative increase is higher for smaller models, thereby resulting in harder-to-detect mutations.

From these experiments, it can be summarised that SUF-E seems to be the best option when suffixes shall be generated to explore additional states in an SUT. From the experimental results, we conclude that such an approach is recommendable for test suites in which additional test-case length does not excessively introduce extra testing effort. As argued before, for system tests, the number of test cases is usually the critical factor, as a system reset is time costly. Additional test steps are usually not as expensive and may be acceptable. The suffix heuristics were able to increase the mutation scores in the presence of additional-states mutations by factors of up to three in our experiments–without a need for a larger number of test cases.

One could argue that the mutation scores are in most cases far from 90 percent–except for the airbag controller. As mentioned earlier, for application to safety-critical systems, this value may be insufficient if our ECPT approach is applied without complementary verification measures. However, we again emphasize that the results are nonetheless remarkable given that the approach is not guided by any implementation details. No information regarding the implementation internals is used, and a more sophisticated approach will probably require internal information to better guide the test generation. In practice, testing activities will be complemented by code-coverage monitoring, which will give some insight into the parts that are not sufficiently covered by ECPT tests. The suffix heuristics may help to cover some previously uncovered parts of the implementation. However, such a best-effort approach can in the best case reduce some costs for manual test-case specification. Some parts of the system will still need to be tested by more sophisticated approaches and manually defined test cases considering expert knowledge of the implementation. Furthermore, safety-related standards like [Eur01] require long-duration tests that are specifically dedicated to the exploration of system states that are to be expected after long test sequences. Thus, the ECPT test can be expected to be complemented by other testing activities to raise the confidence in the absence of harmful additional states in the SUT.

6.4 Threats to Validity

6.4.1 Dependence on Syntactic Model Representation

Usually, MBT and the test strength of the generated test cases can be expected to be dependant on the concrete syntactical representation of the test model. Fortunately, this threat to validity is encountered by our ECPT approach. As mentioned in Section 2.5.4, the ECPT is independent of concrete model representations. The RT approach used for comparison in our experiments is independent of the concrete model representation as well. In case of RT, we use the test model as an oracle only: I.e., we apply the randomly generated input traces to the test model to obtain the expected result, which contains only expectations for output variables. Thus, internal model variables have no impact on the generated test suites and test oracles.

6.4.2 Threats Caused by Model Selection

The selection of models might have an impact on the experimental results. To reduce this threat, we used different models with opposing characteristics. The CSM has a very small recurrence diameter, a small number of internal states, and relatively wide input equivalence classes. The airbag controller, on the other hand, has many internal states, a large recurrence diameter, and narrow input equivalence classes. While both models have input classes that are unions of continuous subsets of $\mathbb{R}^n$, the case studies of the route controllers–Route 12a (Lyngby) and Route 2011 (example network)–work with input classes which are subsets of (very large) discrete sets specified by complex predicates that represent train and track element constellations in a railway network. For these classes, interior values and boundary values have to be defined in a different way than described in Section 4.3.3. Despite these considerable differences, the results obtained with test strategy STRAT-3 are on the same level for all compared case studies. The test model of Route 12a (Lyngby) further illustrates that very large state spaces can be dealt with and that our approach is scalable to systems of high complexity. The mutation experiments conducted for the case studies all yielded comparable results. This raises confidence that the results are generalisable to other models as well. Therefore, we conclude that the residual threat related to model selection is very low.

6.4.3 Threats Caused by the Mutant Generator

Threats to validity might be caused by the mutant generator that we used in our experiments. In our previous work [HHP15], we used three different Java mutant-generation tools. The results from the μJava tool were presented. Apart from that, the PITest[2] and the Major mutation framework [Jus14] were applied. All these tools used similar mutation operators. So far, in all of our experiments, the observed impact of the choice of a specific mutant generation tool was quite low. Based on this, we implemented the "ordinary" mutation operators mentioned above. The results of all three Java-mutant generators were comparable to the results of our SystemC-mutant generator using the subset of ordinary mutation operators.

[2] See `http://pitest.org/`

6.4.4 Dependencies on the DFSM Test Strategies Applied

The comparison of results from previous work also indicates that the test strength of STRAT-1,-2,-3 is robust with respect to the selection of complete DFSM testing strategies: While we used the Wp-Method in this work, in previous work we used both the W-method and the Wp-method. In [PHH16b], we used both methods on the same interlocking case studies. Both methods obtained very similar results in experiments in which Java mutations were used. The Wp-Method [FvBK+91, LvBP94] usually results in fewer test cases than the W-Method, while guaranteeing the same fault coverage at the DFSM level. Fewer test cases, however, also implies that fewer random selections from input equivalence classes will be performed when applying strategies STRAT-2 and STRAT-3. The experiments from [PHH16b] indicate that this does not affect the test strength for SUT behaviours outside the fault domain. Therefore, we expect that our results do not significantly depend on the specific DFSM testing approach that is applied.

6.4.5 Runtime Error Detection

In preliminary stages of our experiments, we also tried out some types of typical runtime-error mutations. Typical runtime errors include stack-overflow errors, memory-access violations caused by invalid pointer addresses or invalid array indexes and race conditions. The results showed that our approach is not well-suited for the detection of runtime errors. It is a well-known fact that runtime errors are hard to detect via functional testing [KF16].

To overcome this problem, a variety of tools exists to detect typical runtime errors without dynamic execution of code. These tools use static analysis methods and abstract interpretation to statically determine the dynamic behaviour of software [CCF+06, CCF+09]. Nonetheless, our testing strategy may, in some situations, reveal runtime errors during execution which may be based on the fact that our approach results in high code coverage, which is correlated with detection probabilities in general. But the sporadic nature of runtime errors—especially errors related to memory-access violations and memory corruption—usually results in indeterministic behaviour during test execution.

6.4.6 Significance of SystemC Mutations as Surrogates for Real HW/SW Integration Faults

While evidence exists that software mutations provide a good way to evaluate the test strength of a testing strategy [ABL05, JJI+14], there is no empirical evidence that the SystemC mutation approach presented in this work is coupled with real HSI faults. The reason is obviously that we were, to the best of our knowledge, the first to consider such an approach for the evaluation of HSI tests. Future experiments following an experimental setup comparable to [ABL05, JJI+14] will have to investigate the representativeness of our proposed mutations. Therefore, a threat to validity considering the applicability of our testing strategy to HSI testing remains. This threat should be a starting point for future work.

6.4.7 Threats Concerning the Deterministic Finite-State Machine Mutations

The use of DFSM mutations in the experiments above was due to the need for mutants containing additional states. These may not be representative of real erroneous SUTs with additional states. However, the way the mutants were generated—by first introducing equivalent states and then seeding a single fault—might find its analogy in real development lifecycles. Use of a software clone–i.e., a piece of code that is duplicated as an exact or very similar copy–is considered a bad practice in software development. This is due to the observation that clones increase the maintenance effort of software and increase the risk of errors. A change in one instance of the clone but not the other instance of the clone will result in inconsistencies that are in many cases the cause of error. Thus, software clones in an implementation and errors emerging from changes in one instance of the clone might result in erroneous SUTs that look similar to the mutants considered in these experiments.

7 Related Work

This chapter provides an overview of related work. We give some references to work that is related to automated test-case generation, including MBT and equivalence class partition testing. For an extensive overview, please also refer to [ABC+13] and the references therein. Finally, we refer to work related to the domain of mutation analysis for test-strength evaluation.

7.1 Model-based Testing

The approaches presented in [HLSU02, HdMR04, EKRV06] show how model checking techniques can be used to generate test suites on extended finite-state machines (EFSMs). [HLSU02] encodes different criteria, such as state and transition coverage, but also includes advanced data-flow criteria like define-use-pairs as CTL formulas. These CTL formulas are then checked on the model and the counter-example that is returned by the model checker represents a test case that is needed to fulfil a certain coverage criterion. Similarly, [HdMR04] uses an incremental approach that generates test cases to fulfil a set of goals. These goals are encoded as model-checking instances. The set of test cases is iteratively augmented, and test cases are extended to fulfil all test goals. [EKRV06] additionally tries to optimise the test cases with respect to test-case length. This is done by guiding the search of the UPPAAL model checker. However, all these approaches require the EFSM state space to be finite. Note that EFSMs can be straightforwardly expressed by RIOSTSs. Therefore, our approach is applicable to EFSMs with inputs from possibly infinite domains.

Most UML-based MBT approaches try to fulfil structural-coverage criteria, like state and transition coverage of state machines [FHP02]. For these approaches, the generated test suite largely depends on the syntactical representation of the test model. In contrast to this, our approach is independent of the syntactical representation of the reference model. The RT-Tester MBT component (RTT-MBT) [PVL11, Pel13] that our implementation relies on is an example for an industrial-strength tool making it possible to generate test cases from UML/SysML state machines. The tool makes it possible to generate test cases covering different goals, including goals that contribute to the following coverage criteria: state coverage, transition coverage, hierarchical transition coverage, basic control-states pair coverage and MC/DC coverage [Pel13]. Additionally, user-defined goals can be specified using linear temporal logic (LTL) specifications. The implementation presented in this work reuses several parts of RTT-MBT and augments RTT-MBT by the novel ECPT approach.

[FG09] evaluates different MBT test-case generation techniques. The authors define different existing coverage criteria, including different structural, control and data-flow criteria and mutation coverage. Again, these coverage criteria are formulated as CTL model-checking instances. The main result of the evaluation is that no coverage criterion can be considered superior to another. All criteria "cross-cover" other criteria but no criterion completely subsumes all other criteria. The authors conclude that a combination of different criteria in combination with a

flexible framework, as given by the use of a model checker, is supposed to achieve the best results. Taking these results into account, future work could address the combination of our approach with other test criteria. These could be used for IEC refinement or for more sophisticated heuristics to be used for concrete input calculation from IECs.

Other MBT approaches [dVT00, KATP02, Tre96b, Bel10] are based on LTSs. These approaches use the ioco-conformance relation [Tre96a]. LTSs are an event-based formalism which, in its original form, relies on a finite event alphabet. To model concrete data from large or even infinite domains, Symbolic transition systems (SyTSs) can be used. In [FTW04], the ioco-conformance relation is lifted to SyTSs. [FTW04] describes an algorithm to derive a test suite that is sound and complete for a given set of system traces. However, the generated test suite is not necessarily finite and no equivalence partitioning of system variables is considered. Therefore, the concrete value selection for input variables from an infinite input domain results in an infinite number of test cases.

An MBT approach that comes close to the approach presented in this work is presented in [DBI12]. Stream X-machines (SXMs) are used as a modelling formalism. SXMs are FSMs enriched by a possibly infinite memory and transitions that are labelled by *processing functions*. [DBI12] uses a modified version of the W-method [Cho78] to generate a complete test suite. The main assumption used by the authors of [DBI12] is that the processing functions are correctly implemented. In contrast to this, our approach guarantees completeness for all systems that are part of the fault domain, as defined in Section 2.5.3. This is a less restrictive assumption compared to that in [DBI12].

[Gau95] investigates the theory of complete (respectively exhaustive) testing theories. Restricting an exhaustive test set to a finite subset can be regarded as the introduction of hypotheses called *selection hypotheses* in [Gau95]. For example, the *uniformity hypothesis* relates to equivalence class partition testing which states that the SUT shows equivalent behaviour for all members of an equivalence class. The notion of "completeness with respect to a fault model" as used throughout this work was first introduced in [PYvB96].

For DFSMs, different complete testing theories exist. This work uses the W-method originally proposed in [Cho78]. The Wp-method [FvBK+91] modifies the W-method to produce smaller test suites. Other examples of complete testing theories are the D-method (distinguishing sequence) [Gon70], T-method (transition tour) [NT81], UIO-method (unique input output) [SD85], and modified versions of the UIO method: i.e., the SUIO-method (single UIO) [ADLU91] and the MUIO-method (multiple UIO) [SLD92]. A generalised version of the Wp-method [LvBP94] is applicable to non-deterministic FSMs as well. Other approaches that are applicable to non-deterministic FSMs are presented in [PY14, Hie04].

Another possible way to generate test cases is to intentionally introduce faults into the SUT and then derive test cases that are able to reveal this specific fault, as in [KST+15] where UML models are mutated. Section 7.4 gives some more examples of work related to mutation testing.

The MBT approach treated in this work is based on a wide range of previous work mainly conducted at the Department of Operating Systems at the University of Bremen [PVL11, Pel13]. This work led to the development of RTT-MBT, which has been the baseline for the development of our implementation of the ECPT approach. ECPT testing theory has been developed in [HP13], and the completeness of the approach has been proven therein. In [HP16a], more details including an implementation proposal have been laid out. The approach has been generalised for the application to non-deterministic systems as well in [HP16b]. However, our implementation

does not yet consider the results of [HP16b]. The fact that the ECPT is independent of the syntactical representation is discussed in detail in [PH16].

Another aspect of MBT to be considered is the feature of incremental testing. While an SUT will usually be implemented incrementally, it is advantageous if the testing approach supports incremental testing. Incremental test approaches shall make it possible to focus on the changes introduced by some deltas—like addition and modification of transitions in FSM-based test models. In turn, the test generation can be accelerated, and in the best case, test execution can be limited to the changes that were introduced. In [VBM15], in the context of product-line testing (where deltas are introduced by configuration of products from a common core model of the product line), incremental testing has been applied to test models obtained from the DeltaJava framework. The authors propose an incremental test-case generation approach, and it is shown that the test-case generations can be accelerated by a factor of two using this delta-oriented approach. The complexity considerations given in [VBM15] indicate that this result can be generalised to other applications. Therefore, the consideration of incremental FSM-based testing may be worth pursuing as a way for our ECPT approach to speed up test-case generation and limit the test-case execution to test cases that are needed to verify the changes introduced between different software versions.

7.2 Equivalence Class Partition and Boundary-value Testing

Testing based on equivalence classes, often referred as *partition testing*, has long been known as a worthwhile approach. Therefore, safety-related standards [Eur01, RTC92, ECS09] mandate for the use of equivalence-class testing. Early work on the formalisation of testing and equivalence-class testing [GG75, WO80, RC81] focused on stateless specifications and implementations (e.g., the implementation and specification of a single function). [WO80] introduced the concept of *revealing subdomains*. A *revealing subdomain* is a subset of the input domain with the following property: If one input of this domain reveals an error, all other inputs of this subdomain reveal an error as well. Note the correspondance to our definition of a fault domain given in Section 2.5.3–especially Equation 2.78. Given an input partitioning of revealing subdomains, a test suite derived by selection of one representative from each subdomain is complete in our words or *ideal*–i.e., *valid* and *reliable*–in the words of [GG75]. Because the proof that a subdomain is revealing is infeasible in general, [WO80] and [RC81] propose an approach that combines implementation (white-box) information and specification (black-box) information. Symbolic execution techniques can be used to calculate an input partition of the implementation (the *path partition*) and an input partition according to the specification (the *problem partition* [WO80] or *specification domain* [RC81]). The intersection of both partitions can then be used for test-data generation.

[DF93] addresses automated partition analysis for state-based models. VDM specifications are used for partition analysis. The approach requires the calculation of a DNF representation of the specification. From this DNF, an FSM is calculated and this FSM is used in turn to generate test sequences. In our work, we intentionally prevented the use of a DNF representation of the transition relation, because, in general, this DNF will grow exponentially for more complex specifications. Instead, we use our optimisation as presented in Section 4.3.5.1.

Furthermore, state-equivalence class calculation for reactive systems is addressed in [GGSV02]. The authors describe how systems defined in a formalism called abstract state machines can be abstracted to an FSM. The approach presented in [GGSV02] is applicable to systems with

finite inputs but with potentially infinite state space. The algorithm depends on a user-defined equivalence relation and a relevance condition: i.e., a condition to prune state-space exploration. The state space is explored and state-equivalence classes, called *hyperstates* by the authors, are calculated. In general, the choice of the equivalence relation and relevance condition may result in an under-approximation of the final FSM: I.e., not all reachable state equivalence classes may be found and hence the FSM may be incomplete. The approach presented there is related to our reachability analysis presented in Section 4.3.5.1. In contrast to [GGSV02], we are able to deal with infinite input domains by use of our IEC abstraction. The use of the ∼MO equivalence for abstraction in combination with the restriction of model and output variable domains to finite sets results in an FSM abstraction that is guaranteed to be complete and finite in our special case.

In the aforementioned work [RC81], the authors suggest the use of boundary values from IECs to be used to reveal so-called domain errors. Evidence for the fact that boundary-value testing yields higher fault coverage than equivalence-partition testing in isolation is given in [Rei97]. A formal definition of boundary coverage criteria has been introduced in [KLPU04]. [Rei97] again focuses on stateless specifications. In contrast to this, [KLPU04] addresses the sequencing problem, which, if we understand it correctly, solves the problem of finding a sequence from the initial state to a state in which a boundary value is applicable. However, the approach described in [KLPU04] uses an equivalence-class definition that considers only the local effect predicates—i.e., the pre and post conditions of functions—while our definition of IECs always considers the effect of inputs on every possible state. Furthermore, in our work, we merely use the *One-Boundary* criterion to select a single boundary value. We do not use one of the stricter boundary-coverage criteria (such as *All-Edges*) presented in [KLPU04], although the effectiveness of these criteria integrated with our test-case generation could be investigated in future work. However, at the moment of this writing, we consider a value a boundary value if it lies on at least one "edge" of an IEC. However, in combination with the randomisation, we use multiple boundary values. We intentionally do not refine the IECP to fulfil a strict boundary coverage criterion, because this would result in an exponential growth of the IECP. Our heuristics, denoted as STRAT-3 in our experiments, showed good results while not affecting the size of the generated test suites. Still, this heuristic approach preserves the completeness of our approach in contrast to that of [Rei97] and [KLPU04], which do not give any statement about guaranteed fault coverage, as defined by a formal fault model.

7.3 Adaptive Random Testing

Adaptive random testing (ART) [CKMT10], in contrast to RT, focuses on optimised distributions of test data over the input domain. This distribution is favoured based on the assumption that failures–respectively inputs that provoke failures–form contiguous regions. The distance of new test inputs to existing inputs that have not yet raised a failure should thus be maximised. Therefore, randomly selected test data should be evenly spread over the input domain. Again, most research in the domain of ART focuses on state-less tests in which input vectors are generated in contrast to sequences of input vectors that are needed for the testing of reactive systems. However, in [AIB10], a combination of ART and search-based testing is applied to real-time embedded systems.

[CTY01] presents surprising results of comparing random testing and partition testing. Note that partition testing there means the general approach to the partitioning of the input domain

by arbitrarily pre-defined criteria. The authors show that equivalence partition testing does not necessarily always perform better than RT. The test strength of partition testing strongly depends on the failure patterns: i.e., on the location of inputs that are able to provoke failures. If these failure inputs are uniformly spread over all IECs then partition testing behaves worse than random testing. The authors propose a technique called *proportional sampling* to ensure that partition testing behaves at least as well as RT. Proportional sampling extends partition testing by restricting partition testing in a way which assures that the number of inputs from each IEC used in testing is proportional to the size of the IEC. This is called the proportional sampling condition. It guarantees that partition testing is at least as good as random testing regardless of the failure patterns. If the failure pattern has a form—in which many failures are in the same partition—then partition testing is the right method; otherwise, proportional sampling is able to improve partition testing. The interval bisection introduced in Section 4.3.1.1 ensures that the proportional sampling condition will finally be ensured. Given a refined IECP obtained by interval bisection, it is ensured by construction that the number of refined IECs is proportional to the size of the original IECs. It must be noted, however, that proportional sampling is able to improve the test strength in case of poor choices of equivalence classes only. These are cases in which RT performs better than partition testing. We believe that our ECPT approach will most likely result in partitionings that are "good enough" to surpass RT. Our experimental results support this hypothesis. However, in the unlikely case in which our ECPT approach performs worse than RT, interval bisection can be used to ensure that at least the test strength of RT is achieved. As argued before, this will result in an enlarged fault domain as well.

7.4 Mutation Analysis and Mutation Testing

The first mutation-analysis experiments were preformed in the late seventies [DLS78]. [JH11] gives an overview of work that evolved from these early experiments. While most of this work is related to mutation analysis, [ABM98] proposes the use of mutation testing: i.e., the concept to drive the test-case generation by mutations.

When using mutation analysis for experimental test-strength evaluation or mutation testing for the automated generation of test cases, one relies on the representativeness of the generated mutants. The central question is whether the mutants that are used are a valid surrogate for real faults. Some experimental evidence exists to show that this is the case [ABL05, JJI+14]. [ABL05] investigates whether there are statistically significant differences in failure detection ratios of different test suites randomly drawn from large test pools when using real faults and when using randomly seeded mutations. The authors used eight subject programs, written in C. For these programs, large pools of test cases and faults are available. The experiments show that mutation scores obtained using SW mutation operators seem to be a good indication of the real fault-detection capability of a test suite. The results suggest that mutations are neither easier nor significantly harder to detect than real faults. In their statistical experiments, the authors of [JJI+14] use a similar approach. Additionally, the effect of code coverage on fault-detection capabilities is considered in the experiments. Code coverage may have an impact on both mutation score and real fault-detection capability. Therefore, the authors control this variable in their experiments. The results indicate that there is a positive correlation between mutation score and real fault-detection capabilities and that this correlation is significantly stronger than the correlation between statement coverage and fault-detection capabilities.

7.4.1 Model-Based Mutation Testing

A model-based approach for mutation testing is presented in [KST+15]. The tool MoMuT::UML supports the mutation based test-case generation from UML models that are composed of class diagrams, state machines and instance diagrams. The tool uses strong mutation testing: I.e., the equivalence of a mutant to the original model is checked using the ioco-conformance relation of LTSs. For non-equivalent mutants, a trace can be computed that reveals the ioco non-conformance. Because of the use of ioco and LTS semantics, this approach is limited to finite systems.

Another model-based mutation-testing approach is proposed by [PM11, PM12]. This approach follows the weak mutation-testing approach: I.e., a mutant is considered non-equivalent as soon as the internal state differs from the original model (in contrast to strong mutation testing in which the difference must be observable). Therefore, this approach is applicable as a white-box testing strategy in which the internal state can be observed.

[HM09] describes a way to generate mutation-based test cases for probabilistic and stochastic FSMs: i.e., FSMs whose transitions contain probabilities or stochastic times. The generated test sequences are applied several times and the output is investigated using statistical methods to decide whether the observations match the expected frequency of observations.

Mutation-testing approaches allow for a mutation score of 100 percent given enough computational effort. This value can hardly be achieved by other approaches. While mutation testing intentionally tailors the test cases to achieve 100 percent mutation score, our approach is completely independent of any mutation engine. The fact that our approach nearly achieves full mutation coverage is therefore a remarkable result. In addition to this high mutation score, our approach guarantees completeness with respect to a fault model that may contain a fault domain of infinite size. Mutation testing guarantees only the absence of certain syntactically seeded faults, which represent a necessarily finite fault domain which is dependent on the syntactical representation of the reference model and the mutation operators used.

7.4.2 SW-Mutation Analysis

A variety of approaches and tools related to SW mutation testing exist. In the late seventies, [DLS78] first proposed the use of simple errors in programs to assess test data. The authors suggested selecting test data that is able to reveal these errors and make use of the coupling effect: I.e., complex error conditions are composed of simple errors. Thus, test data that reveals simple errors will also be sufficient to reveal complex errors. Since then, a wide range of SW mutation testing and analysis tools emerged. [Jus14] presents the Major mutation framework,[1] which makes it possible to mutate Java programs and run JUnit test cases against the mutants. For the practical application of SW mutation analysis, mutation tools are mainly concerned with optimisations to allow for scalability for large programs. [Jus14] uses a compiler-integrated mutation engine that implements optimisations. Most mutation tools use a common set of traditional mutation operators that are mainly based on replacement and deletion of arithmetical, logical and conditional operators, of constants and of statements. Besides these traditional mutation operators, [MOK05] proposes object-oriented mutation operators for Java programs. These operators have been implemented in the μJava tool[2].

[1] see `http://mutation-testing.org/`
[2] see `https://cs.gmu.edu/~offutt/mujava/`

7.4.3 HW-Fault Injection

[CP95] gives a thorough overview of fault injection methods. The main objective of the fault injection methods discussed there and in the references given there is the assessment of a system's dependability. To this end, faults are injected into a safety-critical system to determine whether the fault-tolerance mechanisms of the system are sufficient. The injected faults simulate hardware-related permanent and transient faults. Some of the approaches model software component faults as well. These faults may result from the functional incorrectness of the system resulting in the crash of a software component. However, these classical fault injections methods are not directly applicable for the test-strength evaluation of the HSI tests. Our evaluation approach aims at the detection of systematic functional errors that result in permanent non-conforming I/O behaviour of the SUT. Hardware faults, at least transient faults, relate to unexpected events that are triggered by external events. Software component faults as simulated by fault injection methods mostly correlate with runtime errors. Recall that such errors are out of the scope of our HSI test-evaluation approach, as functional tests are not beneficial for the detection of runtime errors. We assume that abstract interpretation [KF16] is used before HSI tests are applied.

7.4.4 SystemC-Based Fault Injection

[MMGS14] presents an approach that uses SystemC as the target language of a fault injection approach. The authors propose applying fault injection on multiple levels. Faults are injected at the register transfer level (RTL). Additionally, high-level errors are injected at the behavioural level. This is achieved by manipulation of variable values. The main contribution of [MMGS14] is an improved approach to the assessment of the fault tolerance of a system by reducing the number of masked faults. The authors of [PAP10] propose a similar approach by using fault injection methods on SystemC models for an early evaluation of fault-tolerance mechanisms during the design phase. The approach is demonstrated on a safety-critical odometry example from the ETCS specification. However, the approaches proposed in [MMGS14, PAP10] are not directly applicable to the test-strength evaluation of HSI tests, as their work focuses on transient faults on the hardware and software component levels. Again, our approach is not intended to be used for the detection of transient and runtime errors.

7.4.5 High-Order Mutation Testing

In our experiments, we observed some limitations of first-order mutants. In our case, first-order mutants were not sufficient to evaluate the test strength for erroneous implementations whose DFSM abstractions contain additional states. This observation of limitations is in line with the results of [JH09]. The authors propose the use of high-order mutants. They are specially interested in the identification of subsuming mutants: i.e., mutants that are harder to kill than the first-order mutants they are constructed from. [JH09] reports some empirical results obtained from ten non-trivial case studies. The results there led to the publication of a manifesto for high-order mutation testing [HJL10]. In [HJL10], the authors argue against the wide-spread belief that high-order mutations are too expensive because of an exponential explosion of the possible number of high-order mutants. Search-based optimisation can be used to efficiently calculate subsuming mutants from the theoretically infinite set of possible high-order mutants. Second,

the authors thoroughly discuss widespread beliefs and wrong conclusions from the competent-programmer hypothesis, which states that incorrect versions of a program written by a competent programmer are very likely to be almost correct and thus are very close to the correct version. The authors argue that 'semantically close' does not necessarily indicate 'syntactically close'.

8 Conclusion

In this work, we presented a complete testing strategy applicable to reactive systems with potentially infinite input domains. To exhaustively test such systems, an infinite set of test cases would be required. However, under the prerequisite that the internal and output variables are from a finite domain, equivalence class partitioning can be used to abstract these systems to DFSMs. This allows for the application of complete DFSM testing theories on the abstract DFSM level. Calculation of concrete test cases from abstract test cases yields a concrete test suite that is applicable to the concrete SUT. It has been proven that the overall approach is complete–i.e., sound and exhaustive for a formal fault model–by the authors of [HP16a]. Thus, two of the main challenges in testing–namely, the selection of adequate test cases and test data and the justification of this adequateness–are remedied by this approach.

We have shown that this approach is feasible by implementation of a fully automated workflow for test-case generation from SysML state machines. The tool can easily be augmented to deal with other description means, given that the behavioural semantics can be expressed by RIOSTSs. Furthermore, we have demonstrated that the test-case generation scales up for real-world case studies—including a test model for a modern interlocking system of very high complexity.

While the completeness property of our approach is beneficial to justify the approach from a formal perspective, at the end, a testing strategy can only be considered reasonable, if (1) the testing effort is reasonably low (usually the testing effort is bounded by some project budgets limiting resources such as time and money for a test campaign of an SUT), and (2) the testing strategy has a high test strength: i.e., it is able to find as many errors as possible. For many MBT strategies, it is not possible to define a formal fault model. Therefore, a comparison of test strength by comparison of fault models is in most cases infeasible. Furthermore, the connection of fault models to real faults is hard to establish by mathematical means. To show that the test strength of our approach is reasonably high, we therefore proposed a novel mutation-analysis approach aiming at the evaluation of the test strength in the context of HSI testing. For a comparison, we used random testing. The experiments have shown the superiority of the ECPT approach against random testing. Furthermore, the experiments show that the ECPT approach in its original form can be enhanced by a combination of randomisation and boundary-value selection. The improvements to the ECPT approach we propose in this work are able to significantly improve the mutation score of the ECPT approach without increasing testing effort. We consider the mutation score achieved for all considered case studies–ranging from 90 percent up to 98 percent–a very high value. This value can be achieved if the ECPT is combined with randomisation and boundary-value selection. We therefore propose this combination as the best alternative to yield a testing theory that is complete with respect to the fault model, as proven in [HP16a] and shows a very high test strength in experiments using mutations and examining SUTs that are independent of the fault domain. In fact, many of the mutants used in our experiments are outside the fault domain. At least all mutants that are not killed by the original ECPT strategy, denoted as STRAT-1, must be outside the fault domain. SUTs that are outside the fault domain because of a greater number of states in their DFSM representation impose special challenges on our testing approach. While in general, the W/Wp-method makes

it possible to generate complete test suites for arbitrarily large SUTs, the resulting test suite will grow exponentially with the expected number of states in the SUT. We presented heuristics that are based on the uniform random generation of paths in a graph. These heuristics have been shown to significantly improve the test strength using DFSM mutations. Still, the number of test cases was not increased by these heuristics.

The selection of models and the wide range of systems that we examined gives hope that the results shown in this work are generalisable to other systems as well. In particular, we have demonstrated the applicability of ECPT testing to the railway domain. It can be summarised that ECPT with its guaranteed completeness with respect to a fault model and its experimentally measured high mutation scores can be considered a valuable formal method for the verification of interlocking systems and safety-related train on-board computer functionality. We believe that our approach is able to ensure the safety and reliability of complex embedded systems in the railway domain and other application areas. For the interlocking case studies, we have additionally shown how model checking and MBT in combination [HP15] can be used to verify that an implementation fulfills its specification, which itself has been proven to fulfil safety constraints [VHP17].

Besides the notable results presented in this work, there are many opportunities for future research. While we used SysML state machines in this work for the modelling of system behaviour, our approach is in general applicable to every formalism whose semantics can be expressed by RIOSTSs. In our mutation-analysis approach, we used SystemC models as the modelling language for the implementation used for the experiments. SystemC might be a candidate for a modelling formalism to be supported by our ECPT testing strategy. It might be worthwhile to investigate the potential of the extraction of a transition relation from SystemC models.

Another possible extension of the ECPT approach is to allow for the modelling of physical constraints. For system tests, a general problem for testing is to find realistic input data. While our approach currently assumes that every input variable can change its value in an arbitrary way, it is necessary for system tests that inputs obey certain constraints. In particular, inputs that are related to physical entities are usually subject to physical constraints. One possible way to augment our approach is to allow for physical constraints to be modelled–for example by *parametrics*, a SysML diagram type used to model physical laws and other types of constraints.

In addition to the approaches for IECP refinement presented above, further extensions are possible. One possible approach is refinement using implementation details. If some of the implementation details of the SUT are known (e.g., if source code of parts of the SUT is available), they can be used to either, (1) check whether the IECP used for test-case generation is fine-grained enough or, (2) to refine the IECP using implementation details. The second option could be implemented, for example, by using conditions in if- and loop statements, assuming an implementation language like C/C++. But for programs of arbitrary form, this is not a trivial task. For example, C allows for expressions including function calls, which are not necessarily side-effect free. Even worse, C++ allows for polymorphism: For virtual member function calls, the behaviour can only be predicted at run-time. Exceptions introduce another level of complication. Therefore, it can be expected that, for general programs, it will most likely not be possible to extract an IECP that can be proven to fulfil Equation 2.78. However, some information of the code can be used to refine an IECP to get closer to an optimal IECP and to improve the test strength.

In this work, we have presented the integration of random test concepts and boundary-value tests to the ECPT strategy. Future work could investigate the integration of other test criteria to be

considered for IECP refinement or concrete value selection. Finally, for our proposed mutation-analysis approach, a detailed investigation of the representativeness for typical HSI faults needs to be conducted to counter possible threats to the validity of the experiments presented in this work.

We hope that the results obtained in this work and future research results will finally bring MBT and ECPT to industry and make the approach a best practice for the verification of safety-critical systems. Progress in the area of MBT is necessary to cope with the increasing complexity of embedded systems and to help to improve the safety and reliability of future technologies. With this work, we demonstrate that MBT is both applicable and beneficial due to its automation and fault-detection capabilities.

Bibliography

[ABC+13] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[ABL05] James H. Andrews, Lionel C. Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 402–411. ACM, 2005.

[ABM98] Paul Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In *Second IEEE International Conference on Formal Engineering Methods, ICFEM 1998, Brisbane, Queensland, Australia, December 9-11, 1998, Proceedings*, pages 46–55. IEEE Computer Society, 1998.

[ADLU91] Alfred V. Aho, Anton T. Dahbura, David Lee, and M. Ümit Uyar. An optimization technique for protocol conformance test generation based on UIO sequences and rural chinese postman tours. *IEEE Trans. Communications*, 39(11):1604–1615, 1991.

[AIB10] Andrea Arcuri, Muhammad Zohaib Z. Iqbal, and Lionel C. Briand. Black-box system testing of real-time embedded systems using random and search-based testing. In Alexandre Petrenko, Adenilso da Silva Simão, and José Carlos Maldonado, editors, *Testing Software and Systems - 22nd IFIP WG 6.1 International Conference, ICTSS 2010, Natal, Brazil, November 8-10, 2010. Proceedings*, volume 6435 of *Lecture Notes in Computer Science*, pages 95–110. Springer, 2010.

[AT12] Morten Aanæs and Hoang Phuong Thai. Modelling and verification of relay interlocking systems. *Series: IMM-MSC-2012-14. Master's thesis. Technical University of Denmark, DTU Informatics*, 2012.

[BABJ10] Bill Bunton, Anna Keist, David C. Black, and Jack Donovan. *SystemC: from the ground up*. Springer, New York, NY, 2. ed. edition, 2010.

[BCC97] Sergey Berezin, Sérgio Vale Aguiar Campos, and Edmund M. Clarke. Compositional reasoning in model checking. In Willem P. de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference, International Symposium, COMPOS'97, Bad Malente, Germany, September 8-12, 1997. Revised Lectures*, volume 1536 of *Lecture Notes in Computer Science*, pages 81–102. Springer, 1997.

[Bel10] Axel Belinfante. Jtorx: A tool for on-line model-driven test derivation and execution. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of*

Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings, volume 6015 of *Lecture Notes in Computer Science*, pages 266–270. Springer, 2010.

[BHH+14] Cécile Braunstein, Anne Elisabeth Haxthausen, Wen-ling Huang, Felix Hübner, Jan Peleska, Uwe Schulze, and Linh Vu Hong. Complete model-based equivalence class testing for the ETCS ceiling speed monitor. In Stephan Merz and Jun Pang, editors, *Formal Methods and Software Engineering - 16th International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, Luxembourg, November 3-5, 2014. Proceedings*, volume 8829 of *Lecture Notes in Computer Science*, pages 380–395. Springer, 2014.

[BHP+14] Cécile Braunstein, Wen-ling Huang, Jan Peleska, Uwe Schulze, Felix Hübner, Anne E. Haxthausen, and Linh Vu Hong. A SysML Test Model and Test Suite for the ETCS Ceiling Speed Monitor. Technical report, Embedded Systems Testing Benchmarks Site, April 2014. Available under `http://www.mbt-benchmarks.org`.

[Boo67] Taylor L. Booth. *Sequential machines and automata theory.* Wiley, 1967.

[BPF15] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. νz - an optimizing SMT solver. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of *Lecture Notes in Computer Science*, pages 194–199. Springer, 2015.

[CCF+06] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of abstractions in the astrée static analyzer. In Mitsu Okada and Ichiro Satoh, editors, *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference, Tokyo, Japan, December 6-8, 2006, Revised Selected Papers*, volume 4435 of *Lecture Notes in Computer Science*, pages 272–300. Springer, 2006.

[CCF+09] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Why does astrée scale up? *Formal Methods in System Design*, 35(3):229–264, 2009.

[Cho78] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Software Eng.*, 4(3):178–187, 1978.

[CKMT10] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. Adaptive random testing: The ART of test case diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.

[CP95] Jeffrey A. Clark and Dhiraj K. Pradhan. Fault injection: A method for validating computer-system dependability. *IEEE Computer*, 28(6):47–56, 1995.

[CTY01] Tsong Yueh Chen, T. H. Tse, and Yuen-Tak Yu. Proportional sampling strategy: a compendium and some insights. *Journal of Systems and Software*, 58(1):65–81, 2001.

[DBI12] Dimitris Dranidis, Konstantinos Bratanis, and Florentin Ipate. JSXM: A tool for automated test generation. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings*, volume 7504 of *Lecture Notes in Computer Science*, pages 352–366. Springer, 2012.

[DF93] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In Jim Woodcock and Peter Gorm Larsen, editors, *FME '93: Industrial-Strength Formal Methods, First International Symposium of Formal Methods Europe, Odense, Denmark, April 19-23, 1993, Proceedings*, volume 670 of *Lecture Notes in Computer Science*, pages 268–284. Springer, 1993.

[DGG04] Alain Denise, Marie-Claude Gaudel, and Sandrine-Dominique Gouraud. A generic method for statistical testing. In *15th International Symposium on Software Reliability Engineering (ISSRE 2004), 2-5 November 2004, Saint-Malo, Bretagne, France*, pages 25–34. IEEE Computer Society, 2004.

[Dij72] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972.

[DLL62] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[DLS78] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, 1978.

[dVT00] René G. de Vries and Jan Tretmans. On-the-fly conformance testing using SPIN. *STTT*, 2(4):382–393, 2000.

[ECS09] ECSS. *ECSS-E-ST-40C - Software General Requirements*. ECSS (European Cooperation for Space Standardization), 2009.

[EKRV06] Juhan P. Ernits, Andres Kull, Kullo Raiend, and Jüri Vain. Generating tests from EFSM models using guided model checking and iterated search refinement. In Klaus Havelund, Manuel Núñez, Grigore Rosu, and Burkhart Wolff, editors, *Formal Approaches to Software Testing and Runtime Verification, First Combined International Workshops, FATES 2006 and RV 2006, Seattle, WA, USA, August 15-16, 2006, Revised Selected Papers*, volume 4262 of *Lecture Notes in Computer Science*, pages 85–99. Springer, 2006.

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[Eur96] European Council. Council Directive 96/48/EC of 23 July 1996 on the interoperability of the trans-European high-speed rail system. In *Official Journal of the European Communities L.235*, pages 6–24. June 1996.

[Eur01] European Committee for Electrotechnical Standardization. *EN 50128 – Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems*. CENELEC, Brussels, 2001.

[Eur02] European Commission. 2002/731/EC: Commission Decision of 30 May 2002 concerning the technical specification for interoperability relating to the control-command and signalling subsystem of the trans-European high-speed rail system referred to in Article 6(1) of Council Directive 96/48/EC. In *Official Journal of the European Communities L.245*, pages 37–142. September 2002.

[FG09] Gordon Fraser and Angelo Gargantini. An Evaluation of Model Checkers for Specification Based Test Case Generation. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, ICST '09, pages 41–50, Washington, DC, USA, 2009. IEEE Computer Society.

[FHP02] Eitan Farchi, Alan Hartman, and Shlomit S. Pinter. Using a model-based test generator to test for standard conformance. *IBM Systems Journal*, 41(1):89–110, 2002.

[FTW04] Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. Test generation based on symbolic specifications. 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*, volume 3395 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2004.

[FvBK+91] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Trans. Software Eng.*, 17(6):591–603, 1991.

[FZC94] Philippe Flajolet, Paul Zimmermann, and Bernard Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *Theor. Comput. Sci.*, 132(2):1–35, 1994.

[Gau95] Marie-Claude Gaudel. Testing can be formal, too. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22-26, 1995, Proceedings*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 1995.

[GG75] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Trans. Software Eng.*, 1(2):156–173, 1975.

[GGSV02] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Generating finite state machines from abstract state machines. In Phyllis G. Frankl, editor, *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*, pages 112–122. ACM, 2002.

[Gil62] Arthur Gill. *Introduction to the theory of finite-state machines*. New York: McGraw-Hill, 1962.

[Gin62] Seymour Ginsburg. *An Introduction to Mathematical Machine Theotry*. Addison-Wesley Publishing Company, 1962. Google-Books-ID: 8wYnAAAAMAAJ.

[Gon70] Guney Gonenc. A method for the design of fault detection experiments. *IEEE Trans. Computers*, 19(6):551–558, 1970.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[HdMR04] Grégoire Hamon, Leonardo Mendonça de Moura, and John M. Rushby. Generating efficient test sets with a model checker. In *2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), 28-30 September 2004, Beijing, China*, pages 261–270. IEEE Computer Society, 2004.

[HHP15] Felix Hübner, Wen-ling Huang, and Jan Peleska. Experimental evaluation of a novel equivalence class partition testing strategy. 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 155–172. Springer, 2015.

[HHP17] Felix Hübner, Wen-ling Huang, and Jan Peleska. Experimental evaluation of a novel equivalence class partition testing strategy. *Software & Systems Modeling*, pages 1–21, March 2017.

[Hie04] Robert M. Hierons. Testing from a nondeterministic finite state machine using adaptive state counting. *IEEE Trans. Computers*, 53(10):1330–1342, 2004.

[HJL10] Mark Harman, Yue Jia, and William B. Langdon. A manifesto for higher order mutation testing. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010, Workshops Proceedings*, pages 80–89. IEEE Computer Society, 2010.

[HLSU02] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A temporal logic based theory of test coverage and generation. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2280 of *Lecture Notes in Computer Science*, pages 327–341. Springer, 2002.

[HM09] Robert M. Hierons and Mercedes G. Merayo. Mutation testing from probabilistic and stochastic finite state machines. *Journal of Systems and Software*, 82(11):1804–1818, 2009.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[HN96] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[HP13] Wen-ling Huang and Jan Peleska. Exhaustive model-based equivalence class testing. In Hüsnü Yenigün, Cemal Yilmaz, and Andreas Ulrich, editors, *Testing Software and Systems - 25th IFIP WG 6.1 International Conference, ICTSS 2013, Istanbul, Turkey, November 13-15, 2013, Proceedings*, volume 8254 of *Lecture Notes in Computer Science*, pages 49–64. Springer, 2013.

[HP15] Felix Hübner and Jan Peleska. Integrated model-based testing and model checking with the benefits of equivalence partition testing. In Rolf Drechsler and Ulrich Kühne, editors, *Formal Modeling and Verification of Cyber-Physical Systems, 1st International Summer School on Methods and Tools for the Design of Digital Systems, Bremen, Germany, September 2015*, pages 287–289. Springer, 2015.

[HP16a] Wen-ling Huang and Jan Peleska. Complete model-based equivalence class testing. *STTT*, 18(3):265–283, 2016.

[HP16b] Wen-ling Huang and Jan Peleska. Complete model-based equivalence class testing for nondeterministic systems. *Formal Aspects of Computing*, pages 1–30, November 2016.

[JH09] Yue Jia and Mark Harman. Higher order mutation testing. *Information & Software Technology*, 51(10):1379–1393, 2009.

[JH11] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Software Eng.*, 37(5):649–678, 2011.

[JJI+14] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey, editors, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 654–665. ACM, 2014.

[JKDW01] Luc Jaulin, Michel Kieffer, Olivier Didrit, and Éric Walter. *Applied Interval Analysis*. Springer-Verlag, London, 2001.

[Jus14] René Just. The major mutation framework: efficient and scalable mutation analysis for java. In Corina S. Pasareanu and Darko Marinov, editors, *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 433–436. ACM, 2014.

[JW93a] Luc Jaulin and Eric Walter. Guaranteed nonlinear parameter estimation from bounded-error data via interval analysis. *Mathematics and Computers in Simulation*, 35(2):123–137, April 1993.

[JW93b] Luc Jaulin and Eric Walter. Set inversion via interval analysis for nonlinear bounded-error estimation. *Automatica*, 29(4):1053–1064, 1993.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[KATP02] Pieter W. M. Koopman, Artem Alimarine, Jan Tretmans, and Marinus J. Plasmeijer. Gast: Generic automated software testing. In Ricardo Pena and Thomas Arts, editors, *Implementation of Functional Languages, 14th International Workshop, IFL 2002, Madrid, Spain, September 16-18, 2002, Revised Selected Papers*, volume 2670 of *Lecture Notes in Computer Science*, pages 84–100. Springer, 2002.

[KF16] Daniel Kästner and Christian Ferdinand. Applying abstract interpretation to verify EN-50128 software safety requirements. In Thierry Lecomte, Ralf Pinger, and Alexander Romanovsky, editors, *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification - First International Conference, RSSRail 2016, Paris, France, June 28-30, 2016, Proceedings*, volume 9707 of *Lecture Notes in Computer Science*, pages 191–202. Springer, 2016.

[KLPU04] Nikolai Kosmatov, Bruno Legeard, Fabien Peureux, and Mark Utting. Boundary coverage criteria for test generation from formal models. In *15th International Symposium on Software Reliability Engineering (ISSRE 2004), 2-5 November 2004, Saint-Malo, Bretagne, France*, pages 139–150. IEEE Computer Society, 2004.

[KST+15] Willibald Krenn, Rupert Schlick, Stefan Tiran, Bernhard K. Aichernig, Elisabeth Jöbstl, and Harald Brandl. Momut: : UML model-based mutation testing for UML. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, pages 1–8. IEEE Computer Society, 2015.

[LGPC16] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Exponential recency weighted average branching heuristic for SAT solvers. In Dale Schuurmans and Michael P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, pages 3434–3440. AAAI Press, 2016.

[LvBP94] Gang Luo, Gregor von Bochmann, and Alexandre Petrenko. Test selection based on communicating nondeterministic finite-state machines using a generalized wp-method. *IEEE Trans. Software Eng.*, 20(2):149–162, 1994.

[MMGS14] Daniel Mueller-Gritschneder, Petra R. Maier, Marc Greim, and Ulf Schlichtmann. System C-based multi-level error injection for the evaluation of fault-tolerant systems. In *2014 International Symposium on Integrated Circuits (ISIC), Singapore, December 10-12, 2014*, pages 460–463. IEEE, 2014.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001.

[MOK05] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. Mujava: an automated class mutation system. *Softw. Test., Verif. Reliab.*, 15(2):97–133, 2005.

[Mon16] David Monniaux. A survey of satisfiability modulo theory. In Vladimir P. Gerdt, Wolfram Koepf, Werner M. Seiler, and Evgenii V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing - 18th International Workshop, CASC 2016, Bucharest, Romania, September 19-23, 2016, Proceedings*, volume 9890 of *Lecture Notes in Computer Science*, pages 401–425. Springer, 2016.

[NT81] S. Naito and M. Tsunoyama. Fault detection for sequential machines by transition tours. In *Proc. IEEE Fault Tolerant Comput. Conf.*, pages 162–178, 1981.

[Obj15a] Object Management Group. *OMG Systems Modeling Language*. September 2015.

[Obj15b] Object Management Group. *OMG Unified Modeling Language*. March 2015.

[Pac02] J. Pachl. *RAILWAY OPERATION AND CONTROL*. 2002.

[PAP10] Jon Pérez, Mikel Azkarate-askasua, and Antonio Perez. Codesign and simulated fault injection of safety-critical embedded systems using systemc. In *Eighth European Dependable Computing Conference, EDCC-8 2010, Valencia, Spain, 28-30 April 2010*, pages 221–229. IEEE Computer Society, 2010.

[Pel13] Jan Peleska. Industrial-strength model-based testing - state of the art and current challenges. In Alexander K. Petrenko and Holger Schlingloff, editors, *Proceedings Eighth Workshop on Model-Based Testing, MBT 2013, Rome, Italy, 17th March 2013.*, volume 111 of *EPTCS*, pages 3–28, 2013.

[PH16] Jan Peleska and Wen-ling Huang. Model-based testing strategies and their (in)dependence on syntactic model representations. In Maurice H. ter Beek, Stefania Gnesi, and Alexander Knapp, editors, *Critical Systems: Formal Methods and Automated Verification - Joint 21st International Workshop on Formal Methods for Industrial Critical Systems and 16th International Workshop on Automated Verification of Critical Systems, FMICS-AVoCS 2016, Pisa, Italy, September 26-28, 2016, Proceedings*, volume 9933 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 2016.

[PHH16a] Jan Peleska, Wen-ling Huang, and Felix Hübner. A novel approach to HW/SW integration testing of route-based interlocking system controllers. In Thierry Lecomte, Ralf Pinger, and Alexander Romanovsky, editors, *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification - First International Conference, RSSRail 2016, Paris, France, June 28-30, 2016, Proceedings*, volume 9707 of *Lecture Notes in Computer Science*, pages 32–49. Springer, 2016.

[PHH16b] Jan Peleska, Wen-ling Huang, and Felix Hübner. A Novel Approach to HW/SW Integration Testing of Route-Based Interlocking System Controllers. Technical report, accessible under http://www.informatik.uni-bremen.de/agbs/jp/papers/peleska-huang-hubner-rssr-2016.html, March 2016.

[PHH18] Jan Peleska, Wen-ling Huang, and Felix Hübner. Complete Model-based Testing. In Andreas Spillner, Mario Winter, and Andrej Pietschker, editors, *Test, Analyse und Verifikation von Software - gestern, heute, morgen*, pages 81–92. dpunkt.verlag, 2018.

[PM11] Mike Papadakis and Nicos Malevris. Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing. *Software Quality Journal*, 19(4):691–723, 2011.

[PM12] Mike Papadakis and Nicos Malevris. Mutation based test case generation via a path selection strategy. *Information & Software Technology*, 54(9):915–932, 2012.

[PVL11] Jan Peleska, Elena Vorobev, and Florian Lapschies. Automated Model-Based Testing with RT-Tester, Technical Report 2011-05-25, 2011.

[PY14] Alexandre Petrenko and Nina Yevtushenko. Adaptive testing of nondeterministic systems with FSM. In *15th International IEEE Symposium on High-Assurance Systems Engineering, HASE 2014, Miami Beach, FL, USA, January 9-11, 2014*, pages 224–228. IEEE Computer Society, 2014.

[PYvB96] Alexandre Petrenko, Nina Yevtushenko, and Gregor von Bochmann. Fault models for testing in context. In Reinhard Gotzhein and Jan Bredereke, editors, *Formal*

Description Techniques IX: Theory, application and tools, IFIP TC6 WG6.1 International Conference on Formal Description Techniques IX / Protocol Specification, Testing and Verification XVI, Kaiserslautern, Germany, 8-11 October 1996, volume 69 of *IFIP Conference Proceedings*, pages 163–178. Chapman & Hall, 1996.

[RC81] Debra J. Richardson and Lori A. Clarke. A partition analysis method to increase program reliability. In Seymour Jeffrey and Leon G. Stucki, editors, *Proceedings of the 5th International Conference on Software Engineering, San Diego, California, USA, March 9-12, 1981.*, pages 244–253. IEEE Computer Society, 1981.

[Rei97] Stuart Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In *4th IEEE International Software Metrics Symposium (METRICS 1997), November 5-7, 1997, Albuquerque, NM, USA*, pages 64–73. IEEE Computer Society, 1997.

[RTC92] RTCA. *Software Considerations in Airborne Systems and Equipment Certification RTCA DO-178B/ED-12B.* RTCA, Inc., 1992.

[SD85] Krishan K. Sabnani and Anton T. Dahbura. A new technique for generating protocol test. In William Lidinsky and Bart W. Stuck, editors, *SIGCOMM '85, Proceedings of the Ninth Symposium on Data Communications, British Columbia, Canada, September 10-12, 1985*, pages 36–43. ACM, 1985.

[Sil99] João P. Marques Silva. The impact of branching heuristics in propositional satisfiability algorithms. In Pedro Barahona and José Júlio Alferes, editors, *Progress in Artificial Intelligence, 9th Portuguese Conference on Artificial Intelligence, EPIA '99, Évora, Portugal, September 21-24, 1999, Proceedings*, volume 1695 of *Lecture Notes in Computer Science*, pages 62–74. Springer, 1999.

[SLD92] Yinan N. Shen, Fabrizio Lombardi, and Anton T. Dahbura. Protocol conformance testing using multiple UIO sequences. *IEEE Trans. Communications*, 40(8):1282–1287, 1992.

[Tre96a] Jan Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29(1):49–79, 1996.

[Tre96b] Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.

[UNI12] UNISIG. ERTMS/ETCS System Requirements Specification, Chapter 3, Principles. volume Subset-026-3. 2012. Issue 3.3.0.

[VBM15] Mahsa Varshosaz, Harsh Beohar, and Mohammad Reza Mousavi. Delta-oriented fsm-based testing. In Michael J. Butler, Sylvain Conchon, and Fatiha Zaïdi, editors, *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings*, volume 9407 of *Lecture Notes in Computer Science*, pages 366–381. Springer, 2015.

[VHP17] Linh Hong Vu, Anne Elisabeth Haxthausen, and Jan Peleska. Formal modelling and verification of interlocking systems featuring sequential release. *Sci. Comput. Program.*, 133:91–115, 2017.

[Vu15] Linh Hong Vu. *Formal Development and Verification of Railway Control Systems – In the context of ERTMS/ETCS Level 2.* PhD thesis, Technical University of Denmark, Kongens Lyngby, 2015.

[Wei09] Stephan Weißleder. *Test Models and Coverage Criteria for Automatic Model-Based Test Generation with UML State Machines.* PhD thesis, Humboldt-Universität zu Berlin, 2009.

[WO80] Elaine J. Weyuker and Thomas J. Ostrand. Theories of program testing and the the application of revealing subdomains. *IEEE Trans. Software Eng.*, 6(3):236–246, 1980.

List of Figures

List of Tables

List of Listings

Glossary

CSM Ceiling speed monitor. The ceiling speed monitor denotes a mode of the speed-and-distance monitoring function of the on-board computer of ETCS trains. This mode is active when the train is moving on the track without approaching its target destination. In this mode, the on-board computer supervises the agreement of the current train speed with the current speed limit. 23, 70, 71, 91, 124, 127–129, 131, 132, 135–138

DFSM Deterministic finite-state machine. A method for the description of finite-state systems, also known as finite automaton. In this work, we are using transducers: i.e., finite-state machines with an input and output alphabet. 14, 33, 38–47, 59, 62–67, 71, 73–75, 77, 88–90, 92, 94, 106, 108, 109, 115, 116, 118, 127, 128, 134–137, 139, 140, 142, 147, 149, 150, 163, 165

DNF Disjunctive normal form. A special form of a Boolean formula that is composed of ORs of clauses. Clauses are ANDs of literals. Literals are variables in positive form or negated. 110, 113, 143

DSL Domain specific language. A modelling language that is application specific. 14, 80

ECPT Equivalence-class partition testing. Our testing approach, which is based on input and state equivalence classes. The partitioning of infinitely many concrete inputs to a finite number of input-equivalence classes allows for the application testing of theories that are complete with respect to a fault model. 15, 16, 79, 80, 88, 89, 92, 95, 101, 109, 127–138, 141–143, 145, 149–151, 163

ERTMS European rail-traffic management system. A standard for European rail-traffic management. It is composed of ETCS and GSM-R: a mobile communication standard. 19

ETCS European train-control system. The subsystem of ERTMS which includes signalling, train control and train-protection functionality. 15, 19, 20, 23, 69, 147

EVC European vital computer. The on-board computer of ETCS trains. 23, 69, 70

FSM Finite-state machine. A method for describing finite-state systems, also known as finite automaton. In this work we are using transducers, i.e., finite-state machines with an input and output alphabet. 14, 19, 37, 38, 44, 48, 49, 59, 60, 142–144, 146, 163

HSI HW/SW integration is a phase in the lifecycle of system development. The phase is characterized by the integration of HW and SW components. Usually, the phase is followed by the system integration phase. 15–17, 24, 33, 79, 80, 106, 117, 119–121, 127, 134, 139, 147, 149, 151

HSI test HW/SW integration tests are tests performed in the HW/SW integration phase of system development. 16, 35

IEC An input equivalence class is a member of an input equivalence-class partitioning. An input equivalence class contains inputs that produce exactly the same outputs when applied to any quiescent system state. 16, 33, 58, 59, 62, 63, 66, 74, 88, 90–96, 98–105, 108, 109, 113, 114, 128, 133–136, 142, 144, 145

IECP An equivalence class partitioning of the input domain of a RIOSTS that has the property, that every input equivalence class contains inputs that produce exactly the same outputs when applied to any quiescent system state. 58–60, 62–64, 66, 67, 71, 72, 74, 75, 88, 90–94, 102, 105, 106, 113, 127, 128, 134, 144, 145, 150, 151, 163

INF Index-normal-form. A normalised form of the transition-logical predicate describing the transition relation of a RIOSTS. This form can be used to derive an IECP of the RIOSTS. 61, 62, 110, 114

LTS Labelled transition system. A description means for systems. A system is described by means of states and transitions that are labelled by events. 37, 142, 146

MBT Model-based testing. Approach for the verification of systems. MBT is based on the automatic generation of test cases from formal test models. 14, 15, 20, 25, 36, 37, 72, 79–81, 88, 120, 138, 141–143, 149–151, 163

MC/DC Modified condition/decision coverage. A coverage metric for decisions. A decision is a Boolean formula composed of conditions by logical operators. Each condition of a decision must take each possible value at least once, and values must be chosen such that each condition has affected the overall evaluation of the decision at least once (meaning that a change in any of the other conditions will change the overall outcome of the decision). 105, 141

PRNG Pseudo random number generator. 96, 100

RIOSTS Reactive input-output state-transition system. A method for describing state-based reactive systems. The behavior of these systems is influenced through the setting of input variables by the environment of the system. The behaviour of the system can be observed through the values of the output variables. 25, 50–55, 57, 59–63, 66, 67, 88, 89, 92, 94, 111, 141, 149, 150

RSM Release speed monitor. A mode of the speed-and-distance monitoring function of the on-board computer of ETCS trains that is active when a train is very close to its target destination, where it is to come to a standstill. 23

RT Random testing. A light-weight testing approach in which test inputs are randomly drawn from the input domain. 16, 119, 127–129, 131–138, 144, 145

SAT Boolean satisfiability problem. An NP-complete decision problem. The problem is to find a solution for a Boolean formula or to prove that no such formula exists. 94–96, 115

SEC A state equivalence class is a member of a state equivalence-class partitioning. A state equivalence class collects states that produce exactly the same outputs for every possible input trace. 56–59, 74, 92, 93, 109

SECP State equivalence-class partitioning. An equivalence-class partitioning of the state space of a RIOSTS that has the property, that every state equivalence class contains states that produce exactly the same outputs for every possible input trace. 56, 57, 59–64, 67, 71, 74, 92, 93, 110

SIVIA Set inverter via interval analysis. An algorithm based on interval analysis that is used to calculate the inner and outer approximation of a set. 96, 98–101

SMT Satisfiability modulo theories. An extension of the SAT problem to background theories such as integers or floating-point arithmetics. SMT instances are typically expressed by first-order logic. 88, 94–96, 100, 103, 104, 111, 113, 115, 131

STS State-transition system. A description means for state-based systems. 37, 50, 106

SUT System under test. The system that is going to be tested. 14–16, 33, 35–37, 51, 66, 67, 80, 88, 90–94, 99, 105, 106, 108, 109, 117–121, 124, 131, 133, 135, 137, 139, 142, 143, 147, 149, 150

SysML Systems modeling language. An extension to UML aiming at the modelling of systems. 14, 19, 23–26, 37, 52, 70, 72, 74, 88, 110, 118, 119, 141, 149, 150

TSM Target speed monitor. A mode of the speed-and-distance monitoring function of the on-board computer of ETCS trains that is active when the train approaches its target destination. 23

UML Unified modeling language. A standardised modelling language providing different description means for the modelling of software including class diagrams, state machines, activity diagrams and other items. 14, 24–26, 36, 37, 52, 87, 118, 119, 141, 142, 146