TU WIEN Informatics

Starkes modellbasiertes Mutationstesten

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

Andreas Fellner, M.Sc
Matrikelnummer 0825918

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assoc. Prof. Georg Weissenbacher, DPhil

Diese Dissertation haben begutachtet:

Keijo Tapio Heljanko	Görschwin Fey

Wien, 11. Jänner 2021

Andreas Fellner

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Strong model-based mutation testing

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Andreas Fellner, M.Sc
Registration Number 0825918

to the Faculty of Informatics

at the TU Wien

Advisor: Assoc. Prof. Georg Weissenbacher, DPhil

The dissertation has been reviewed by:

Keijo Tapio Heljanko	Görschwin Fey

Vienna, 11th January, 2021

Andreas Fellner

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Andreas Fellner, M.Sc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 11. Jänner 2021

Andreas Fellner

Acknowledgements

I thank Georg Weissenbacher for guiding me through this journey that was my PhD. Your insights and continuous feedback steered me well through this project. I thank Rupert Schlick and Willibald Krenn for providing me with the exciting opportunity of a project between science and industry as well as the freedom to pursue my interests within the topic. I thank Thorsten Tarrach for your endless help in small and large difficulties throughout my PhD. You made this endeavor feasible not only technically, but also mentally by being a good friend to me. I thank Mitra Tabaei Befrouei for the fruitful and efficient collaboration that significantly shaped parts of this thesis. I thank the Dependable Systems Engineering group at the AIT Austrian Institute of Technology as well as the LogiCS doctoral school and the Forsyte research group at the TU Wien for providing an environment of scientific excellence and warm friendship. I had many interesting on-topic as well as off-topic discussions with the members of these groups, gained valuable feedback and insights from our joint seminars, and had lots of fun visiting conferences and workshops together. A big group-hug goes to my EMCL people, you know who you are. You are true friends, you were early co-conspirators in the pursuit of the PhD, and provide the best opportunities for international get-aways. Shoutout to the Stahlburg folks! I am glad you don't need to understand what I am doing. Without being aware of it you contributed to this thesis by providing much needed (in)sanity. I thank my parents Bernhard and Eva for always standing by me. You, foremost of all people, made this possible. Finally, I thank my wife Marlene and my children Florentina and Konstantin, who endured times of long working hours and travel. You ground me, make me happy, and show me what really matters in life.

Kurzfassung

Leistungsstarke Verifikations- und Validierungsmethoden sind nötig um mit der Komplxität moderner Systeme, die immer häufiger aus einer Vielzahl an interagierenrden Sub-systemen bestehen, Schritt zu halten. Testen ist die weit verbreitetste V&V Methode, aber der Skalierung dieses Ansatzes auf große und ineinandergreifende Systeme stellt eine enorme Herausforderung im Bezug auf Rechenleistung, sowie (und womöglich schwerwiegender) im Bezug auf die Fähigkeit solche Systeme in ihrer Gesamtheit zu verstehen, dar. In dieser Arbeit adressieren wir diese Herausforderung indem wir skalierende Methoden für modellbasierte- und mutationsgetriebene Testfallgenerierung (MBMT) mit einem speziellen Fokus auf starke Mutationsanalyse, sowie nichtdeterministische Systeme nebenläufiger, sowie reaktiver Systeme präsentieren.

MBMT hat das Ziel automatisch Testfälle aus einem Modell des zu testenden Systems zu erstellen, welche etwaige Unterschiede in den Ausgaben dieses Modells und Mutanten derens sichtbar machen. Mutationen immitieren dabei Implementierunsfehler und da nachgewiesen wurde, dass künstliche und reale Fehler häufig durch die selben Tests aufgedeckt werden, gilt starke Mutationsanalyse weithin als eine Testabdeckungsmetrik mit hoher Qualität. Ihr Hauptnachteil ist ihr hoher nötiger Rechenaufwand. Außerdem ist starke Mutationsanalyse auf nichtdeterministischen System noch nicht hinreichend erforscht. Das ist unzufriedenstellend, weil Nichtdeterminismus ein nützliches Werkzeug in der Modellierung ist, um zum Beispiel Nebenläufigkeit, ein unbekanntes Umfeld, oder unterspezifizierte Aspekte des Systems, auszudrücken.

In dieser Arbeit widmen wir uns diesen Unzulänglichkeiten von starter Mutationsnalyse im Kontext der modellbasierten Testfallgenerierung. Wir beginnen mit einem rigorosen theoretischen Unterbau und betten diesen in die Theorie der Hyperproperties, welche den Zusammenhang mehrerer Ausführungspfade erforscht, ein. Diese Einbettung erlaubt es uns Hyperproperty model checking für MBMT zu nützen und dadurch rigorose Mutationsanalyse zu betreiben.

Des Weiteren präsentieren wir einen auf Skalierung getrimmten Algorithmus für MBMT und große Modelle mit einer komplexen Struktur, welcher Mutanten resourchenschonende ausführt und den Zustandsraum des Modells in einer verzweigenden Suche durchsucht. Final widmen wir uns der Testfallgenerierung von in hohem Maße nebenläufigen Modellen indem wir Testfallgenerierung mit partial order reduction verbinden. Hierzu übersetzen wir starke Mutationsanalyse in ein Inklusionsproblem über den Sprachen zweiter Ereignisstrukturen, beweisen die Komplexität dieses Problems und entwickeln einen Algorithmus zum Lösen des Problems. Diese Methodik erlaubt es uns eine neuartige Klasse an Testfällen zu erhalten, welche das nebenläufige Verhalten des zu testenden Systems abbildet.

Abstract

Powerful verification and validation methods are needed to keep up with the complexity of modern systems, which increasingly consist of a myriad of interacting sub-systems. While testing remains the most prevalent verification and validation method, scaling testing to huge and interdependent systems poses a big challenge in terms of computational complexity and (perhaps even more severely) in terms of understanding such systems in their entirety. In this work, we address this challenge by presenting scalable methods for model-based mutation testing with a focus on strong mutation and non-deterministic models of concurrent reactive systems.

Model-based testing aims to automatically create test cases from a model of some system under test. As has been done successfully before, we use mutations as the driving criterion in model-based test generation. Strong mutation analysis aims to reveal differences in the output of a model and mutations of it via test cases. Mutations mimic implementation errors and a key assumption of mutation testing is that the ability of a test suite to reveal artificial errors carries over to its ability to reveal actual faults in systems. Strong mutation is well accepted as a powerful test suite quality metric. However, its main drawback is the computational cost associated with it. Furthermore, strong mutation in presence of non-determinism is not well studied so far. This is unfortunate, since non-determinism is a useful modeling tool, for example to represent concurrency, to model an unknown environment, or to under-specify certain aspects of a system.

In this work, we tackle these shortcomings of strong mutation analysis in the context of model-based testing. We start by establishing a rigorous theoretical framework for strong model-based mutation testing in presence of non-determinism. We embedded this framework into the theory of hyperproperties, which studies the relationship of multiple system traces. This embedding yields a logic characterization of mutation killing. In addition, it enables model-based mutation testing via hyperproperty model checking and thus rigorous mutation analysis.

Towards mutation-driven test generation for large models with complex structure, we propose an explicit state and exploration-based test case generation algorithm that is tuned for scalability. It executes mutants lazily and explores state spaces in a branching search manner, which is inspired by the successful rapidly exploring random trees path planning algorithm. Finally, we enable test case generation for highly concurrent models by connecting this algorithm with event structure-based partial order reduction. To this end, we map strong killing onto a language inclusion problem over event structures, prove the computational complexity of this problem, provide a decision algorithm for it, and obtain a novel type of test cases that incorporates the concurrent behavior of the model.

Contents

CHAPTER 1

Introduction

1.1 Motivation

From implementing complex software systems over medical research to constructing a bridge, testing is an integral component of engineering and science, which makes up a significant percentage of overall development time and costs across many domains. However, inadequate testing in an early project stage can incur even higher costs in a later project stage, when a complex software system reveals a severe bug, a medical treatment proves harmful, or a bridge collapses. Unfortunately, manually creating test cases is not trivial and often an unbeloved task. The complexity of developed systems as well as tunnel vision can lead to untested corner cases. Furthermore, testing does not immediately produce value in terms of some tangible output. Therefore, it can be skipped or done sloppily, especially in early stages of development. Formal methods, such as static analysis, software model checking, or automated theorem proving, can fill certain gaps in the verification process and augment it in very useful ways. However, we are far from completely relying on such methods for quality and correctness assurance, for which testing remains the de-facto standard. All of these factors call for automated test generation methods to augment or replace the manual test generation process.

In this work, we study automated test generation for reactive systems, which we understand as systems that might run indefinitely and continuously process inputs as well as produce outputs. Automated test generation extracts test cases from some description of the reactive system under test. What constitutes a suitable description heavily depends on the system itself. A suitable and widely used description for software is its source code. However, for many systems there simply is no source code at all (e.g. physical systems) or source code that describes its entirety (e.g. cars). Furthermore, source code may contain too many implementation details that obscure test goals. Therefore, more abstract descriptions, such as designs or specifications, can be used as basis for automated test generation. Model-based testing subsumes methods to create test cases from an abstract description, i.e. model, of some system under test. Test cases created via model-based testing are templates that have to be concertized for the actual system under test. For example, a

test case for a railway interlocking system could be a script consisting of a sequence of actions that have to be performed on the components of the interlocking system, such as switches or relays, that should trigger a sequence of observations, such as signals or lights. Executing such a test means that some tester, which could either be a person or some automated system, executes the test's inputs, and compares the observed outputs to the ones prescribed by the test.

An orthogonal consideration in test case generation to the abstraction level of the model is the purpose of a test. The purpose of a test could be to exhibit as much diverse behavior of a system as possible. Alternatively, its purpose could be to detect errors of the system, if there are any. Such different test purposes can be captured via the notion of test coverage metrics. The former purpose can be captured by structural coverage metrics, such as branch or decision coverage. The latter purpose can be captured by fault-based coverage metrics, such as mutation coverage, which is the metric studied in this thesis.

A mutant is a copy of the model except for a small syntactic modification that mimics a potential implementation fault, such as an off-by-one error or a blocked transition. The mutation coverage metric measures how many differences between the model and its mutants can be revealed by a test or a whole test suite. We say that a test kills a mutant if it reveals a difference of the mutant and its underlying model. The fundamental assumptions of mutation testing are the *competent programmer hypothesis* [DLS78, BLDS79], which states that implementations typically are close to correct, and the *coupling effect* [DLS78, Off92], which states that a test suite's ability to detect many simple errors (and mutations) correlates with its ability to detect complex errors. In other words, while the competent programmer hypothesis states that the respective correct implementation is within a close neighborhood of most erroneous implementations, the coupling effect states that this neighborhood does not need to be searched exhaustively. Thus, the coupling effect reduces the search space of close-to-correct implementations from an astronomical or even infinite amount of potential implementations to a manageable amount of implementations that are the result of applying carefully selected mutation operators. In summary, a test suite that kills many mutants is desirable, because it is able to effectively differentiate between correct and faulty implementations.

The sought differences between model and mutant can be in terms of internal state (weak killing [How82]) or in terms of output (strong killing [DLS78]). The benefit of weak over strong killing is the reduced computational cost during test case generation or evaluation, because in order to demonstrate a weak kill, internal differences do not have to be propagated to outputs. However, since a difference in internal state may never (or only via very specific tests) propagate to a difference in output, the coverage guarantees obtained via weak mutation analysis are inherently weaker than those of strong mutation analysis. The trade-off between execution cost and coverage guarantees between weak and strong mutation was empirically evaluated in multiple studies [Mar91, OL91, OL94, KPM10] and the general agreement is that the computational savings of weak mutation are worthwhile, since the coverage reduction is small. However, in the realm of model-based testing, the internal structure of the model and the system under test may be completely different. As a result, differences in internal state may not manifest on the system under test. In contrast, reasonable models should include output that is closely related to the output of the system under test. Furthermore, while weak mutation testing of non-reactive programs

aborts mutant execution early, the resulting tests are assumed to be run on the system under test until termination. Since reactive systems do not terminate by definition, it is not clear how and how long to proceed with tests of such systems that were created via weak mutation, whereas tests created via strong mutation drive the system to a point where either the error is visible or it is not present. Therefore, strong mutation analysis is an appropriate and powerful coverage metric for model-based testing of reactive systems.

Non-determinism is an important modeling tool, for example to describe uncertainty of the environment, to abstract implementation details via under-specification, or to model concurrency. Therefore, there is a need for mutation analysis within the model-based setting to handle it well. Unfortunately, non-determinism induces complexity to strong mutation analysis that is not addressed thoroughly in the mutation testing literature. This complexity stems from the fact that tests created from non-deterministic models may subsume multiple of its execution paths. All these paths need to be taken into account when deciding whether the test kills a mutant. Non-determinism is both a challenge for the theory of mutation analysis, where the definition of mutation killing has to take into account multiple paths, as well as the practice of mutation analysis, where an exploding number of execution paths needs to be tamed.

1.2 Contributions

We address the theoretical challenge of adequately formulating non-determinism-sensitive mutation killing in a model-based setting, as well as the practical challenges of performing mutation-driven test case generation both rigorously and on a large scale on top of this formulation.

Towards the theoretical challenge, we define mutation testing semantics on Mealy machines, which are a universal and widely known theoretical formalism. Although Mealy machines do not capture all types of systems for which mutation testing is conceivable, by expressing finite state reactive systems, our semantics are nevertheless applicable to a large class of concrete systems. Furthermore, we map the powerful action systems modeling language as well as symbolic transition systems to Mealy machines and thereby demonstrate the universality of the given mutation testing semantics. In addition to incorporating the folklore notions of weak- and strong killing, our mutation testing theory establishes a novel and orthogonal degree of fault detection guarantee in presence of non-determinism. In particular, we introduce potential- and definite killing, which differentiate whether some or every execution of a test can be expected to detect the fault expressed by a killed mutant. Similar to weak- and strong killing, potential- and definite killing allow to trade-off test generation costs and coverage guarantees, since potential killing is cheaper to establish whereas definite killing induces stronger fault detection guarantees.

Model checking is a promising method for tackling the challenge of performing mutation-driven test case generation rigorously. In fact, the method has been applied successfully for test case generation. The basic idea is to encode test coverage into a model checking problem in such a way that a test case with some coverage guarantee is a counter-example to that problem. While this approach has been applied to mutation coverage before, e.g. [CDK85, ABM98, GH99, BOY00, OBY03, SFBD08, XeAXW12], existing solutions often lack universality and thus the approach is not widely adopted. In this thesis, we address one of the main barriers to a wider

adoption of model checking for mutation-driven test case generation. Namely, we show that strong killing is a hyperproperty [CDK85], i.e. a property that relates multiple execution paths. In contrast, classic model checking solves trace-properties, i.e. properties that are either true or false for individual execution paths. Thus, in order to leverage model checking for mutation-driven test case generation, previous approaches needed to work around this fundamental disparity in property type. This was typically achieved either by mutating properties of the model instead of the model itself, e.g. [ABM98, BOY00], by constructing and model checking the product of model and mutant, e.g. [OBY03], or by formulating model- and mutant- specific properties capturing some custom notion of killing, e.g. [GH99, XeAXW12]. In contrast, we enable a generic model checking approach for mutation-driven test case generation by embedding our mutation killing semantics into the theory of hyperproperties. Not only does this embedding yield a novel test case generation methodology, it also provides a characterization of killing in terms of logic, which in turn enables rigorous analysis of the concept. For example, a formal argument for the different computational costs of different killing concepts can be given in terms of properties of their respective logic formulations. Finally, the embedding provides a novel application to the emerging field of hyperproperties outside of its classic application domain, which is the verification of security properties.

Models of complex systems are often quite non-trivial themselves, because not all aspects of the system can and should be abstracted away. As a result, test case generation via model checking is not always feasible, either due to model sizes for which formal approaches hit their scalability ceiling, or due to complex syntactic constructs, which can not be handled well by logic solvers. Therefore, in order to succeed in the practical challenge of performing mutation-driven test case generation on a large scale, automated test case generation methods that can handle large and complex models are needed. To this end, we propose an explicit state and exploration-based test case generation algorithm for models given as action systems [BKS83, BKS98], which is a rich and versatile modeling language, capable of expressing a large class of other modeling formalisms, such as UML state machines [KSA09] or Event-B models [BDH+19]. The algorithm is parametrized by its exploration scope as well as by several heuristics. Thus, it can be well adjusted to model requirements and computation budget. For example, exhaustive explorations can be performed on small models, whereas bounded explorations might be more appropriate on large models. We evaluate this parameter space on a series of industrial case studies. In addition to efficient state space exploration, the algorithm performs lazy mutant execution. We discussed above that weak mutation was found to be worthwhile, since it reduces the execution costs of mutants by suppressing execution suffixes. However, we also discussed why weak mutation is not an ideal criterion in our setting. To gain the best of both worlds, lazy mutation reduces the execution costs by soundly suppressing prefixes of mutant executions in contrast to their suffixes. This is enabled by the use of conditional mutants allowing us to switch on mutants at intermediate states that are guaranteed to be reachable in the mutant and that have outgoing transitions affected by the respective mutation.

The complexity of systems can be due to their shear size. However, in the emerging era of the internet of things, this complexity often does not stem from the complexity of individual components, but from the large number of sub-systems that interact and co-operate, which of-

ten can act concurrently and independently of each other. In order to automatically create test cases from models of such system, dedicated methods to cope with the exploding number of execution interleavings represented by these models are necessary. To this end, in this work we show how unfolding-based partial order reduction can be leveraged in three aspects during test case generation. Firstly, unfolding-based partial order reduction can exponentially speed up model exploration. Secondly, the results of this exploration induce a novel type of concurrent test, representing a potentially exponential number of test cases that are equivalent up to concurrency interleaving. Thirdly, strong kill analysis can directly be performed on the unfolding structures obtained during unfolding-based partial order reduction without the need to construct the potentially exponential number of interleavings explicitly. This is achieved by casting the strong killcheck to a language inclusion problem over event structures. Since this problem was not solved before, we prove the computational complexity of the language inclusion problem over finite labeled prime event structures and provide a decision algorithm for it.

The main contributions of this work can be summarized as follows:

- Semantics for mutation analysis of reactive systems in the presence of non-determinism.
- A formal characterization of strong mutation analysis via hyperproperties.
- A model-based mutation testing algorithm that scales to models of industrial size.
- Partial order reduction enabled mutation-driven test case generation via event structures.
- Foundational results on the event structure language inclusion problem.

1.3 Research Questions

Accordingly, our work answers the following research questions:

- How does non-determinism influence mutation killability?
- Is strong killability a hyperproperty?
- Is there a logical characterization of strong killability?
- Can the logical characterization of strong killability be leveraged to automatically create test suites with high mutation coverage?
- Can model-based mutation testing scale to models of industrial size?
- Can partial order reduction be integrated into mutation analysis?
- What is the computational complexity of the finite labeled prime event structure language inclusion problem?
- How can the finite labeled prime event structure language inclusion problem be solved?

1.4 Methodology

The research questions presented above are of mixed theoretical and practical nature. This mix is reflected in our approach to answering these questions, ranging from purely theoretical computational complexity proofs to practical algorithms, whose value we demonstrate via experiments on large real world problem instances.

The methodologies of the individual experiments are described in detail in the following chapters. However, the general theme of all conducted experiments is to implement our method and to study its properties using base-line comparisons as well as large sets of benchmark models and parameter configurations. We implemented the methods presented in Chapter 5 as a toolchain of freely available off-the-shelf tools and the methods presented in Chapter 6 as well as Chapter 7 on top of the model-based mutation testing tool MoMuT. The experiments presented in Chapter 5 compare to MoMuT on a case-study and demonstrate computational as well as coverage properties of the method on a range of models originating from two different modeling languages. The experiments presented in Chapter 6 compare a range of parameter configurations on a series of industrial benchmark models. Finally, the experiments in Chapter 7 compare the proposed language inclusion algorithm to classical automaton-based language inclusion as well as unfolding-based partial order reduction to sequential model exploration.

The theoretical results are presented in a classic definition, theorem, and proof style. In addition, we discuss the meaning of our definitions and the implications of the proven theorems. This work aims to be self-contained and all necessary concepts are defined within it. Furthermore, all theoretical results are demonstrated on an simple examples.

1.5 Publications

This thesis is based on the following publications (joint work is presented with the permission of all co-authors):

- Andreas Fellner, Willibald Krenn, Rupert Schlick, Thorsten Tarrach, and Georg Weissenbacher. Model-based, mutation-driven test case generation via heuristic-guided branching search. In Jean-Pierre Talpin, Patricia Derler, and Klaus Schneider, editors, *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2017, Vienna, Austria, September 29 - October 02, 2017*, pages 56–66. ACM, 2017

- Andreas Fellner, Mitra Tabaei Befrouei, and Georg Weissenbacher. Mutation testing with hyperproperties. In Peter Csaba Ölveczky and Gwen Salaün, editors, *Software Engineering and Formal Methods - 17th International Conference, SEFM 2019, Oslo, Norway, September 18-20, 2019, Proceedings*, volume 11724 of *Lecture Notes in Computer Science*, pages 203–221. Springer, 2019

- Andreas Fellner, Willibald Krenn, Rupert Schlick, Thorsten Tarrach, and Georg Weissenbacher. Model-based, mutation-driven test-case generation via heuristic-guided branching search. *ACM Trans. Embed. Comput. Syst.*, 18(1):4:1–4:28, January 2019

- Andreas Fellner, Thorsten Tarrach, and Georg Weissenbacher. Language inclusion for finite prime event structures. In Dirk Beyer and Damien Zufferey, editors, *Verification, Model Checking, and Abstract Interpretation - 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16-21, 2020, Proceedings*, volume 11990 of *Lecture Notes in Computer Science*, pages 314–336. Springer, 2020

- Andreas Fellner, Mitra Tabaei Befrouei, and Georg Weissenbacher. Mutation testing with hyperproperties. *Software and Systems Modeling*, Special Issue, 2020. Accepted. Publication pending

CHAPTER 2

Related Research

In this section, we present the state of the art of model-based mutation testing and related fields. Additional topic specific related work is further discussed in the technical chapters of this work.

2.1 Model-based Testing

Model-based testing is a rich and diverse field of study. In fact [Bin00] argues that all testing must be model-based and if the notion of a model is viewed broad enough to include mental models of software engineers, in fact all testing is model-based [BJK+05]. Thus, in this assessment of the state of the art of model-based testing we restrict ourselves to tracing the origins of the field, pointing to comprehensive sources that provide an overview of the field, and discussing selected pieces of work that are most relevant to this thesis.

2.1.1 Historical Context

The broad definition of the term model prohibits to exactly pin-point the origin of model-based testing. However, the idea to automatically create test cases from an abstraction of the system under test was originally coined specification based testing, requirements-based testing, or testing for finite state machines. These related approaches were later subsumed under the term model-based testing in the late 90's.

The earliest model-based test case generation like approaches were conceived for sequential circuits for which the boundary between model and implementation is blurry. For example, [Hen64] considers the fault detection problem for sequential circuits and describes a procedure to obtain so called checking experiments from the state-table description of the circuit. The same problem is considered in [Gon70], which provides a more algorithmic solution and abstractly defines the testing problem for sequential circuits as the search for a pair of input and output sequence (X,Z), such that the circuit responds to X with Z if and only if the circuit is correct.

Towards automated test case generation for more universal types of models, [Cho78] presents an early approach to the problem for finite state machine representations of the control flow of some system under test by exhaustively constructing transition-covering input sequences.

[Gou83] presents a formal framework to assess the correctness of programs with respect to specifications as well as quantifying the power of test suites with respect to this measure. It can therefore be seen as a precursor to conformance testing, which emerged as an influential early model-based testing method in the field of testing communication protocols [CDK85, Tre92].

Interestingly, mutation testing is formally shown in [Gou83] to be a powerful test metric, although it is viewed as a test metric that solely depends on the implementation and is thus specification independent. In contrast, model-based mutation-driven test case generation turns this view around by generating tests solely based on specifications. This is by no means a novel idea, as can be seen from another early specification-based test case generation method [BG85] that studies mutants of specifications, as well as [PDMM94] that studies mutation-driven test case generation for finite state machines.

DAISTS [GMH81] is an early specification-based testing tool, which combines concrete implementations with specifications given as algebraic axioms. The tool automatically compiles both components together, providing a test driver that can verify the correctness of the specification using tests. Another early tool in this domain is ATLAS [JKRS76], which derives test cases automatically from an abstract directed graph representation of the sequential stimulus-response behavior of the system under test.

2.1.2 Surveys

In [UPL12] a taxonomy of model-based testing methods is presented, which demonstrates the diversity of the field well. The authors categorize model-based testing approaches according to three main criteria: model specification, test generation, and test execution. Each of the categories are further partitioned to eventually arrive at 25 distinct attributes to distinguish model-based testing methods. The approaches discussed in this work have the following attributes with respect to this taxonomy. We analyze models that are input-output, non-deterministic, and transition-based. The test case generation methods are fault-based and combine random generation, search-based algorithms, and model checking technology. Finally, we target offline test execution.

A detailed study of model-based testing of reactive system is presented in [BJK+05]. The book discusses conformance testing for finite state machines, given as Mealy machines. This problem is very similar to the problem we tackle, when mutation killability is cast as a conformance problem between the model (taking on the role of a specification) and the mutant (taking on the role of an implementation). Furthermore, [BJK+05] discusses practical aspects of model-based testing. For example, the process of test case concretization and aspects of model-driven development are explored. Furthermore, tools for model-based testing are examined. All these aspects are very relevant to our work and need to be considered in order to make full use of the methods we present.

Further surveys on model-based testing include [DNSVT07] that, similar to [UPL12], classifies approaches according to additional criteria like abstraction level, level of automation, and scope of application, [SL10, SL15, LLGS17] that focus on model-based testing tools, and [Pel13] that focuses on the application of model-based testing in an industrial context, which is also the aim of the methods presented in this work.

2.1.3 Related Topics

Model-based testing is particularly relevant in the context of **model-driven development**, which aims to develop systems via continuous refinement of ever more concrete models. The development, challenges, and reasons model-driven development is not (yet) widely adapted in industry are discussed in [MAB+14]. Furthermore, a survey of work on testing within model-driven development is presented in [MOSH09].

An interesting approach to ease the use of models is to automatically learn them from the system under test. Model-based testing can be used to guide this process by providing a teaching mechanism via generated test cases. A survey of work in the space of **model learning** and model-based testing is presented in [AMM+18]. In this work, we purely focus on the model-based testing aspect, but our methods could be leveraged within the model-based learning approach.

Conformance testing is the problem of verifying whether an implementation correctly implements or conforms to a specification. This is the problem we ultimately help solving in this work by generating test cases from the specification that support or reject conformance of implementations to this specification. This classic problem is tackled for Mealy machines in [LY96, BJK+05], as well as via a bounded model checking approach over 1-safe Petri nets in [PH03]. Conformance Testing is defined via relations between labeled transition systems in [Tre92, Tre96]. This interpretation of conformance testing is applied to mutation testing in [AJT14, Jöb14].

2.1.4 Relevant Modeling Formalisms

In this work, we build our theory of mutation testing on top of finite state Mealy machines [Mea55], which serve as archetypical reactive systems. However, in practice, models are rarely given in this abstract form. Thus, we demonstrate our test case generation methods on more practical modeling formalisms. We consider action systems [BKS83], which can be seen as an extension of the well known Dijkstra's guarded command language [Dij75]. In fact, our implementation considers an object oriented extension of action systems [BKS98]. Due to its expressivity this formalism can serve as both a standalone modeling language as well as the target of automated translations from other modeling formalisms. Currently, UML state machine models [ABJ+15a] as well as Event-B [Abr10b] models can automatically be translated to action systems via the MoMuT mutation testing framework. Additionally, we present a method for test case generation on symbolic transition systems as well as experiments on concrete forms of such transition systems given as SMV [McM92b] and Verilog [TM08] models.

2.2 Mutation Testing

2.2.1 Historical Context

Mutation testing originates in the late 70's [Ham77, DLS78]. [Ham77] studies the effects of altered expressions on the verdicts of a given test suite and proposes an automated compiler-based system to execute tests on programs with such altered expressions. [DLS78] proposes the coupling effect, studies common errors made by programmers, and establishes mutations as a systematic way to evaluate the error detection capability of a test suite.

Early mutation testing tool support was provided via PIMS [BLDS78, BLDS79], which is an interactive tool to measure the mutation score of test suites for Fortran programs. Another noteworthy early mutation testing tool is Mothra [DGM+88], which provides an interactive mutation system for Fortran and C. In addition, Mothra is capable of mutation-driven test case generation, which is the problem studied in this work. Later extensions to Mothra perform test case generation by solving constraints that express mutation killability [DO91], which can be seen as a precursor to model checking based test case generation and our work presented in Chapter 5.

2.2.2 Surveys

There exist excellent, comprehensive surveys on mutation testing that cover everything from fundamental assumptions over evaluating weak versus strong mutation to mutation testing tools. Most notable and most up-to-date are [JH11] and [PKZ+19]. Furthermore, [OU01] provides a good overview of the historic context of the topic. Finally, [Off11] presents a thorough overview and discussion of mutation testing from one of the founding fathers of the field, 35 years after it was established.

2.2.3 Fundamental Assumptions

As mentioned in the introduction, mutation analysis builds on two fundamental assumptions: the competent programmer hypothesis, i.e. implementations are typically close-to-correct, and the coupling effect, i.e. the ability to detect many minor errors correlates with the ability to detect complex errors. The competent programmer hypothesis was proposed in [BLDS79] and made formal in [BDLS80] by defining a program neighborhood for simple LISP programs. Implicitly, this assumption is made in most debugging and testing efforts, since programs are most often not fixed by completely rewriting them from scratch. Furthermore, the growing field of automated program repair (see [GMM18] for a comprehensive survey) likewise assumes that for many bugs, there exists a simple fix that can even be found automatically. The validity of the coupling effect has been shown within the Mothra mutation framework [Off89, Off92], for functions in the mathematical sense [Wah95, Wah00], for logical faults in Boolean constraints [Kap06], and with respect to multiple active mutations in comparison to a single active mutant [Wah03].

Since the two fundamental assumptions of mutation testing are essentially empiric claims, their validity can be argued and studied, but they can ultimately not be proven. Therefore, experiments were conducted to empirically evaluate the effectiveness of mutation testing. [DM90, DT96, ABL05, JJI+14] present experiments that measure the correlation of mutant and real

fault detection capability of test suites. All these experiments conclude that there is indeed a strong correlation and that careful selection of mutation operators strengthens this correlation. [MW94, FWH96, OPTZ96, MR01] present experiments that compare the effectiveness and efficiency of different test adequacy metrics, such as the data-flow all-use criterion in [MW94, FWH96, OPTZ96] and boundary value as well as equivalence class testing in [MR01]. The general conclusion of these studies is that mutation testing indeed is superior to the other test adequacy metrics in terms of fault detection capability, but the metric comes with caveats, as for example its high computational costs.

2.3 Model-based Mutation Testing

2.3.1 Historical Context

As hinted above, mutation testing was already applied in the realm of model-based testing in its early days. Mutation testing was applied to predicate calculus in [BG85], to network protocols in[SL88], to finite state machines in [PDMM94], to Petri nets in [FMM+95], and to UML diagrams modeling the interactions of objects in object oriented programs in [YCJ98]. These early approaches mostly aim to establish the validity of mutation testing on a design level as well as formulate adequate sets of mutation operators. We build on the shoulders of these giants, focusing on the test case generation problem and studying killability in presence of non-determinism.

2.3.2 Surveys

Both mutation testing surveys [JH11] and [PKZ+19] have dedicated chapters on model-based mutation testing. In addition, [BBH+16] provides a comprehensive study of the topic, including a detailed discussion of related work. The authors of [BBH+16] differentiate between approaches that test the specification and those that test the implementation. Our work falls into the latter category, as we assume the given model to be correct and aim to provide tests that should increase confidence that the system under test correctly implements the specification given by the model.

2.3.3 Mutation Operators

The vast majority of mutation testing research within model-based testing is concerned with studying mutation operators for models. Indeed, careful design and choice of mutation operators is an important problem for mutation testing, but it is not the main focus of this work. Nevertheless, we provide an overview of such research here, since mutation operators depend heavily on the used modeling formalism and our test case generation methods are meant to be applicable on a wide array of them. In particular, large parts of our work are directly applicable to modeling formalisms that express finite state machines. In [BBH+16] it is argued that mutation operators *insertion* and *omission* as well as combinations of them subsume all other mutation operators for such models. For example, a variable assignment replacement can be simulated with an omission, followed by an insertion. This is an interesting theoretical insight. However, in practice mutation operators manipulate higher language features than state machine transitions,

and allowing multiple simultaneously active mutants (which is typically referred to as higher order mutation) induces its own sets of problems like an exploding number of mutants and a more vague mapping between mutants and faults. Mutation operators for finite state-like modeling formalisms were defined for finite state machine directly in [PDMM94, FMMD99], for extended finite state machines in [BPGQ02, BVCU07], for probabilistic and/or stochastic finite state machines in [HM07, HM09, SMC+17], for SDL specifications in [KPLV+03, SMW04], for statechart models in [FMSM99, Tra10], for Estelle specifications in [SdSFLdSM00], for pushdown-automata in [BBTF11], for timed automata in [ALN13], and for UML state machines in [ABJ+15a].

Besides finite state machine-like models, mutation operators were defined for feature models of software product lines in [HPP+13b, HPP+13c, HPT14, AGV15], for algebraic specifications in [Woo93], for Simulink models in [SAC14, PRWN16], for XML schemas in [LO01, LM05a, XOL05], for Alloy models in [SWZK17], for aspect-oriented models in [XeAXW12, LAO+15], for agent-based models in [AM10], for SMV models in [AGR15, AGR17], for temporal logics in [Tra17], and for Object-Z specifications in [LM05c].

Besides syntax-based mutation, semantic mutation is proposed in [CDH13] and demonstrated on C as well as on statecharts. Semantic mutation changes the interpretation of language constructs, for example the interpretation of integer division, and thereby models wrong assumptions of their effects.

2.3.4 Deployment of Mutants for Models

Mutants are applied in different scenarios in model-based testing. Roughly, these scenarios can be categorized into approaches for **testing of implementations** and for **testing of specifications**.

The most widespread use of mutants in models is to **test implementations** of models. The classic story within this cluster of approaches is to mutate a given model of some system under test and to generate test cases that reveal the difference between the mutant and the model, which can later be used to verify that a system under test implements the model rather than the mutant. Approaches following this story can be further classified by the test case generation method applied. Tests can be extracted by exploring and comparing the state space of the model and mutant [KPLV+03, BBTF11, BBTF12, ABJ+15a, ABJ+15b], by casting mutation killing as an optimization problem [HPT14, MFV15, MFV16, SMC+17], by solving a formal representation of mutation killing via symbolic tools, such as model checkers or SMT solvers, [ABM98, GH99, BOY00, BOY01, OBY03, FW08, XeAXW12, ALN13, AJT14, AGV15], by applying model specific algorithms, such as state identification [EFDYB12], state machine equivalence checking [HM07, HM09] or a tree construction [ZSL+14], by representing mutation killing as a refinement problem and using dedicated refinement checking algorithms [ABJK11, AL15, AHL+17], by casting mutation killing as an optimal strategy in a suitable game [LLNN17], or by a combination of these methods [AAJ+14]. This work also follows this classic story and the presented test case generation methods fall under the exploration-based as well as the symbolic category. Furthermore, this story is sometimes extended by automatically

extracting the model from the system under test and executing the generated tests right away [ZDK07, BBTF11, SMC+17].

Besides this classic story, mutations have been used in combination with pre-existing test suites by applying mutations to the test suite itself [LO01, XOL05, ZDK07], as a fuzzing method to test network protocols [ZWT12], to improve the test suite of model transformers [AME+15], to aid debugging timed automata models [AHL14a], to test the semantics of W3C XML Schemas [LM05b], and to remove conformance faults from feature models [AGV16].

Besides testing implementations, mutants of models have been used to **test specifications**. In contrast to deriving test cases from a model that ought to be applied to some implementation of it, the derived test cases are used to validate or debug the model itself. These approaches can be viewed as classic program-based mutation testing applied to models. The purpose can be further classified into validating models, i.e. assuring that the model captures the intended behavior, [FMMD99, SdSFLdSM00, SCSP03, SMW04, LM05c, SWZK17, Tra17], verifying models, i.e. assuring that the model is error free, [Woo93, AGR15, PRWN16, SWZK17], assessing the quality of existing test suites, for example by calculating its mutation score directly on the model, [BFP08, HPP+13a, HPLT14, BBH+16], and detecting model clones [RC09, SASC13].

2.3.5 Semantics of Strong Mutation Killing

In the classic sense, a mutant is strongly killed if for some inputs, the output of the mutant is different from the output of the non-mutated program (or model) [DLS78]. However, this definition of killing is not directly applicable to model-based mutation testing of reactive systems, since inputs and outputs come in sequences for such models, non-determinism can result in multiple possible outputs, and the system under test is generally different to the model at hand. Thus, the notion of killing has to be adapted for this setting. The approaches to defining strong mutation killing for model-based testing can roughly be classified into three categories.

Firstly, the **basic definition** is that a mutant is strongly killed by a test if the response (typically in terms of output) of the model to the test is different from the response of a mutant. The majority of model-based mutation testing approaches use this definition and thus we do not cite any specific instance. Often this simple definition is used for deterministic and non-reactive settings.

Secondly, the **property-based definition** states that a mutant is strongly killed upon violating a certain set of properties. These properties are commonly called trap properties [BOY01, FW08, XeAXW12] and encode a potential, noticeable defect of the mutant. Similarly, killing of SMV mutants is defined in [AGR15] via a set of 10 model properties, such as every assignment condition being satisfiable, or no property being satisfied vacuously. A mutant is killed if it violates more of these properties than the original model. The property-based definition is often used in combination with symbolic methods, such as model checking-based test case generation, for which property satisfaction is naturally verified.

Thirdly, the **conformance-based definition** states that a mutant is strongly killed if it does not conform to the model. This definition borrows notions from conformance theory that studies

how to test whether some implementation conforms to a specification. In the mutation testing setting, the mutant takes the role of the implementation, whereas the model takes the role of the specification. The resulting criteria are asymmetric, since conformant implementations are not required to implement every specified behavior. Conformance relations are used as the basis for strong mutation testing for the Unifying Theory of Programming [AH09], for timed automata [ALN13, LLNN17], for action systems [AJT14, Jöb14, AL15], for input-output symbolic transition systems [ZSL+14], and for UML State machines [ABJ+15a]. A slightly less formal conformance-based approach is presented in [BPGQ02, BPG07], where mutation killing is defined for communicating extended finite state machines respectively Kripke structures with inputs and outputs by comparing some or all possible non-deterministic input-output sequences following a sequence of inputs between a model and a mutant of it. In Chapter 4, we propose notions of strong killing that can be viewed as a combination of these two types of conformance-based approaches, importing the asymmetry from conformance relations as well as considering different killing types due to variation in non-deterministic outcomes.

2.3.6 Preceding Line of Research

This work is largely a continuation of efforts around the model-based mutation tool MoMuT::UML. The general approach of MoMuT::UML is described in [ABJ+15b] and applied on an industrial use-case in [AAJ+14]. The tool leverages an automatic translation of UML state machine models to action systems [KSA09]. The methods described in [ABJ+15b] as well as in Chapter 6 and Chapter 7 are state space exploration-based methods. In contrast, the methods described in [AJT14, Jöb14], which solves model-based mutation testing via a symbolic encoding of refinement relations for action systems, as well as in Chapter 5 complement these techniques with symbolic methods. Both exploration-based and symbolic methods have their pros and cons, where the general trade-off is between rigor and scalability. A systematic combination of both techniques is a very promising line of future work. MoMuT::UML has two sister model-based mutation testing tools MoMuT::REQS [AHL+14b, AHL+15, AHL+17] for assume-guarantee contracts and MoMuT::TA [ALN13, AHL14a] for timed automata.

CHAPTER 3

Model-based Testing

The word "model" is heavily overloaded and its meaning ranges from a person advertising some product, over a set of differential equations, to discrete transition systems. Even for the intended meaning within this work, which is the latter, there are endless ways to express such systems.

Thus, in this chapter, we want to achieve two main objectives. Firstly, we fix the semantics of models and model-based testing. We do this on a rather abstract type of system: Mealy machines. This formalism is well suited to describe the type of systems we are interested in, is widely known, and is very universal in the sense that many real life systems can be understood as Mealy machines. However, the formalism is not well suited to reasonably express any system of considerable size. In fact, at the end of this chapter, we introduce a simple example model whose full Mealy machine representation is too large for reasonably displaying it in this work.

Thus, secondly, we present two formalisms that can express Mealy machines and that are situated one abstraction layer above Mealy machines. We show how to translate both formalisms to Mealy machines and thereby supply them with model-based testing semantics. The first formalism are symbolic transition systems, which describe transition systems via formulas. This formalism is well suited for formal processing via logic based methods, such as model checking. While symbolic transition systems open the door for rigorous methods and precise results, they classically suffer from their own set of limitations. The language features are limited in order for the resulting model to still be processable by formal methods. For example, lists are widely used modeling feature, but pose quite a challenge for any logic solver. Perhaps an even more severe limitation is that writing logic formulas is not intuitive to many practitioners.

Therefore, we introduce action systems, which describe transition systems via variables and guarded actions that transform the values of these variables. This paradigm can be understood as a simplified programming language and is thus more intuitive than symbolic transition systems to many engineers. Nevertheless, action systems are powerful enough so that other modeling formalisms, such as UML state machines, can automatically be translated to them.

3.1 Modeling Reactive Systems

We study model-based testing of finite state reactive systems, which we understand as systems that continuously process inputs from the environment and emit outputs to the environment. Such systems can be abstractly characterized by Mealy machines, which we introduce in the following. Note that reactive systems can equivalently be characterized by other formalisms, such as (input-output-) labeled graphs. In this work, we chose finite non-deterministic Mealy machine as the basis of model-based testing semantics, since the formalism is well known and it is free of semantic assumptions that are not necessary for the methods presented in this work.

Definition 3.1.1 (Mealy machine)**.** A *Mealy machine* is a tuple $\mathcal{M} = \langle S, S_\iota, \Sigma, \Lambda, \delta \rangle$, where S is a finite set of states, $S_\iota \subseteq S$ is a set of initial states, Σ is the set of input symbols, Λ is the set of output symbols, and $\delta \subseteq S \times \Sigma \times \Lambda \times S$ is the transition relation.

We assume the existence of an input $\eta \in \Sigma$ and an output $\eta \in \Lambda$ that represent absence of input and output respectively. We say that s' is a successor of v with output o after input i, if $(s, i, o, s') \in \delta$ and denote that fact by $s \xrightarrow{i|o} s'$. A Mealy machine is *deterministic* if and only if there is a single initial state and for each state s and input i, there is at most one pair of successor state s' and output o, such that $s \xrightarrow{i|o} s'$. Otherwise, it is *non-deterministic*. Note that the use of a set of initial states is non-standard in Mealy machines, but improves readability throughout this work. A Mealy machine with multiple initial states can always be transformed to a Mealy machine with a single initial state that has transitions to the original initial states. The two machines behave equivalently up to the initial transition, which, if needed, can be labeled with an empty letter in order to preserve the language of the Mealy machine.

A Mealy machine is *input-enabled* if for every state $s \in S$ and input $i \in \Sigma$, there is a pair of output o and successor state s', such that $s \xrightarrow{i|o} s'$. Input-enabledness can easily be achieved by introduction self-loops for missing inputs with η outputs, which we assume always to be given implicitly throughout this work. This requirement simplifies the presentation of our methods, but it is also a natural requirement to systems, since it is not clear what it should mean not to accept an input. For example, on a keyboard, even if it is not connected to a computer, one can always press buttons. From hereon, we will refer to input-enabled Mealy machines as models.

Definition 3.1.2 (Trace)**.** A *trace* of M is a (potentially infinite) sequences of state, input, and output tuples $\langle (s_1, i_1, o_1), (s_2, i_2, o_2), \ldots \rangle$, such that $s_1 \in S_\iota$, for every $j \in \mathbb{N}$ smaller than the length of the sequence $s_j \xrightarrow{i_j|o_j} s_{j+1}$ and for a finite trace of length l $s_l \xrightarrow{i_l|o_l} s'$ for some state s'.

Given a trace $p = \langle (s_1, i_1, o_1), (s_2, i_2, o_2), \ldots \rangle$, we write $p[j]$ for the tuple (s_j, i_j, o_j), $p[j, l]$ for the sub-sequence $\langle (s_j, i_j, o_j), \ldots, (s_l, i_l, o_l) \rangle$, $p[j, \infty]$ for the suffix $\langle (s_j, i_j, o_j), \ldots \rangle$, and $p|_I, p|_O, p|_S, p|_{I/O}$ for the restricted sequences $\langle i_0, i_1, i_2 \ldots \rangle$, $\langle o_0, o_1, o_2 \ldots \rangle$, $\langle s_0, s_1, s_2 \ldots \rangle$, and $\langle (i_0, o_0), (i_1, o_1), (i_2, o_2) \ldots \rangle$ respectively. We lift restriction to sets of traces T by defining $T|_Q$ as $\{p|_Q \mid t \in T\}$ for $Q \in \{I, O, S, I/O\}$.

We now present the running example we will use throughout this work.

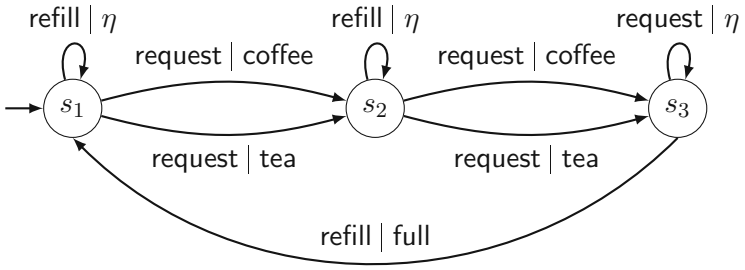


Figure 3.1: A model of a beverage machine.

Example 3.1.1. Consider a model of a beverage machine depicted in Figure 3.1, which non-deterministically serves coffee or tea upon a request, assuming that there is still enough water in its water tank. An empty water tank can be refilled to its initial capacity, which is enough for two beverages. The model has three states, corresponding to the beverage capacity of the water tank $2, 1, 0$ and one initial state s_1. Transitions are labeled input | output. Self-loops with absent input and absent output $\eta \mid \eta$ are not depicted. The multi-edges for the input request make the model non-deterministic. The following sequence is an example trace: $\langle(s_1, \mathsf{request}, \mathsf{coffee}), (s_2, \mathsf{request}, \mathsf{tea}), (s_3, \mathsf{refill}, \mathsf{full}), (s_1, \mathsf{request}, \mathsf{tea})\rangle$.

3.2 Testing Semantics

The simplest definition of a test for a reactive system is a sequence of inputs and outputs, which we call a linear test. The execution of a linear test on a system under test *passes*, if upon supplying the system with the sequence of inputs of the test (where supplying input η means to wait for output), the sequence of outputs of the test is observed. Otherwise, the execution of the test *fails*. Note that in the literature, this mode of assigning test verdicts is sometimes referred to as a positive test, in contrast to negative tests that pass if the witnessed outputs deviate from the test's outputs. In this work, we only consider positive tests. Formally, linear tests are defined as follows:

Definition 3.2.1 (Linear test)**.** A *linear test* t of model $\mathcal{M}$ with length n is a sequences of inputs and outputs $\langle(i_1, o_1), \ldots, (i_n, o_n)\rangle$, such that there is a trace $p \in \mathcal{T}_{rc}(\mathcal{M})$ with $p|_{I/O}[1, n] = t$. We denote by $\mathcal{T}_{st}(\mathcal{M})$ the set of all *tests* of $\mathcal{M}$.

Linear tests can be insufficient for testing non-deterministic systems. A conformant implementation of a non-deterministic model may resolve some non-deterministic choice of the model in a different order to a given linear test. As a result, the implementation delivers an output that is different from the output of the test and the test fails, even though the delivered output is allowed by the model. To remedy this situation, tests can be extended with information on multiple non-deterministic outcomes. This can either be done by extending a linear test to a fully adaptive tree that branches out in every non-deterministic choice, or by adding sets of allowed outputs to the test. We discuss here the latter variant.

A partially adaptive test is a sequence of inputs, outputs, and set of allowed outputs. The sets of allowed outputs enumerate all outputs that can follow after the test prefix up to the respective test step. The execution of a partially adaptive test on a system under test *passes* if its sequence of inputs triggers its exact sequence of outputs. The execution is *inconclusive* as soon as an allowed output is given by the system under test that is different to the test's output. The execution *fails* as soon as an output that is not allowed is given by the system under test.

Definition 3.2.2 (Partially adaptive test)**.** A *partially adaptive test* of $\mathcal{M}$ with length n is a sequence of inputs, outputs, and sets of outputs $\langle(i_1, o_1, O_1), \ldots, (i_n, o_n, O_n)\rangle$, such that its sequence of inputs and outputs $\langle(i_1, o_1), \ldots, (i_n, o_n)\rangle$ is a linear test and for every $j = 1, \ldots, n$ and allowed output $a \in O_j$ if and only if there is a trace $p \in \mathcal{T}_{rc}(\mathcal{M})$ that has equal inputs and outputs to p up to step $j - 1$ and then produces output a after input i_j, i.e. $p = \langle(s_1, i_1, o_1), \ldots, (s_j, i_j, a), \ldots\rangle$.

In practice, producing partially adaptive tests is expensive, since being an allowed output is a system global property due to non-deterministic transitions. That is, in order to produce correct partially adaptive tests, all paths of the model corresponding to the sequence of inputs and outputs need to be taken into account. Therefore, we weaken the definition of allowed outputs. A locally adaptive test is a partially adaptive test that only lists the allowed outputs after one specific model state resulting from executing the prefix of inputs. The production of locally adaptive tests only requires looking ahead for allowed outputs from a current state during test production, which makes it much more feasible in practice. The price of the weaker form of tests is that some test executions might fail which should actually be inconclusive.

Definition 3.2.3 (Locally adaptive test)**.** A *locally adaptive test* of $\mathcal{M}$ with length n is a sequence of inputs, outputs, and sets of outputs $\langle(i_1, o_1, O_1), \ldots, (i_n, o_n, O_n)\rangle$, such that the sequence of inputs and output $\langle(i_1, o_1), \ldots, (i_n, o_n)\rangle$ is a linear test and there is a trace $p \in \mathcal{T}_{rc}(\mathcal{M})$ with $p = \langle(s_1, i_1, o_1), (s_2, i_2, o_2), \ldots\rangle$ such that for every $j = 1, \ldots, n$ and allowed output $a \in O_j$ if and only if there is a trace $q \in \mathcal{T}_{rc}(\mathcal{M})$ that is equal to p up to step $j-1$ and then produces output a after input i_j, i.e. $q = \langle(s_1, i_1, o_1), (s_2, i_2, o_2), \ldots, (s_j, i_j, a), \ldots\rangle$.

Definition 3.2.4 (Test suite)**.** A *test suite* is a set of tests.

Example 3.2.1. Consider again the model presented in Example 3.1.1. An example of a linear test for the model is $\langle(\mathsf{request}, \mathsf{coffee}), (\mathsf{request}, \mathsf{tea}), (\mathsf{refill}, \mathsf{full}), (\mathsf{request}, \mathsf{tea})\rangle$. The corresponding partially adaptive test is $\langle(\mathsf{request}, \mathsf{coffee}, \{\mathsf{coffee}, \mathsf{tea}\}), (\mathsf{request}, \mathsf{tea}, \{\mathsf{coffee}, \mathsf{tea}\}), (\mathsf{refill}, \mathsf{full}, \{\mathsf{full}\}), (\mathsf{request}, \mathsf{tea}, \{\mathsf{coffee}, \mathsf{tea}\})\rangle$.

For the example, every locally adaptive test is also partially adaptive. However, consider the version of the model presented in Figure 3.2. This version initially flips a coin (modeled by two initial states). Depending on the outcome of the coin flip, the beverage machine works as the original one by serving either coffee or tea, or it serves only tea. Both the linear- as well as the partially adaptive test given above are linear respectively partially adaptive tests for this version. However, the following sequence is a locally adaptive test for this version, but not the original one: $\langle(\mathsf{request}, \mathsf{tea}, \{\mathsf{tea}\}), (\mathsf{request}, \mathsf{tea}, \{\mathsf{tea}\}), (\mathsf{refill}, \mathsf{full}, \{\mathsf{full}\}), (\mathsf{request}, \mathsf{tea}, \{\mathsf{tea}\})\rangle$. The locally adaptive tests corresponds to a trace following the part of the model only serving tea.

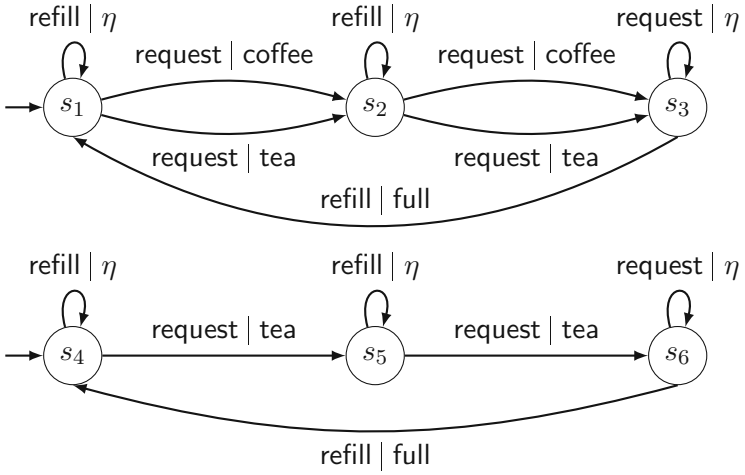


Figure 3.2: A model of a beverage machine with initial non-determinism.

3.3 Symbolic Transition System

A symbolic representation of models is useful when model-based testing is embedded into logic, as we will do later in this work. Therefore, we introduce symbolic transition systems as a symbolic representation of Mealy machines.

A symbolic transition system (STS) is a tuple $\mathcal{S} = \langle \mathcal{I}, \mathcal{O}, \mathcal{X}, \alpha, \delta \rangle$, where $\mathcal{I}, \mathcal{O}, \mathcal{X}$ are finite sets of input, output, and state variables, α is a formula over $\mathcal{X}$ (the initial state predicate), and δ is a formula over $\mathcal{I} \cup \mathcal{O} \cup \mathcal{X} \cup \mathcal{X}'$ (the transition relation predicate), where $\mathcal{X}' = \{x' \mid x \in \mathcal{X}\}$ is a set of primed variables representing the successor states. An input I, output O, state X, and successor state X' of $\mathcal{S}$, respectively, is a mapping of $\mathcal{I}$,$\mathcal{O}$, $\mathcal{X}$, and $\mathcal{X}'$, respectively, to values in a range that includes the elements $\top$ and $\bot$ (representing true and false, respectively). A tuple (X, I, O) of input I, output O and state X is called a *system state*. The set of all system states of $\mathcal{S}$ is denoted by $\mathcal{Y}^{\mathcal{S}}$. $Y|_{\mathcal{V}}$ denotes the restriction of the domain of mapping Y to the variables $\mathcal{V}$. Given a valuation Y and a Boolean variable $v \in \mathcal{V}$, $Y(v)$ denotes the value of v in Y (if defined) and $Y[v \mapsto a]$ denotes Y with variable v set to value a.

We assume that the initial state- and transition relation predicate are defined in a logic that includes standard Boolean operators *not* $\neg$, *and* $\wedge$, *or* $\vee$, *implication* $\rightarrow$, and *equivalence* $\leftrightarrow$. We omit further details, as our results do not depend on a specific formalism. We write $X \vDash \alpha$ and $X, I, O, X' \vDash \delta$ to denote that α and δ evaluate to true under an evaluation of inputs I, outputs O, states X, and successor states X'. We assume that $\vDash$ interprets the standard Boolean operators using the standard semantics, i.e. $X, I, O, X' \vDash \neg\delta$ if and only if it is not the case that $X, I, O, X' \vDash \delta$, $X, I, O, X' \vDash \delta_0 \wedge \delta_1$ if and only if $X, I, O, X' \vDash \delta_0$ and $X, I, O, X' \vDash \delta_1$, $X, I, O, X' \vDash \delta_0 \vee \delta_1$ if and only if $X, I, O, X' \vDash \delta_0$ or $X, I, O, X' \vDash \delta_1$, $X, I, O, X' \vDash \delta_0 \rightarrow \delta_1$ if and only if $X, I, O, X' \vDash \delta_0$ then $X, I, O, X' \vDash \delta_1$, and $X, I, O, X' \vDash \delta_0 \leftrightarrow \delta_1$ if and only if $X, I, O, X' \vDash \delta_0$ if and only if $X, I, O, X' \vDash \delta_1$. We assume that every STS has a distinct input I_η and a distinct output O_η representing absence of input and output respectively. A state X such that $X \vDash \alpha$ is an *initial state*. A state X has a transition with input I to its *successor state*

X' with output O iff $X, I, O, X' \vDash \delta$. We write $X \xrightarrow{I|O} X'$ if X' is the successor state of X with input I and output O, or X does not have any successor for input I, $O = O_\eta$, and X' equals X when every variable x is replaced by x'. Thus, we assume that symbolic transition systems are always implicitly input-enabled.

A *symbolic trace* of $\mathcal{S}$ is a (potentially infinite) sequence of states, input, and outputs of $\mathcal{S}$ $\langle (X_1, I_1, O_1), (X_2, I_2, O_2), \ldots \rangle$ such that $X_1 \vDash \alpha$ and for every $j \in \mathbb{N} : X_j \xrightarrow{I_j|O_j} X_{j+1}$. We denote by $\mathcal{ST}_{rc}(\mathcal{S})$ the set of all *symbolic traces* of $\mathcal{S}$. We extend domain restriction to symbolic traces. To this end, given a symbolic trace $p \in \mathcal{ST}_{rc}(\mathcal{S})$ and set of variables $\mathcal{V}$, we define the restriction of p to $\mathcal{V}$, denoted via $p|_\mathcal{V}$, as the sequence of restricted valuations $\langle (X_1|_\mathcal{V}, I_1|_\mathcal{V}, O_1|_\mathcal{V}), (X_2|_\mathcal{V}, I_2|_\mathcal{V}, O_2|_\mathcal{V}), \ldots \rangle$ and lift restriction to sets of symbolic traces T by defining $T|_\mathcal{V}$ as $\{p|_\mathcal{V} \mid t \in T\}$. Finally, we denote by $\mathcal{ST}_{st}(\mathcal{S}) \stackrel{\text{def}}{=} \mathcal{ST}_{rc}(\mathcal{S})|_{\mathcal{I} \cup \mathcal{O}}$ the set of all symbolic tests of $\mathcal{S}$.

Every symbolic transition system $\mathcal{S}$ induces a Mealy machine $\mathcal{M}^\mathcal{S} \stackrel{\text{def}}{=} \langle S^\mathcal{S}, S^\mathcal{S}_\iota, \Sigma^\mathcal{S}, \Lambda^\mathcal{S}, \delta^\mathcal{S} \rangle$, where for each state X, input I, and output O, there is distinct Mealy machine state s_X, input symbol i_I, and output symbol o_O and $S^\mathcal{S} = \{s_X \mid X \text{ is a state of } \mathcal{S}\}$, $S^\mathcal{S}_\iota = \{s_X \mid X \vDash \alpha\}$, $\Sigma^\mathcal{S} = \{i_I \mid I \text{ is an input of } \mathcal{S}\}$, $\Lambda^\mathcal{S} = \{o_O \mid O \text{ is an output of } \mathcal{S}\}$, and $\delta^\mathcal{S} = \{(s_{X_1}, i_I, o_O, s_{X_2}) \mid X_1, I, O, X_2' \vDash \delta\}$.

Definition 3.3.1. Let $p = \langle (X_1, I_1, O_1), (I_2, O_2, X_2), \ldots \rangle$ be a symbolic trace and let $t = \langle (I_1, O_1), (I_2, O_2), \ldots \rangle$ be a symbolic test. The *Mealy machine correspondence* of p and t are defined as $p^\mathcal{M} \stackrel{\text{def}}{=} \langle (s_{X_1}, i_{I_1}, o_{O_1}), (s_{X_2}, i_{I_2}, o_{O_2}), \ldots \rangle$ and $t^\mathcal{M} \stackrel{\text{def}}{=} \langle (i_{I_1}, o_{O_1}), (i_{I_2}, o_{O_2}), \ldots \rangle$.

Lemma 3.3.1 (Symbolic trace and test correspondence)**.**

$$p \in \mathcal{ST}_{rc}(\mathcal{S}) \text{ if and only if } p^\mathcal{M} \in \mathcal{T}_{rc}(\mathcal{M}^\mathcal{S})$$
$$t \in \mathcal{ST}_{st}(\mathcal{S}) \text{ if and only if } t^\mathcal{M} \in \mathcal{T}_{st}(\mathcal{M}^\mathcal{S})$$

Proof. The lemma follows directly from the definition of symbolic traces, symbolic tests, and $\mathcal{M}^\mathcal{S}$. □

Finally, we say that a symbolic transition system $\mathcal{S}$ is deterministic if and only if its induced Mealy machine $\mathcal{M}^\mathcal{S}$ is deterministic.

Example 3.3.1. In this example, we show how the running example presented in Figure 3.1 can be represented as a symbolic transition system. The system has a single input variable `in` with range $\{\eta, \texttt{request}, \texttt{refill}\}$, a single output variable `out` with range $\{\eta, \texttt{tea}, \texttt{coff}\}$, and a single state variable `water` with range $\{0, 1, 2\}$. For ease of presentation, in the following we express the initial state and transition predicate using equality and integer arithmetic, but note that they could easily be expressed using simple propositional logic as well.

$$\alpha \stackrel{\text{def}}{=} \texttt{water=2}$$

$$\begin{aligned}\delta \stackrel{\text{def}}{=} &(\texttt{in=request} \wedge \texttt{water>0} \wedge \texttt{out=coff} \wedge \texttt{water'=water-1}) \vee \\ &(\texttt{in=request} \wedge \texttt{water>0} \wedge \texttt{out=tea} \wedge \texttt{water'=water-1}) \vee \\ &(\texttt{in=refill} \wedge \texttt{water=0} \wedge \texttt{out=full} \wedge \texttt{water'=2}) \vee \\ &(\texttt{in=request} \wedge \texttt{water=0} \wedge \texttt{out=}\eta \wedge \texttt{water'=water})\end{aligned}$$

3.4 Action Systems

In order to enable modeling of complex systems, powerful modeling languages are necessary. While in theory many systems can be represented as Mealy machines or symbolic transition systems, in practice these formalisms can be limiting. We now present a versatile modeling language that is geared towards expressing a large variety of models of significant size.

3.4.1 Syntax

An *action system* is a tuple $\mathcal{A} = \langle \mathcal{V}, s_\iota, Act, A_\iota \rangle$, where $\mathcal{V}$ is a finite set of typed variables, s_ι is an initial state, Act is a set of actions, and A_ι is the main action. States of action system are mappings of variables to values of their type.

Variable Types

Types of variables $\mathcal{V}$ can be **Boolean**, **enumeration**$(\{e_1, \ldots, e_n\})$, **integer**(n, m), or **list**(T, m). An enumeration type is a finite set of unique values $e_1, \ldots, e_n$. An integer type represents the interval $[n, m]$ and a list type represents all lists of maximum length m, the elements of which are of type T (lists can contain lists). Boolean is a specific enumeration type with the values $\{\mathsf{true}, \mathsf{false}\}$ and their usual semantics. All types are finite.

Actions

An action is either the composition of two other actions, a guarded command with a guard that is a Boolean expression over $\mathcal{V}$, an assignment of a variable to an expression over $\mathcal{V}$ of the same types as the variable, **skip**, or **abort**. Composed actions are formed via sequential $A_1;\ A_2$, non-deterministic $A_1 \ [] \ A_2$, or $A_1 \ // \ A_2$ prioritized composition. In the latter case, we say that every action in A_1 is *prioritized over* every action in A_2.

Each guarded command and each assignment is labeled with a unique label ℓ. Every labeled action is either **observable**, **controllable**, or **internal**. We call observable and controllable labels *visible* and sometimes refer to observable/controllable action labels simply as observables and controllables. We use the convention that observable labels start with the prefix **obs**, controllable labels start with the prefix **ctr**, and all other labels are internal. The differentiation between these kinds of labels is an additional modeling tool, indicating whether an action models an input to

Expression	Explanation	Expression	Explanation
$\mathcal{E}^B$ **Boolean expressions**		$\mathcal{E}^I$ **integer expressions**	
v	Boolean variable	v	Integer variable
$\mathsf{false}, \mathsf{true}$	Constant	k	Constant
$\neg\mathcal{E}^B$	Boolean negation	$\mathcal{E}_1^I \oplus \mathcal{E}_2^I$	$\oplus \in \{+, -, *, /, \mathsf{mod}\}$
$\mathcal{E}_1^B \oplus \mathcal{E}_2^B$	$\oplus \in \{=, \neq, \vee, \wedge, \Rightarrow, \Leftrightarrow\}$	$\mathsf{len}(\mathcal{E}^{L(X)})$	Length of list
$\mathcal{E}_1^I \oplus \mathcal{E}_2^I$	$\oplus \in \{=, \neq, <, >, \leq, \geq\}$	$\oplus\mathcal{E}^I$	$\oplus \in \{-, \mathsf{abs}\}$
$\mathcal{E}_1^E \oplus \mathcal{E}_2^E$	$\oplus \in \{=, \neq\}$	$\mathcal{E}^{L(X)}$ **list expressions**	
$\mathcal{E}_1^{L(X)} \oplus \mathcal{E}_2^{L(X)}$	$\oplus \in \{=, \neq\}$	v	List variable
$\forall\, v : X.\, \mathcal{E}^B$	Universal quantification	nil	Empty list
$\exists\, v : X.\, \mathcal{E}^B$	Existential quantification	$[\mathcal{E}_1^X, \mathcal{E}_2^X, \ldots, \mathcal{E}_n^X]$	Constant
$\mathcal{E}^X \in \mathcal{E}^{L(X)}$	Element of	$\mathsf{tl}(\mathcal{E}^{L(X)})$	Tail
$\mathcal{E}^X \notin \mathcal{E}^{L(X)}$	Not element of	$\mathcal{E}_1^{L(X)} \circ \mathcal{E}_2^{L(X)}$	Concatenation
$\mathcal{E}^E$ **enumeration expressions**		$\mathcal{E}^X$ **Generic expressions**	
v	Enumerate variable	$\mathsf{hd}(\mathcal{E}^{L(X)})$	Head
e_1	Constant	$\mathsf{fold}(\mathcal{E}_1^{L(Y)}, \mathcal{E}_2^X, f : \mathcal{E}^Y \times \mathcal{E}^X \to \mathcal{E}^X)$	Fold
		$\mathsf{ite}(\mathcal{E}^B, \mathcal{E}^X, \mathcal{E}^X)$	if-then-else

Table 3.1: The syntax of action system expressions.

a system, i.e. is controllable, models an output of the system, i.e. is observable, or models a transition occurring inside the system that can not be observed, i.e. is internal. Since labels are unique, we use the label and the action it labels interchangeably.

The distinct action A_ι composes actions in Act and is the entry point of the control structure of $\mathcal{A}$. This action is supposed to be executed indefinitely as long as no **abort** action is executed.

Expressions

In the following, we summarize the syntax of expressions of action systems. We use $\mathcal{E}^B$, $\mathcal{E}^I$, $\mathcal{E}^{L(X)}$, and $\mathcal{E}^X$ to denote expressions resulting in a value of type **Boolean**, **integer**, **list** over X, and the type of X, respectively. Note that some operators can be applied to both integer and enumeration types. The resulting type is always the type of the variable the result is assigned to. The placeholders X and Y denote arbitrary types.

Fold takes as third argument a function f that itself takes as arguments an expression of type $\mathcal{E}^Y$ as well as an expression of type $\mathcal{E}^X$ and returns an expression of type $\mathcal{E}^X$. The semantics of fold are to apply f to each element in $\mathcal{E}_1^{L(Y)}$ (first argument) and pass as second argument the result of the previous evaluation of f. The result of fold is the last evaluation of f.

Example 3.4.1. Figure 3.3 shows the action system representation of the beverage machine presented in Figure 3.1. The system has a single variable water with values $\{0, 1, 2\}$. The controllable action `ctr request` composes actions to serve coffee or tea sequentially with the

internal action decrement, which decrements the value of the water by one. The observable actions **obs** coffee and **obs** tea are guarded by the value of water being greater than zero and do not have an effect on the state, thus the bodies of the actions simply are **skip**. Moreover, the controllable action **ctr** fill, which is guarded by a zero value of the water variable, sequentially composes internal action reset, which resets the value of variable water to 2 with the guarded command **obs** fill, which has a trivial guard and body and thus is only relevant due to its observable label. Finally, both controllable actions **ctr** request and **ctr** fill compose their main actions with an observable action **obs** η that is enabled if and only if no path through the main part of the controllable is enabled. Thus, the action system is input-enabled.

The main action A_ι composes **ctr** request and **ctr** fill non-deterministically. Since the examples does not contain any **abort** actions, the modeled system runs indefinitely, reacting to beverage requests and requests to fill the water tank.

$$
\begin{aligned}
&\mathcal{V} = \{\mathsf{water} : \mathbf{integer}(0,2)\}\\
&s_\iota = [\mathsf{water} \mapsto 2]\\
&Act = \{\\
&\quad \mathbf{ctr}\ \mathsf{request} : ((\mathsf{coffee}\ [\!]\ \mathsf{tea});\ \mathsf{decrement})\ /\!/\ \eta,\\
&\quad \mathbf{ctr}\ \mathsf{refill} : (\mathsf{water} = 0 \triangleright \mathsf{reset};\ \mathsf{full})\ /\!/\ \eta,\\
&\quad \mathbf{obs}\ \mathsf{coffee} : \mathsf{water} > 0 \triangleright \mathbf{skip},\\
&\quad \mathbf{obs}\ \mathsf{tea} : \mathsf{water} > 0 \triangleright \mathbf{skip},\\
&\quad \mathbf{obs}\ \mathsf{full} : \mathsf{true} \triangleright \mathbf{skip},\\
&\quad \mathbf{obs}\ \eta : \mathsf{true} \triangleright \mathbf{skip},\\
&\quad \mathsf{decrement} : \mathsf{water} \leftarrow \mathsf{water} - 1,\\
&\quad \mathsf{reset} : \mathsf{water} \leftarrow 2\\
&\}\\
&A_\iota = \mathsf{request}\ [\!]\ \mathsf{refill}
\end{aligned}
$$

Figure 3.3: An action system representation of the beverage machine running example.

Objects

We study models that are written in an object oriented extension of action systems. However, all objects need to be created statically during initialization. Therefore, for the purpose of this work, we can view objects simply as a partition of variables. For an object, the *object value* is a Cartesian product of the values of its variables, together with a unique identifier of its deriving class. A more detailed discussion of object orientation in action systems can be found in [BKS98].

3.4.2 Semantics

Small-step Semantics

The semantics of an action A are defined by the successor function $su(A, l, s)$. su accepts an action A, a sequence of labels l representing a path that is initially empty, and a state s. It returns a set of pairs, each pair (l', s') consists of a sequence of labels l' and a state s', such that l is a prefix of l'. The labels describe the guards and assignments that lead from s to s'. Table 3.2 provides the successor functions $su(A, l, s)$ of the different action types, where given a Boolean expression P and state s, we say that s satisfies P and write $s \vDash P$, if P evaluates to true under assignment s.

Action (A)	**Notation**	$\mathbf{su(A, l, s)}$
skip	**skip**	$\{(l, s)\}$
abort	**abort**	$\varnothing$
Assignment	$\ell : v \leftarrow e$	$\{(l{\cdot}\langle\ell\rangle, s[v \mapsto e])\}$
Guarded command	$\ell : g \triangleright A_1$	$(s \vDash g)\ ?\ su(A_1, l{\cdot}\langle\ell\rangle, s)\ :\ \varnothing$
Sequential Composition	$A_1; A_2$	$\bigcup_{(l',s') \in su(A_1,l,s)} (su(A_2, l', s'))$
Non-det Composition	$A_1 \ [\!] \ A_2$	$su(A_1, l, s) \cup su(A_2, l, s)$
Prioritized Composition	$A_1 \ // \ A_2$	$su(A_1, l, s) \neq \varnothing\ ?\ su(A_1, l, s)\ :\ su(A_2, l, s)$

Table 3.2: The action system successor state semantics.

We define $succ_{\mathcal{A}}(s) \stackrel{\text{def}}{=} \{s' \mid \exists l'.\ (l', s') \in su(\mathcal{A}, \mathsf{nil}, s)\}$ as the set of successor states of s and $succ_{\mathcal{A}}(s, l) \stackrel{\text{def}}{=} \{s' \mid (l, s') \in su(\mathcal{A}, \mathsf{nil}, s)\}$ as the set of successor states of s following the sequence of labels l. We use $path_{\mathcal{A}}(s)$ as an abbreviation for $su(\mathcal{A}, \mathsf{nil}, s)$. For a tuple $\pi \in path_{\mathcal{A}}(s)$ we use the notation $\pi.l$ to refer to its path and $\pi.s$ to refer to its state component. We call a state s' *reachable* if it is s_ι or $s' \in succ_{\mathcal{A}}(s)$ for some reachable state s and denote by $states_{\mathcal{A}}$ the set of reachable states. We call a labeled action with label ℓ *reachable*, if there exists a reachable state s, such that ℓ is contained in the sequence $\pi.l$ for some $\pi \in path_{\mathcal{A}}(s)$. We say that an assignment $\ell : v \leftarrow e$ *writes* variable v. Furthermore, we say that an assignment $\ell : v \leftarrow e$ respectively a guarded command $\ell : g \triangleright A_1$ *reads* all variables occurring in expressions e respectively g.

Example 3.4.2. Consider again the action system presented in Figure 3.3. Let us, for ease of presentation, abbreviate a state $[\mathsf{water} \mapsto x]$ simply by $[x]$ and use this notation throughout examples in this work. The action system has three reachable states $s_\iota = [2], [1], [0]$ with the

following successor paths:

$$
\begin{aligned}
path_{\mathcal{A}}([2]) = \{&(\langle \mathbf{ctr}\,\mathsf{request}, \mathbf{obs}\,\mathsf{coffee}, \mathsf{decrement}\rangle, [1]),\\
&(\langle \mathbf{ctr}\,\mathsf{request}, \mathbf{obs}\,\mathsf{tea}, \mathsf{decrement}\rangle, [1]),\\
&(\langle \mathbf{ctr}\,\mathsf{refill}, \mathbf{obs}\,\eta\rangle, [2])\}\\
path_{\mathcal{A}}([1]) = \{&(\langle \mathbf{ctr}\,\mathsf{request}, \mathbf{obs}\,\mathsf{coffee}, \mathsf{decrement}\rangle, [0]),\\
&(\langle \mathbf{ctr}\,\mathsf{request}, \mathbf{obs}\,\mathsf{tea}, \mathsf{decrement}\rangle, [0]),\\
&(\langle \mathbf{ctr}\,\mathsf{refill}, \mathbf{obs}\,\eta\rangle, [1])\}\\
path_{\mathcal{A}}([0]) = \{&(\langle \mathbf{ctr}\,\mathsf{refill}, \mathsf{reset}, \mathbf{obs}\,\mathsf{full}\rangle, [2]),\\
&(\langle \mathbf{ctr}\,\mathsf{request}, \mathbf{obs}\,\eta\rangle, [0])\}
\end{aligned}
$$

Finite Path Unrolling

The small state semantics provide the atomic unit of successor computation of action systems by specifying the effect of all enabled paths through the action composition tree. However, this granularity of successor paths is insufficient for our requirements during test case generation and mutation analysis.

For example, consider the two actions $\mathbf{ctr}\,\mathsf{i} : \mathsf{x} = 0 \triangleright \mathsf{x} \leftarrow 1$ and $\mathbf{obs}\,\mathsf{o} : \mathsf{x} = 1 \triangleright \mathsf{x} \leftarrow 0$ as well as an action system $\mathcal{A}_{[]}$ that composes the actions non-deterministically, i.e. $\mathsf{i}[]\mathsf{o}$, and an action system $\mathcal{A}_{;}$ that composes the actions sequentially, i,e. $\mathsf{i};\mathsf{o}$. Both action systems produce output o in response to input i and can thus be considered equivalent from an input-output sequence perspective. However, the successor path of state $[0]$ in $\mathcal{A}_{[]}$ is $path_{\mathcal{A}_{[]}}([0]) = \{(\langle \mathbf{ctr}\,\mathsf{i}\rangle, [1])\}$, while in $\mathcal{A}_{;}$ it is $path_{\mathcal{A}_{;}}([0]) = \{(\langle \mathbf{ctr}\,\mathsf{i}, \mathbf{obs}\,\mathsf{o}\rangle, [0])\}$. Thus, if we used the successor paths $path_{\mathcal{A}}(.)$ as a basis for comparing these two systems, we would spuriously conclude that the systems have different input-output behavior.

To remedy this situation, we unroll successor paths until every path can only be extended by a controllable action. In order not to unroll successor paths indefinitely, we require that action systems do not encode infinite paths of non-controllable actions. An action system $\mathcal{A}$ is *finitely responsive*, if there is no infinite sequence of paths $\pi_1, \pi_2, \ldots$, such that $\pi_1 \in path_{\mathcal{A}}(s_\iota)$ and for every $i > 0$ it is the case that $\pi_{i+1} \in path_{\mathcal{A}}(\pi_i.s)$ and $\pi_i.l$ contains only observable and internal labels. From hereon, if not specified otherwise, we require that action systems are finitely responsive. In practice, this requirement might be difficult to verify, but can be replaced by requiring bounded responsiveness, i.e. that all sequence of paths have a controllable label at least every n labels for some large $n \in \mathbb{N}$.

Next, we recursively define the set of *extended successor paths* that unroll successor paths until every path can only be extended by a controllable action. Formally, we define $extpath_{\mathcal{A}}(s)$ as a set of sequence labels $\vec{\ell} = \langle \ell_1, \ldots, \ell_n\rangle$ and state s pairs, such that $(\vec{\ell}, s') \in extpath_{\mathcal{A}}(s)$ if $(\vec{\ell}, s') \in path_{\mathcal{A}}(s)$ and every path in $path_{\mathcal{A}}(s')$ contains a controllable action, or there is an $m < n$ such that $(\langle \ell_1, \ldots, \ell_m\rangle, s'') \in path_{\mathcal{A}}(s)$, there is a path in $path_{\mathcal{A}}(s'')$ that does not

contain a controllable action label, and $(\ell_{m+1}, \ldots, \ell_n\rangle, s') \in extpath_{\mathcal{A}}(s'')$. Since we require that action systems are finitely responsive, this definition is well given.

Note that it can be the case that some state has a successor path that contains controllable actions as well as a successor path that does not contain controllable actions. We call such states *mixed states*, which are generally undesirable, because from a testing perspective, correctly operating in mixed states requires the ability of the tester to time inputs before some output occurs. However, such an assumption is problematic, since outputs are supposed to be under the control of the system under test. Therefore, with our definition of extended successor paths, we chose to let the system provide all its outputs before a new input can be supplied to the model.

Finally, we define the *visible successor paths* by projecting the extended successor paths to visible labels. Formally, we define $vispath_{\mathcal{A}}(s)$ as a set of sequence labels and state pairs, such that $(\langle \ell_{j_1}, \ldots, \ell_{j_m}\rangle, s') \in vispath_{\mathcal{A}}(s)$ if $(\langle \ell_1, \ldots, \ell_n\rangle, s') \in extpath_{\mathcal{A}}(s)$, $1 \leq j_1 < j_2 < \ldots < j_m \leq n$ and $\ell_{j_1}, \ldots, \ell_{j_m}$ are all visible labels within $\ell_1, \ldots, \ell_n$. We call a state s' *visibly reachable* if it is s_ι or $s' = \pi.s$ for some $\pi \in vispath_{\mathcal{A}}(s)$ and some visibly reachable state s. We denote by $visStates_{\mathcal{A}}$ the set of *visibly reachable states*.

3.4.3 Action System as Mealy Machine

We transform action systems to Mealy machines and thereby connect action systems with the testing semantics defined in this work. The essential idea of the transformation is to encode visible successor paths. Controllable action labels correspond to inputs, whereas observable action labels correspond to outputs.

Note that action system successor paths are mixed sequences of controllable and observable action labels, whereas Mealy machines represent inputs and outputs as separate components of transitions. In order to bridge this gap, let us define the *interleaved input-output sequence* of a sequence of action system labels. To this end, let us define helper functions $in(.)$ and $out(.)$ that map action system labels to inputs and outputs respectively. For a label ℓ, $in(\ell) \stackrel{\text{def}}{=} \ell$ if ℓ is controllable and $in(\ell) \stackrel{\text{def}}{=} \eta$ otherwise. Likewise, $out(\ell) \stackrel{\text{def}}{=} \ell$ if ℓ is observable and $out(\ell) \stackrel{\text{def}}{=} \eta$ otherwise respectively. Given a sequence of tuples of inputs and outputs its *immediate output sequence* is the same sequence, where recursively successive tuples (i, η) and (η, o) are replaced by the tuple (i, o). Finally, the *interleaved input-output sequence* $io(\vec{\ell})$ of a sequence of action labels $\vec{\ell} = \langle \ell_1, \ldots, \ell_n\rangle$ is defined as the immediate output sequence of input and output tuples $\langle (in(\ell_1), out(\ell_1)), \ldots, (in(\ell_n), out(\ell_n))\rangle$.

Let $\mathcal{A} = \langle \mathcal{V}, s_\iota, Act, A_\iota\rangle$ be a finitely responsive action system. Its corresponding Mealy machine $\mathcal{M}^{\mathcal{A}} \stackrel{\text{def}}{=} \langle S^{\mathcal{A}}, S^{\mathcal{A}}_\iota, \Sigma^{\mathcal{A}}, \Lambda^{\mathcal{A}}, \delta^{\mathcal{A}}\rangle$ is defined as follows:

Every visibly reachable state of the action system induces a state in the Mealy machine. In addition, we introduce states to the Mealy machine that correspond to intermediate steps of visible successor paths. Formally, every visibly reachable state $s \in visStates_{\mathcal{A}}$ induces the following set of Mealy machine states, where $s^{\mathcal{M}}, s^{\pi}_1, \ldots, s^{\pi}_{n-1} \in S^{\mathcal{A}}$ are fresh states:

$$S^{\mathcal{A}}_s \stackrel{\text{def}}{=} \{s^{\mathcal{M}}\} \cup \{s^{\pi}_1, \ldots, s^{\pi}_{n-1} \mid \pi \in vispath_{\mathcal{A}}(s), |io(\pi.l)| = n\}$$

The set of Mealy machine states is the union over induced states of all reachable states:

$$S^{\mathcal{A}} \stackrel{\text{def}}{=} \bigcup_{s \in visStates_{\mathcal{A}}} S^{\mathcal{A}}_s$$

The initial state of the Mealy machine is the singleton of the state representing the initial state of the action system, where $s^{\mathcal{M}}_\iota \in S^{\mathcal{A}}$ is a fresh state:

$$S^{\mathcal{A}}_\iota \stackrel{\text{def}}{=} \{s^{\mathcal{M}}_\iota\}$$

The sets Σ and Λ contain controllable and observable labels, respectively:

$$\Sigma^{\mathcal{A}} \stackrel{\text{def}}{=} \{\ell \mid \ell \text{ is a controllable action label of } \mathcal{A}\}$$

$$\Lambda^{\mathcal{A}} \stackrel{\text{def}}{=} \{\ell \mid \ell \text{ is an observable action label of } \mathcal{A}\}$$

The transition relation of $\mathcal{M}^{\mathcal{A}}$ corresponds to the interleaved input-output sequences of visible successor paths as defined above. To this end, let s be a visibly reachable state of $\mathcal{A}$. Its outgoing transitions in $\mathcal{M}^{\mathcal{A}}$ are defined as follows:

$$\delta^{\mathcal{A}}_s \stackrel{\text{def}}{=} \big\{\{(s^{\mathcal{M}}, i_1, o_1, s^{\pi}_1), \ldots, (s^{\pi}_{n-1}, i_n, o_n, (\pi.s)^{\mathcal{M}})\} \mid \\ \pi \in vispath_{\mathcal{A}}(s) \text{ and } io(\pi.l) = \langle (i_1, o_1), \ldots, (i_n, o_n)\rangle\big\}$$

The final transition relation is the union of outgoing transitions for all reachable states:

$$\delta^{\mathcal{A}} \stackrel{\text{def}}{=} \bigcup_{s \in visStates_{\mathcal{A}}} \delta^{\mathcal{A}}_s, \text{ where}$$

We define the set of tests respectively traces of $\mathcal{A}$ as $\mathcal{T}_{st}(\mathcal{A}) \stackrel{\text{def}}{=} \mathcal{T}_{st}(\mathcal{M}^{\mathcal{A}})$ and $\mathcal{T}_{rc}(\mathcal{A}) \stackrel{\text{def}}{=} \mathcal{T}_{rc}(\mathcal{M}^{\mathcal{A}})$ respectively. As demonstrated in Example 3.4.1, every action system can easily be made input-enabled by composing $\mathbf{obs}\,\eta$ prioritized below the effect of every controllable action. The resulting Mealy machine of such an action system is technically not input-enabled, since inputs are refused at intermediate states within sequences of outputs. However, the meaning of such intermediate states is that they can not be observed nor controlled from the outside. Thus, input at such states is not foreseen and input-enabledness up to intermediate states is sufficient for our purpose.

Example 3.4.3. In Figure 6.1, we present the Mealy machine corresponding to the action system presented in Example 3.4.1. Besides labeling inputs with a $\mathbf{ctr}$ and outputs with a $\mathbf{obs}$ prefix, the translated Mealy machine is equivalent to Figure 6.1, which is the intent of the translation.

In order to demonstrate the concepts used in the Mealy machine translation of action systems, we introduce a variant of the example in Figure 3.4. In contrast to managing the water level in the water tank, this model of a coffee machine describes the process of making the coffee, which is assumed to consist of adding together water, beans, and sugar.

For a simpler presentation, we abbreviate states by their values. For example, as a shorthand for the state $[\mathsf{r} \mapsto \mathsf{false}, \mathsf{b} \mapsto \mathsf{false}, \mathsf{s} \mapsto \mathsf{false}, \mathsf{w} \mapsto \mathsf{false}]$, we write $[\mathsf{false}, \mathsf{false}, \mathsf{false}, \mathsf{false}]$.

$$
\begin{aligned}
&\mathcal{V} = \{\mathsf{r} : \mathbf{Boolean}, \mathsf{b} : \mathbf{Boolean}, \mathsf{s} : \mathbf{Boolean}, \mathsf{w} : \mathbf{Boolean}\}\\
&s_\iota = [\mathsf{r} \mapsto \mathsf{false}, \mathsf{b} \mapsto \mathsf{false}, \mathsf{s} \mapsto \mathsf{false}, \mathsf{w} \mapsto \mathsf{false}]\\
&Act = \{\\
&\quad \mathbf{ctr}\ \mathsf{request} : (\neg \mathsf{r} \triangleright \mathsf{flip\_r}) \mathbin{/\!/} \eta,\\
&\quad \mathbf{obs}\ \mathsf{beans} : \mathsf{r} \wedge \neg \mathsf{b} \triangleright \mathsf{flip\_b},\\
&\quad \mathbf{obs}\ \mathsf{sugar} : \mathsf{r} \wedge \neg \mathsf{s} \triangleright \mathsf{flip\_s},\\
&\quad \mathbf{obs}\ \mathsf{water} : \mathsf{r} \wedge \neg \mathsf{w} \triangleright \mathsf{flip\_w},\\
&\quad \mathbf{obs}\ \mathsf{coffee} : \mathsf{r} \wedge \mathsf{b} \wedge \mathsf{s} \wedge \mathsf{w} \triangleright \mathsf{flip\_b};\ \mathsf{flip\_s};\ \mathsf{flip\_w};\ \mathsf{flip\_r},\\
&\quad \mathbf{obs}\ \eta : \mathsf{true} \triangleright \mathbf{skip},\\
&\quad \mathsf{flip\_r} : \mathsf{true} \triangleright \mathsf{r} \leftarrow \neg \mathsf{r},\\
&\quad \mathsf{flip\_b} : \mathsf{true} \triangleright \mathsf{b} \leftarrow \neg \mathsf{b},\\
&\quad \mathsf{flip\_s} : \mathsf{true} \triangleright \mathsf{s} \leftarrow \neg \mathsf{s},\\
&\quad \mathsf{flip\_w} : \mathsf{true} \triangleright \mathsf{w} \leftarrow \neg \mathsf{w}\\
&\}\\
&A_\iota = \mathsf{request} \ [\!]\ \mathsf{beans} \ [\!]\ \mathsf{water} \ [\!]\ \mathsf{sugar} \ [\!]\ \mathsf{coffee}
\end{aligned}
$$

Figure 3.4: An action system representing a coffee brewing machine.

The initial state has one successor path, that is

$$path_{\mathcal{A}}([\mathsf{false}, \mathsf{false}, \mathsf{false}, \mathsf{false}]) = \{(\langle \mathbf{ctr}\ \mathsf{request}, \mathsf{flip\_r} \rangle, [\mathsf{true}, \mathsf{false}, \mathsf{false}, \mathsf{false}])\}$$

The resulting state can be extended with non-controllable action labels all the way until the observable action label coffee and its preceding internal action labels are included, that is

$$
\begin{aligned}
&(\langle \mathbf{ctr}\ \mathsf{request}, \mathsf{flip\_r}, \mathbf{obs}\ \mathsf{beans}, \mathsf{flip\_b}, \mathbf{obs}\ \mathsf{sugar}, \mathsf{flip\_s},\\
&\quad \mathbf{obs}\ \mathsf{water}, \mathsf{flip\_w}, \mathbf{obs}\ \mathsf{coffee}, \mathsf{flip\_r}, \mathsf{flip\_b}, \mathsf{flip\_s}, \mathsf{flip\_w} \rangle,\\
&[\mathsf{false}, \mathsf{false}, \mathsf{false}, \mathsf{false}]) \in extpath_{\mathcal{A}}([\mathsf{false}, \mathsf{false}, \mathsf{false}, \mathsf{false}])
\end{aligned}
$$

The corresponding visible successor path to this extended successor path is

$$
\begin{aligned}
&(\langle \mathbf{ctr}\ \mathsf{request}, \mathbf{obs}\ \mathsf{beans}, \mathbf{obs}\ \mathsf{sugar}, \mathbf{obs}\ \mathsf{water}, \mathbf{obs}\ \mathsf{coffee} \rangle,\\
&[\mathsf{false}, \mathsf{false}, \mathsf{false}, \mathsf{false}]) \in vispath_{\mathcal{A}}([\mathsf{false}, \mathsf{false}, \mathsf{false}, \mathsf{false}])
\end{aligned}
$$

Besides this extended/visible successor path, paths corresponding to all 6 permutations of **obs** beans, **obs** sugar, and **obs** water form extended/visible successor paths. All such paths lead back to the initial state, making it to only visibly reachable state of this action system. Note that in the initial

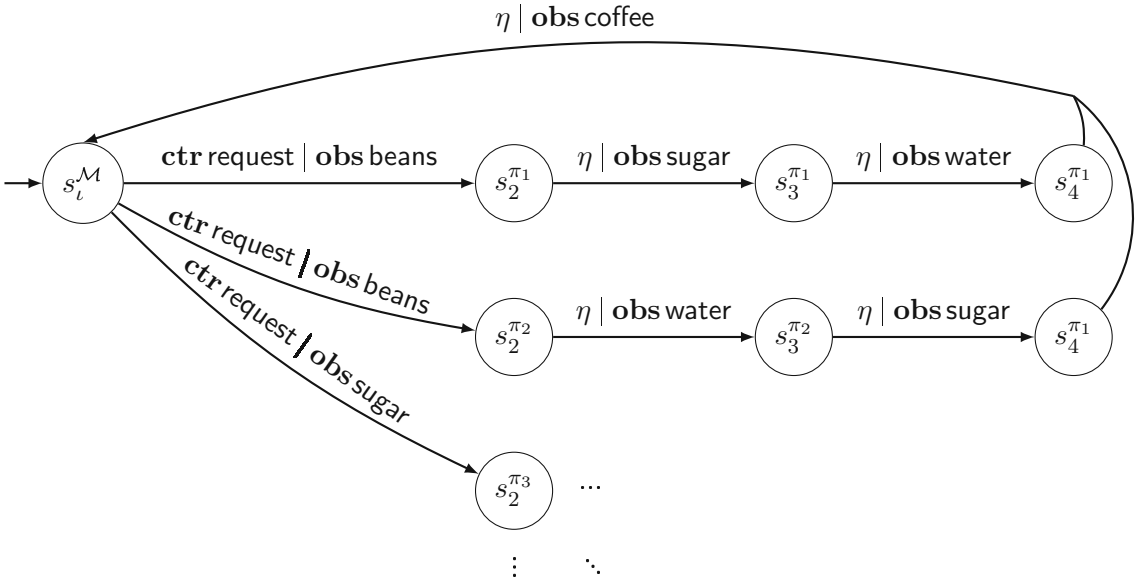


Figure 3.5: The Mealy machine representation of a coffee brewing machine.

state, the main part of the controllable action $\mathbf{ctr}$ request is always enabled. Therefore, the refusing action $\mathbf{obs}\,\eta$ is never enabled. Furthermore, note that for example the sequence of labels $\langle \mathbf{ctr}\,\mathsf{request}, \mathbf{ctr}\,\mathsf{request}, \mathbf{obs}\,\eta \rangle$ is **not** a visible successor path of the initial state, because the action system state $[\mathsf{true}, \mathsf{false}, \mathsf{false}, \mathsf{false}]$ after the instance of $\mathbf{ctr}$ request has successor paths without controllable actions (for example $\langle \mathbf{obs}\,\mathsf{beans}, \mathsf{flip\_b} \rangle$). The Mealy machine translation of this action system is partially depicted in Figure 3.5.

CHAPTER 4

Mutation Testing

Mutation testing analyzes the effect of small syntactic modifications - mutants - of a system under test on the results of test cases. A test case that produces a different result on the system under test and the mutant is said to kill that mutant. Historically, mutation testing was used as a test suite coverage metric. To this end, the mutation score [JH11] is defined as the ratio of mutants killed by the test suite to the number of mutants seeded into the system. The intuition is that the mutation score positively correlates with the error detection capability of the test suite. In addition to using mutants for an a-posteriori coverage metric, mutation-driven test case generation already considers mutants as the basis for test suite generation, specifically aiming to construct killing tests.

In this work, we present mutation-driven test case generation methods in combination with model-based testing of reactive systems. This setting is different from the standard sequential program setting in two main aspects. Firstly, the system that is the basis for test case generation is different from the system under test. In model-based testing, an abstract model is the former and an implementation of it is the latter. Secondly, tests for reactive systems are sequences of inputs and outputs, in contrast to a single set of inputs followed by a single set of outputs. Both aspects have to be taken into account in the definition of mutant killing, as we will do later in this chapter, and during test case generation, which we will present in the following chapters.

In this chapter, we fix the semantics of mutation-driven test case generation in our model-based setting. To this end, we define mutants first abstractly on Mealy machines, and then concretely on symbolic transition- and action systems. We show how multiple mutants can be combined into one model, which makes test case generation methods simpler and more efficient. Finally, we discuss killing mutants in our setting and multiple degrees of killing strength.

4.1 Mutants

In this section, we discuss mutants of models. We start by defining the concept abstractly on Mealy machines and thereafter concretize it on symbolic transition systems and action systems.

Definition 4.1.1 (Mutant)**.** An *abstract mutation operator* for a model $\mathcal{M} = \langle S, S_\iota, \Sigma, \Lambda, \delta \rangle$ is a tuple $\mu = \langle S^\mu, {S_\iota}^\mu, \delta^\mu, \overset{\mu}{\hookrightarrow} \rangle$, where S^μ is a finite set of *mutated states*, ${S_\iota}^\mu \subseteq S^\mu$ is a set of *mutated initial states*, $\delta^\mu \subseteq S^\mu \times \Sigma \times \Lambda \times S^\mu$ is a *mutated transition relation* using the original input and output alphabet, and the *state transformer* $\overset{\mu}{\hookrightarrow} : S \to S^\mu$ is a partial, injective function. The *mutant* induced by μ is the Mealy machine $\mathcal{M}^\mu \overset{\text{def}}{=} \langle S^\mu, {S_\iota}^\mu, \Sigma, \Lambda, \delta^\mu \rangle$. We define the *state projection* function $\overset{\mu}{\hookleftarrow} : S \cup S^\mu \to S \cup S^\mu$ as $\overset{\mu}{\hookleftarrow}(s) \overset{\text{def}}{=} s_1$ if $s \in S^\mu$ and $\overset{\mu}{\hookrightarrow}(s_1) = s$, and $\overset{\mu}{\hookleftarrow}(s) \overset{\text{def}}{=} s$ otherwise.

We again assume that mutants are input-enabled. Thus, in case a mutant disables an input transition, it is implicitly replaced by a self-loop with output η. The intuition of state transformers is to map identical system states onto each other while states that are unique to either the model or the mutant are not mapped. The notion of identical states is abstract for Mealy machines, but easily concretized for variable-based modeling languages, where it can, for example, be defined via variable value equality. Likewise, the notion of mutation operator is abstract for Mealy machines, but as we will see later in this section, is defined syntactically in higher modeling languages.

Example 4.1.1. Consider again the beverage machine presented in Figure 3.1 and its mutants $\mathcal{M}^{\mu_1}$ and $\mathcal{M}^{\mu_2}$, presented in Figure 4.1a nd Figure 4.1b, respectively. Mutant $\mathcal{M}^{\mu_1}$ serves only coffee. Mutant $\mathcal{M}^{\mu_2}$ refills the water tank only half full. The respective state transformers and state projections are given by $\overset{\mu_j}{\hookrightarrow}(s_i) = {s_i}^{\mu_j}$ and $\overset{\mu_j}{\hookleftarrow}({s_i}^{\mu_j}) = s_i$ for $j \in 1, 2$ and $i \in \{1, 2, 3\}$.

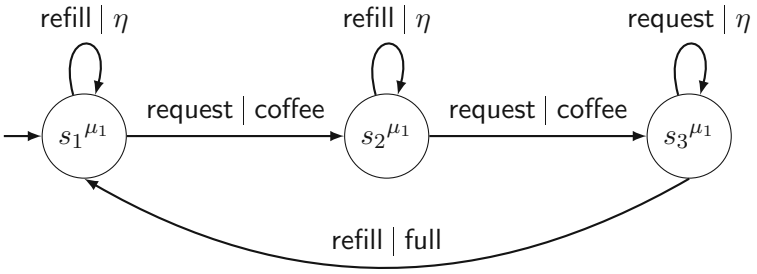
Typically, in mutation testing not only a single-, but a range of mutants are analyzed simultaneously. We capture the simultaneous treatment of multiple mutants in mutation settings:

Definition 4.1.2 (Mutation setting)**.** A *mutation setting* for a model $\mathcal{M} = \langle S, S_\iota, \Sigma, \Lambda, \delta \rangle$ is a set of abstract mutation operators $\{\mu_1, \ldots, \mu_n\}$ for $\mathcal{M}$. For a mutation operator μ_i and its induced mutant $\mathcal{M}^{\mu_i}$, we refer to i as the mutant's *id*.

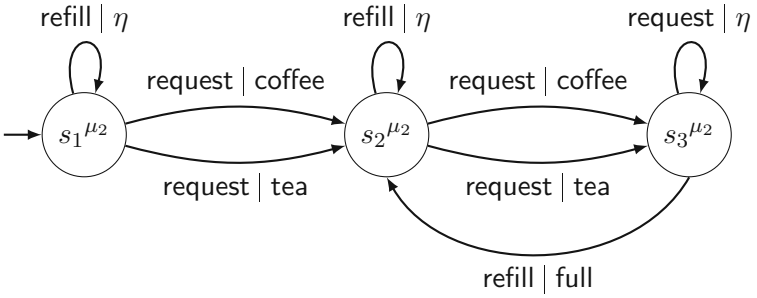
4.1.1 Mutants of Symbolic Transition Systems

Mutants of symbolic transition systems are obtained by applying modifications to the syntactic representation of an STS $\mathcal{S} = \langle \mathcal{I}, \mathcal{O}, \mathcal{X}, \alpha, \delta \rangle$ that we fix throughout this section.

Definition 4.1.3 (Symbolic mutant)**.** A *symbolic mutation operator* is a tuple of formulas $\mu = \langle \gamma, \gamma^\mu, pos \rangle$, such that γ is a sub-formula of α/δ. To disambiguate multiple occurrences of sub-formulas, pos denotes the position of γ in α/δ. The result of applying the mutation operator to $\mathcal{S}$ is a *symbolic mutant* $\mathcal{S}^\mu = \langle \mathcal{I}, \mathcal{O}, \mathcal{X}, \alpha^\mu, \delta^\mu \rangle$, where α^μ/δ^μ is α/δ respectively, where γ at position pos is replaced by γ^μ.



(a) A mutant of the beverage machine model serving only coffee.



(b) A mutant of the beverage machine model filling water only half full.

In order to interpret a symbolic mutation operator μ as an abstract mutation operator on Mealy machines, the set of mutated states, the mutated initial states, and the mutated transition relation are given implicitly via the respective Mealy machine of $\mathcal{S}^{\mu}$ and the state transformer is given as the function that maps a state of $\mathcal{S}$ to a state of $\mathcal{S}^{\mu}$ with equal variable values (if present).

Which mutation operators are applicable heavily depends on the logic used to specify the initial state- and transition relation predicates. In the following, we present mutation operators for modeling languages Verilog in Table 4.1 and SMV in Table 4.2. Models in these languages can be seen as a high level description of symbolic transition systems. Furthermore, such models can be automatically compiled to more literal representations of symbolic transition systems.

We now show how a whole mutation setting can be combined into a single symbolic transition system by introducing an indicating variable and enabling the effect of the respective mutant operator conditioned upon the value of this variable.

Definition 4.1.4 (Symbolic conditional mutant)**.** Let $\{\mu_1, \ldots, \mu_n\}$ be a mutation setting of symbolic mutation operators. The *symbolic conditional mutant* for this mutation setting is defined as $\mathcal{S}^{\bar{\mu}} \stackrel{\text{def}}{=} \langle \mathcal{I}, \mathcal{O}, \mathcal{X} \cup \{\mathrm{mut}\}, \alpha^{\bar{\mu}}, \delta^{\bar{\mu}} \rangle$, where mut is a fresh bounded integer variable[1] with the range $[0, n]$, used to distinguish transitions of the model (represented by value 0) and mutated STS (represented by values 1 to n). The result of conditionally applying a mutation operator $\mu_j = \langle \gamma, \gamma^{\mu_j}, pos \rangle$ with id j to some formula β equals β if γ is not a subformula of β at position pos, and otherwise equals β where γ at position pos is replaced with

[1] Instead of a single bounded integer variable $\lceil \log_2(n+1) \rceil$ Boolean variables can be used to encode mutants.

Type	Mutation
Arithmetic	Exchange binary + and –
	Exchange unary + and –
Relations	Exchange == and ! =
	Exchange <, ≤, >, ≥
Boolean	Exchange ! and ~*
	Drop ! and ~*
	Exchange &&,||,&*,|*, xor and $xnor$
Assignments	Exchange = and <=
	(Blocking & Non-Blocking Assignment)
Constants	Replace Integer Constant c by $0, 1, c+1$, and $c-1$
	Replace Bit-Vector Constant by $\vec{0}$, and $\vec{1}$

Table 4.1: The Verilog mutation operators (* marks bit-wise operations).

Type	Mutation
Structural	Remove branch in case expression
	Swap branches in case expression
	Remove variable assignment
	Remove variable initialization
	Remove transition constraint
Expressions	Expression negation (e is replaced by $\neg e$)
	Logical operator replacement $(\&, \mid, \rightarrow, \leftrightarrow, xor, xnor)$
	Mathematical operator replacement $(+, -, *, /, mod)$
	Relational operator replacement $(=, \neq, <, \leq, >, \geq)$
	Stuck at $0/1$ (replace by false/true)
	Associative shift ($(a\mid b)\&c$ is replaced with $a\mid(b\&c)$)
Values	Enumeration replacement
	Number replacement
	Digit replacement

Table 4.2: The SMV mutation operators.

$((\mathrm{mut} = j) \wedge \gamma^{\mu_j}) \vee ((\mathrm{mut} \neq j) \wedge \gamma)$. The conditionally mutated initial state- and transition relation predicates $\alpha^{\bar{\mu}}, \delta^{\bar{\mu}}$ are defined inductively. Let $\alpha_0, \delta_0 \stackrel{\text{def}}{=} \alpha, \delta \wedge \mathrm{mut}' = \mathrm{mut}$ (the initial value of mut is unconstrained and fixed thereafter) and for $j = 1, \ldots, n$, let α_j, δ_j be the result of conditionally applying mutation operator μ_j to $\alpha_{j-1}, \delta_{j-1}$. During this process, we update the position pos of mutation operators μ_l for $l > j$ such that it points to the potentially altered position of the occurrence of γ_j within the non-mutated branch, i.e. $(\mathrm{mut} \neq j \wedge \gamma)$. Finally, we define the conditionally mutated initial state- and transition relation predicate are defined as the result of this iteration process, i.e. $\alpha^{\bar{\mu}}/\delta^{\bar{\mu}} \stackrel{\text{def}}{=} \alpha_n/\delta_n$.

Example 4.1.2. Consider again the STS presented in Example 3.3.1 and the mutants presented

in Figure 4.1a and Figure 4.1b. The respective symbolic mutation operators are given by $\mu_1 = \langle \texttt{water>0}, \texttt{false}, \delta : 2 : 12 \rangle$ and $\mu_2 = \langle \texttt{water'=2}, \texttt{water'=1}, \delta : 3 : 26 \rangle$, where δ:line:col denotes the line and column in the syntactic representation of δ. The transition relation $\delta^{\bar{\mu}}$ of the conditional mutant for the mutation setting $\{\mu_1, \mu_2\}$ is as follows:

$$
\begin{aligned}
&\big((\texttt{in=request} \wedge \texttt{water>0} \wedge \texttt{out=coffee} \wedge \texttt{water'=water-1}) \vee \\
&\ (\texttt{in=request} \wedge \texttt{out=tea} \wedge \texttt{water'=water-1} \wedge \\
&\qquad (\texttt{(mut=1} \wedge \texttt{false)} \vee \texttt{(mut}\neq\texttt{1} \wedge \texttt{water>0))}) \vee \\
&\ (\texttt{in=refill} \wedge \texttt{water=0} \wedge \texttt{out=full} \wedge \\
&\qquad (\texttt{(mut=2} \wedge \texttt{water'=1}) \vee \texttt{(mut}\neq\texttt{2} \wedge \texttt{water'=2))}) \vee \\
&\ \big(\texttt{in=request} \wedge \texttt{water=0} \wedge \texttt{out=}\eta \wedge \texttt{water'=water}\big)\big) \wedge \\
&\texttt{mut}' = \texttt{mut}
\end{aligned}
$$

4.1.2 Mutants of Action Systems

Mutants of action systems are obtained by applying modification to their expressions. Therefore, mutants of action systems either modify the guard of guarded commands or the assignment of variables. Action system mutants that change the initial state or alter composition of actions are not considered in this work. Throughout this section, we fix an action system $\mathcal{A} = \langle \mathcal{V}, s_\iota, Act, A_\iota \rangle$.

Definition 4.1.5 (Action system mutant)**.** An *action system mutation operator* is a tuple $\mu = \langle \zeta, \zeta^\mu, pos \rangle$, where ζ and ζ^μ are expressions and pos is the position of ζ in the action system that disambiguates multiple occurrences of ζ in actions. The result of applying the mutation operator μ to an action system $\mathcal{A}$ is a *action system mutant* $\mathcal{A}^\mu = \langle \mathcal{V}, s_\iota, Act^\mu, A_\iota \rangle$, where Act^μ is Act in which ζ at position pos is replaced by ζ^μ.

Similarly to symbolic mutation operators, an action system mutation operator can be interpreted via the states and transitions of the Mealy machine corresponding to $\mathcal{A}^\mu$ and the state transformer is given as the function that maps states with equal variable values onto each other.

In Table 4.3, we present the mutation operators for action systems that we consider in this work.

Conditional mutants can be defined on action systems similarly to symbolic conditional mutants.

Definition 4.1.6 (Action system conditional mutant)**.** Let $\{\mu_1, \ldots, \mu_n\}$ be a mutation setting of action system mutation operators. The *action system conditional mutant* for this mutation setting is defined as $\mathcal{A}^{\bar{\mu}} \stackrel{\text{def}}{=} \langle \mathcal{V} \cup \{\mathrm{mut}\}, s_\iota{}^{\bar{\mu}}, Act^{\bar{\mu}}, A_\iota \rangle$, where mut is a fresh bounded integer variable[2] with the range $[0, n]$, used to distinguish transitions of the model (represented by value 0) and mutated action systems (represented by values 1 to n). The initial state $s_\iota{}^{\bar{\mu}}$ is equal to s_ι besides mapping the new variable mut to 0. The result of conditionally applying a mutation

[2] Similarly to symbolic conditional mutants, $\lceil \log_2(n+1) \rceil$ Boolean variables can be used to encode mutants.

Name	Replace Expression	Replace by
Boolean		
Replace by constant	$\mathcal{E}^B$	false or true
Add negation	$\mathcal{E}^B$	$\neg\mathcal{E}^B$
Remove negation	$\neg\mathcal{E}^B$	$\mathcal{E}^B$
Replace binary operator	$\mathcal{E}_1^B \oplus \mathcal{E}_2^B$	$\mathcal{E}_1^B \oplus^\mu \mathcal{E}_2^B$
	$\oplus \in \{=, \neq, \vee, \wedge, \Rightarrow, \Leftrightarrow\}$	$\oplus^\mu \in \{=, \neq, \vee, \wedge, \Rightarrow, \Leftrightarrow\} \setminus \{\oplus\}$
Replace integer relation	$\mathcal{E}_1^I \oplus \mathcal{E}_2^I$	$\mathcal{E}_1^I \oplus^\mu \mathcal{E}_2^I$
	$\oplus \in \{=, \neq, <, >, \leq, \geq\}$	$\oplus^\mu \in \{=, \neq, <, >, \leq, \geq\} \setminus \{\oplus\}$
Replace enum relation	$\mathcal{E}_1^E \oplus \mathcal{E}_2^E$	$\mathcal{E}_1^E \oplus^\mu \mathcal{E}_2^E$
	$\oplus \in \{=, \neq\}$	$\oplus^\mu \in \{=, \neq\} \setminus \{\oplus\}$
Replace quantification	$\oplus v : X.\ \mathcal{E}^B$	$\oplus^\mu v : X.\ \mathcal{E}^B$
	$\oplus \in \{\exists, \forall\}$	$\oplus^\mu \in \{\exists, \forall\} \setminus \{\oplus\}$
Replace set inclusion	$\mathcal{E}^X \oplus \mathcal{E}^{L(X)}$	$\mathcal{E}^X \oplus^\mu \mathcal{E}^{L(X)}$
	$\oplus \in \{\in, \notin\}$	$\oplus^\mu \in \{\in, \notin\} \setminus \{\oplus\}$
Enumeration		
Replace by constant	$\mathcal{E}^E$	Enumeration constant e
Integer		
Replace by constant	$\mathcal{E}^I$	Integer constant k
Replace unary operator	$\oplus\mathcal{E}^I$	$\oplus^\mu \mathcal{E}^I$
	$\oplus \in \{-, \mathsf{abs}\}$	$\oplus^\mu \in \{-, \mathsf{abs}, \mathsf{nop}\} \setminus \{\oplus\}$
Replace binary operator	$\mathcal{E}_1^I \oplus \mathcal{E}_2^I$	$\mathcal{E}_1^I \oplus^\mu \mathcal{E}_2^I$
	$\oplus \in \{+, -, *, /, \mathsf{mod}\}$	$\oplus^\mu \in \{+, -, *, /, \mathsf{mod}\} \setminus \{\oplus\}$
Lists		
Insert tail	$\oplus(\mathcal{E}^{L(X)}), \oplus \in \{\mathsf{tl}, \mathsf{hd}\}$	$\oplus(\mathsf{tl}(\mathcal{E}^{L(X)}))$
Replace tail by head	$\mathsf{tl}(\mathcal{E}^{L(X)})$	$[hd(\mathcal{E}^{L(X)})]$

Table 4.3: The action system mutation operators.

operator $\mu_j = \langle \zeta, \zeta^{\mu_j}, pos \rangle$ with id j to the action that contains ζ at position pos is ζ in which ζ is replaced by $\mathsf{ite}(\mathrm{mut} = j, \zeta^{\mu_j}, \zeta)$. Similarly to symbolic conditional mutants, the final set of conditionally mutated actions is defined as an iterative application of mutation operators and adjusting positions pos.

In contrast to symbolic transition systems, we assume that mut for action systems can be controlled from outside the system by the test case generation algorithm. For a conditional mutant $\mathcal{A}^{\bar{\mu}}$, mutation setting $\{\mu_1, \ldots, \mu_n\}$, $j \in \{0, \ldots, n\}$, and state s, we denote by $path_{\mathcal{A}^{\bar{\mu}}}(j, s)$ and $succ_{\mathcal{A}^{\bar{\mu}}}(j, s)$ the successor paths and states of s in mutant μ_j or the original model for $j = 0$, which are obtained by setting mut to j before computing $path_{\mathcal{A}^{\bar{\mu}}}(s)$ respectively $succ_{\mathcal{A}^{\bar{\mu}}}(s)$ according to the successor state semantics presented in Table 3.2. Thus, the value of mut in ${s_\iota}^{\bar{\mu}}$ is irrelevant and we fix it to 0.

Example 4.1.3. Consider again the action system presented in Example 3.4.1 and the mutants presented in Figure 4.1a and Figure 4.1b. The respective action system mutation operators are given by the Boolean constant replacement $\mu_1 = \langle \mathsf{water} > 0, \mathsf{false}, 4 : 11 \rangle$ and the integer constant replacement $\mu_2 = \langle 2, 1, 8 : 17 \rangle$, where line:col denotes the line and column in the syntactic representation of Act. The resulting set of actions is:

$$
\begin{aligned}
&Act^{\bar{\mu}} = \{ \\
&\quad \mathbf{ctr}\ \mathsf{request} : ((\mathsf{coffee}\ [\!]\ \mathsf{tea});\ \mathsf{decrement})\ /\!/\ \eta, \\
&\quad \mathbf{ctr}\ \mathsf{refill} : (\mathsf{water} = 0 \triangleright \mathsf{reset};\ \mathsf{full})\ /\!/\ \eta, \\
&\quad \mathbf{obs}\ \mathsf{coffee} : \mathsf{true} \triangleright \mathbf{skip}, \\
&\quad \mathbf{obs}\ \mathsf{tea} : \mathsf{ite}(\mathrm{mut} = 1, \mathsf{false}, \mathsf{true}) \triangleright \mathbf{skip}, \\
&\quad \mathbf{obs}\ \mathsf{full} : \mathsf{true} \triangleright \mathbf{skip}, \\
&\quad \mathbf{obs}\ \eta : \mathsf{true} \triangleright \mathbf{skip}, \\
&\quad \mathsf{decrement} : \mathsf{true} \triangleright \mathsf{water} \leftarrow \mathsf{water} - 1, \\
&\quad \mathsf{reset} : \mathsf{true} \triangleright \mathsf{water} \leftarrow \mathsf{ite}(\mathrm{mut} = 2, 1, 2) \\
&\}
\end{aligned}
$$

4.2 Killing Mutants

The purpose of introducing mutants is to construct tests that distinguish the model from its mutants, which is generally referred to as killing a mutant. In this section, we formalize the notion of killing mutants in our setting. We start by discussing the purpose of the tests and the implications to the definitions of mutation killing. Thereafter, we discuss multiple degrees of killing strength. In particular, weak and strong killing differentiate two degrees of fault visibility, whereas potential and definite killing differentiate two degrees of fault reachability with respect to non-determinism. Throughout this section, let us denote an arbitrary mutation operator for model $\mathcal{M} = \langle S, S_\iota, \Sigma, \Lambda, \delta \rangle$ by $\mu = \langle S^\mu, {S_\iota}^\mu, \delta^\mu, \overset{\mu}{\hookrightarrow} \rangle$.

4.2.1 Test Purpose & Conformance

The purpose of the tests created via model-based testing within this work is to verify whether an unknown implementation conforms to the given model, which is assumed to be correct by design. In model-based mutation testing mutants serve as proxy implementations. To this end, similarly to approaches described in [AAJ+14, AJT14, Jöb14, ABJ+15b], we borrow ideas from the conformance testing theory [Tre96] for defining the semantics of mutation killing . Roughly speaking, an implementation conforms to a specification, if no specified sequence of inputs returns an unspecified output. In [Tre96] a set of input-output conformance relations are presented that differ in dealing with input refusal and observability of quiescence (i.e. the inability to proceed further without external stimulus) that are used as the basis of defining mutation killing in [AAJ+14, AJT14, Jöb14, ABJ+15b]. Furthermore, their semantic differences are discussed and it is shown that the conformance relations can be translated into each other by considering different notions of traces.

In this work, in contrast to defining killing via conformance relations, we define killing directly via traces of models and mutants due to the following reasons: Mutants do not correspond to arbitrary implementations, but to small variations in the syntax. In particular, the notion of unspecified inputs does not apply to mutants, since all possible inputs are given by the model. Furthermore, this definition can immediately be applied without the need to study or introduce conformance theory. In particular, the following definition does not depend on determinization that is required to apply some conformance relations, which can be prohibitively expensive in practice. Finally, we do not impose the choice how to deal with input refusal and whether to consider quiescence in the definition of killing, but shift it to the interpretation of higher modeling languages as Mealy machines.

Nevertheless, we maintain the idea that a mutant is only killed if it provides output that can not be witnessed in the model in contrast to the converse situation where a model provides output that can not be witnessed in the mutant. The main reason for this choice is that it is hard or impossible to verify whether the system under test is unable to provide some unwitnessed output, which might be witnessed for example by some unexplored non-deterministic outcome or scheduling, whereas witnessing a spurious output is an unambiguous signal for an error.

The following definition of potential killing corresponds to the input-output testing relation $<_{iot}$ given in [Tre96]. The differences of this relation to potential killing include, as discussed above, the implicit modeling of quiescence and input refusal in $<_{iot}$, which is not imposed by potential killing. Furthermore, in addition to $<_{iot}$ only assuming the model to be input-enabled, our framework around potential killing also assume it of the mutant. Finally, $<_{iot}$ filters stuttering states by considering sequences of observable labels, whereas potential killing does not filter such transitions.

We define killability concepts abstractly on Mealy machines. The respective concepts over symbolic transition system as well as action systems are directly given by their respective Mealy machine translations.

4.2.2 Weak and Strong Killing

Weak and strong killing refer to a folklore notion of fault visibility. Originally, strong killing was considered and weak killing emerged to reduce computational costs of mutation analysis. A mutant is weakly killable if it can produce a sequence of states that does not correspond to any sequence of states of the model. In contrast, strong killing seeks for differences in observable output. We subsume both modes of killing by projections of traces. For weak killing, this projection maps traces to sequences of inputs and states, whereas for strong killing this function maps traces to sequences of inputs and outputs. Formally, we define:

Definition 4.2.1 (Weak kill mode)**.** The weak kill mode $w\kappa$ is the function $w\kappa : \delta^\mu \to \Sigma \times (S \cup S^\mu)$, defined via $w\kappa(s,i,o,s') \stackrel{\text{def}}{=} (i, \overset{\mu}{\hookleftarrow}(s))$, where $\overset{\mu}{\hookleftarrow}(.)$ is the state projection function of mutant μ. We extend $w\kappa$ to traces $p = \langle (s_1,i_1,o_1),(s_2,i_2,o_2),\ldots\rangle$ and sets of traces T by defining $w\kappa(p) \stackrel{\text{def}}{=} \left\langle (i_1, \overset{\mu}{\hookleftarrow}(s_1)),(i_2,\overset{\mu}{\hookleftarrow}(s_2)),\ldots\right\rangle$ and $w\kappa(T) \stackrel{\text{def}}{=} \{w\kappa(p) \mid p \in T\}$.

Definition 4.2.2 (Strong kill mode)**.** The strong kill mode $s\kappa$ is the function $s\kappa : \delta^{\mu} \to \Sigma \times \Lambda$, defined via $s\kappa(s, i, o, s') \stackrel{\text{def}}{=} (i, o)$. We extend $s\kappa$ to traces $p = \langle (s_1, i_1, o_1), (s_2, i_2, o_2), \ldots \rangle$ and sets of traces T by defining $s\kappa(p) \stackrel{\text{def}}{=} \langle (i_1, o_1), (i_2, o_2), \ldots \rangle$ and $s\kappa(T) \stackrel{\text{def}}{=} \{s\kappa(p) \mid p \in T\}$.

4.2.3 Potential and Definite Killing

Potential and definite killing refer to a novel notion of fault reachability guarantee with respect to non-determinism. In non-deterministic models some spurious result might or might not happen. Thus, a potentially killed mutant might or might not be detected by a test. In contrast, a definitely killed mutant is guaranteed to be detected irrespective of non-deterministic outcomes. Formally, we define:

Definition 4.2.3 (Potential killing)**.** $\mathcal{M}^{\mu}$ is *potentially killable* for kill mode κ if:

$$\kappa(\mathcal{T}_{rc}(\mathcal{M}^{\mu})) \notin \kappa(\mathcal{T}_{rc}(\mathcal{M}))$$

A test $t \in \mathcal{T}_{st}(\mathcal{M})$ *potentially kills* $\mathcal{M}^{\mu}$ for kill mode κ if:

$$\kappa(\{p \in \mathcal{T}_{rc}(\mathcal{M}^{\mu}) \mid t|_I = p|_I\}) \notin \kappa(\mathcal{T}_{rc}(\mathcal{M}))$$

Definition 4.2.4 (Definite killing)**.** $\mathcal{M}^{\mu}$ is *definitely killable* for kill mode κ if there is a sequence of inputs $\vec{i} \in \Sigma^*$, such that:

$$\kappa(\{q \in \mathcal{T}_{rc}(\mathcal{M}^{\mu}) \mid q|_I = \vec{i}\}) \cap \kappa(\{p \in \mathcal{T}_{rc}(\mathcal{M}) \mid p|_I = \vec{i}\}) = \varnothing$$

A test $t \in \mathcal{T}_{st}(\mathcal{M})$ *definitely kills* $\mathcal{M}^{\mu}$ for kill mode κ if:

$$\kappa(\{q \in \mathcal{T}_{rc}(\mathcal{M}^{\mu}) \mid q|_I = t|_I\}) \cap \kappa(\{p \in \mathcal{T}_{rc}(\mathcal{M}) \mid p|_I = t|_I\}) = \varnothing$$

Following the tradition in mutation testing, equivalent mutants are defined as mutants that can not be killed. In particular, we define equivalent mutants as mutants not being potentially killable, because it is the weaker form of killing, as we will prove in the following.

Definition 4.2.5 (Equivalent Mutant)**.** $\mathcal{M}^{\mu}$ is *equivalent* if $\mathcal{M}^{\mu}$ is not potentially killable for any kill mode κ.

We now show that definite killing implies potential killing and that the notions coincide for deterministic models. Since the result is independent of the killing mode, we express it for kill mode κ that can either be instantiated with the weak kill mode $w\kappa$ or strong kill mode $s\kappa$.

Proposition 4.2.1. If $\mathcal{M}^{\mu}$ is definitely killable for kill mode κ then $\mathcal{M}^{\mu}$ is potentially killable for kill mode κ.
If $\mathcal{M}^{\mu}$ is deterministic then: $\mathcal{M}^{\mu}$ is potentially killable for kill mode κ if and only if $\mathcal{M}^{\mu}$ is definitely killable for kill mode κ.

Proof. Let $\mathcal{M}^\mu$ be definitely killable for kill mode κ. That is, there is a sequence of inputs $\vec{i}$, such that $\kappa(\{q \in \mathcal{T}_{rc}(\mathcal{M}^\mu) \mid q|_I = \vec{i}\}) \cap \kappa(\{p \in \mathcal{T}_{rc}(\mathcal{M}) \mid p|_I = \vec{i}\}) = \varnothing$. Thus clearly, for every trace $q \in \mathcal{T}_{rc}(\mathcal{M}^\mu)$ with $q|_I = \vec{i}$ it is the case that $\kappa(q) \notin \kappa(\mathcal{T}_{rc}(\mathcal{M}))$. Since $\mathcal{M}^\mu$ is input-enabled, such a trace exists and $\mathcal{M}^\mu$ is potentially killable.

Let $\mathcal{M}$ be deterministic and $\mathcal{M}^\mu$ be potentially killable for kill mode κ. From the definition of determinism it follows that for traces $q, q' \in \mathcal{T}_{rc}(\mathcal{M}^\mu)$ with $q|_I = q'|_I$ it is the case that $q = q'$. Together with the input-enabledness of $\mathcal{M}$, for every sequence of inputs $\vec{i}$ it is the case that $|\{q \in \mathcal{T}_{rc}(\mathcal{M}^\mu) \mid q|_I = \vec{i}\}| = 1$. From potential killability it follows that there exists $q \in \mathcal{T}_{rc}(\mathcal{M}^\mu)$, such that $\kappa(q) \notin \kappa(\{p \in \mathcal{T}_{rc}(\mathcal{M}) \mid p|_I = q|_I\})$. Since the set of traces in the mutant sharing inputs with q is a singleton, it is the case that $\kappa(\{q' \in \mathcal{T}_{rc}(\mathcal{M}^\mu) \mid q'|_I = q|_I\}) \cap \kappa(\{p \in \mathcal{T}_{rc}(\mathcal{M}) \mid p|_I = q|_I\}) = \varnothing$. Therefore, q is a witness to $\mathcal{M}^\mu$ being definitely killable for kill mode κ. □

In summary, potential killability implies definite killability, though for deterministic systems, the two notions coincide. Therefore, for deterministic systems, we simply speak of killing and tests that kill.

Example 4.2.1. Consider again the mutants presented in Example 4.1.1. Mutant $\mathcal{M}^{\mu_1}$ is equivalent, since there are only sequences of corresponding states, and the mutant can not produce a spurious output.

Mutant $\mathcal{M}^{\mu_2}$ is definitely killable, both strongly and weakly. The sequence of inputs $\vec{i}_w = \langle \mathsf{request}, \mathsf{request}, \mathsf{refill} \rangle$ is a witness to weak definite killing of $\mathcal{M}^{\mu_2}$. For any trace $t^\mu \in \mathcal{T}_{rc}(\mathcal{M}^{\mu_2})$ with $t^\mu|_I[1,3] = \vec{i}_w$ it is the case that $t^\mu|_S[1,4] = \langle s_1^{\mu_2}, s_2^{\mu_2}, s_3^{\mu_2}, s_2^{\mu_2} \rangle$, for any trace $t \in \mathcal{T}_{rc}(\mathcal{M})$ with $t|_I[1,3] = \vec{i}_w$ it is the case that $t|_S[1,4] = \langle s_1, s_2, s_3, s_1 \rangle$ and $\overset{\mu_2}{\hookleftarrow}(s_2{}^{\mu_2}) \neq s_1$.

In order to witness a strong kill, the sequence has to be extended in order to force a spurious output of the mutant. The sequence of inputs $\vec{i}_s = \langle \mathsf{request}, \mathsf{request}, \mathsf{refill}, \mathsf{request}, \mathsf{request} \rangle$ is a witness to strong definite killing of $\mathcal{M}^{\mu_2}$. For any trace $t^\mu \in \mathcal{T}_{rc}(\mathcal{M}^{\mu_2})$ with $t^\mu|_I[1,5] = \vec{i}_s$ it is the case that $t^\mu[5]|_O = \eta$, since the water tank must be empty at this point, whereas for any trace $t \in \mathcal{T}_{rc}(\mathcal{M})$ with $t|_I[1,5] = \vec{i}_s$ it is the case that $t^\mu[5]|_O = \mathsf{coffee}$ or $t^\mu[5]|_O = \mathsf{tea}$.

Finally, consider a variant of $\mathcal{M}^{\mu_2}$ that non-deterministically fills the water tank to 1 or 2 units of water upon receiving input refill. The mutant is potentially killable, both strongly and weakly, since the traces of mutant $\mathcal{M}^{\mu_2}$ leading to a spurious state respectively output also exists in this mutant. However, on top of the spurious trace, all traces of the model are present in this variant. Which trace is witnessed only depends on the outcome of non-deterministic choices. Thus, this mutant is not definitely killable.

CHAPTER 5

Mutation Testing with Hyperproperties

In the previous section we provided the definitions of various degrees of mutation killability. These range from potentially weak- to definite strong killability. In this chapter, we provide a logical characterization of potential and definite strong killability in the presence as well as absence of non-determinism. The characterization is in terms of hyperproperties, which reason over the relationship between multiple execution paths.

The hyperproperties expressing strong killability are furthermore encoded into a logic formula, which enables rigorous mutation-driven test generation via model checking. A test case killing some mutant is obtained as a witness to a positive result of model checking the strong killing formula over the conditional mutant, or equivalently as a counter-example to a negative result of model checking the negated formula. However, beyond providing a novel test case generation methodology and potentially even more importantly, our logic characterization allows us to order different degrees of killability in terms of computational complexity. It is intuitively clear that weak killing is easier to achieve than strong killing. Likewise, it is intuitively clear that killing in presence of non-determinism is harder than killing in deterministic systems. Although our logic characterizations only provide upper bounds on the computational complexity to solving the respective problems (i.e. we can not rule out the existence of more efficient characterizations), they nevertheless formalize these intuitions.

We do not consider an explicit logic characterization of weak killing, but it is clear that it can be encoded as a classical a trace property. For example, this can be achieved via suitable assertions stating model and mutant state equality after executing some transition and its mutated counterpart. In contrast, in general strong killing requires differences in internal state to be propagate along paths until they bubble up in the output. That is, strong killability is characterized by two traces that are coordinated via a common sequence of inputs. In other words, strong killability is a hyperproperty and is therefore more complex to verify than weak killability, which is a trace-property.

Similarly, strong killability in deterministic systems is simpler than in non-deterministic systems. While in the former system paths are uniquely given by the coordinating input sequence, in the latter case all paths that correspond to the input sequence need to be taken into account. The consequence is that strong killing for deterministic systems can be characterized by a HyperLTL formula without quantifier alternation, whereas the HyperLTL formalization of the concept in non-deterministic systems requires quantifier alternation. It was shown that moving from no quantifier alternation to formulas with quantifier alternation corresponds to a jump in the computational complexity of the HyperLTL satisfiability problem [FH16] as well as the HyperCTL* model checking problem [FRS15].

Moreover, the situation is similar with potential- and definite killability. Our formalization of the former concept requires two quantifiers, whereas the latter concept requires three quantifiers. While this does not correspond to a jump in computational complexity, it nevertheless shows that in general definite killing corresponds to harder instances in the same complexity class than potential killing.

Since strong killability for non-deterministic models poses a difficult challenge for rigorous methods, we also provide solutions to deal with non-determinism in practice. We present a technique to transform a non-deterministic model into a model where non-determinism is controllable via an additional input variable and we show how killability of the transformed model translates to killability of the non-deterministic model. In addition, we provide SMT characterizations of killability up to a bounded horizon that enables deployment of SMT solver infrastructure to the problem.

We conclude the chapter with an experimental evaluation of mutation-driven test case generation via hyperproperty model checking. To this end, we present a toolchain using off-the-shelf tools and apply it to a series of benchmark models written in two different modeling languages. Finally, we evaluate and validate our methodology by comparing our results on a case study to MoMuT, which is the tool that implements the methods presented in the later chapters of this work.

Example 5.0.1. We illustrate the main concepts of this chapter in Figure 5.1. Figure 5.1a shows a representation of our running example in the SMV modeling language [McM92b] that can be understood as a higher level representation language of symbolic transition systems.

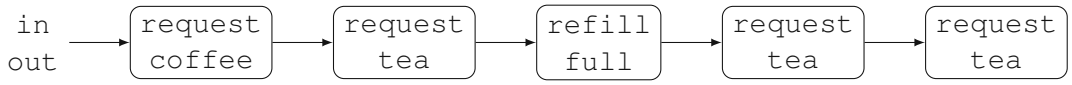
Figure 5.1b contains a hyperproperty over the inputs and outputs of the model formalizing definite killability of the mutant. The test presented in Figure 5.1c is a witness for the strong killability of the mutant: the test requests two drinks after filling the tank. For the mutant, the second request will necessarily fail to produce a beverage, because the water tank is empty. This fact is manifested in Figure 5.1d, which shows all possible output sequences of the mutant to the given test. In particular, the only possible output of the mutant after the last request is η, which is different from the test's output `tea`.

```
1  init(in) :=η
2  init(out):=η
3  init(water):=2
4  init(mut):={0,1}
5  next(in) :={request, fill}
6  next(out):=
7   if(in=request&water>0):{coffee, tea}
8   elif(in=refill&water=0):full
9   else: η
10 next(water):=
11  if(in=refill&water=0):(mut=1 ? 1 : 2)
12  elif(in=request&water>0):water-1
13  else                    :water
14 next(mut):=mut
```

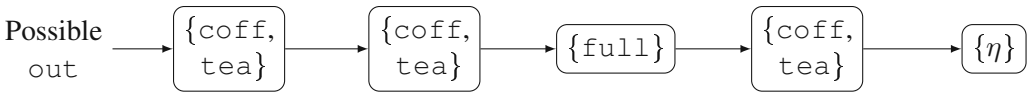
(a) The SMV representation of a conditional mutant of the beverage machine running example.

$$
\begin{aligned}
&\exists\pi\forall\pi'\forall\pi'' \\
&\square(\texttt{mut}_{\pi} = 0) \wedge \\
&\square\big(\texttt{mut}_{\pi'} = 1 \wedge \texttt{mut}_{\pi''} = 0 \wedge \\
&\quad([\texttt{in=request}]_{\pi} \leftrightarrow [\texttt{in=request}]_{\pi'} \\
&\qquad\qquad \leftrightarrow [\texttt{in=request}]_{\pi''}) \wedge \\
&\quad([\texttt{in=refill}]_{\pi} \leftrightarrow [\texttt{in=refill}]_{\pi'} \\
&\qquad\qquad \leftrightarrow [\texttt{in=refill}]_{\pi''})\big) \rightarrow \\
&\Diamond\big(\neg([\texttt{o=}\eta]_{\pi'} \leftrightarrow [\texttt{o=}\eta]_{\pi''}) \vee \\
&\quad\neg([\texttt{o=coffee}]_{\pi'} \leftrightarrow [\texttt{o=coffee}]_{\pi''}) \vee \\
&\quad\neg([\texttt{o=tea}]_{\pi'} \leftrightarrow [\texttt{o=tea}]_{\pi''})\big)
\end{aligned}
$$

(b) The hyperproperty expressing definite killability.



(c) A definitely killing test.



(d) A spurious test response of the mutant.

Figure 5.1: An example demonstrating definite killabiltiy expressed as a hyperproperty.

5.1 Logics for Hyperproperties

In this section, we present HyperLTL and HyperCTL*, two logics expressing hyperproperties. The logics build on atomic propositions, which are basic local statements about properties of system states. To this end, in the following, we show how atomic propositions are introduced to symbolic transition systems. For the remainder of this section, let $\mathcal{S} = \langle \mathcal{I}, \mathcal{O}, \mathcal{X}, \alpha, \delta \rangle$ be a symbolic transition system.

5.1.1 Atomic Propositions

We presume the existence of sets of *atomic propositions* $\mathsf{AP} = \mathsf{AP}_{\mathcal{I}} \cup \mathsf{AP}_{\mathcal{O}} \cup \mathsf{AP}_{\mathcal{X}}$ (intentionally kept abstract)[1] and sets $\mathsf{AP}(I) \subseteq \mathsf{AP}_{\mathcal{I}}, \mathsf{AP}(O) \subseteq \mathsf{AP}_{\mathcal{O}}, \mathsf{AP}(X) \subseteq \mathsf{AP}_{\mathcal{X}}$ that uniquely characterize input I, output O, and state X. In particular, in case $\mathrm{mut} \in \mathcal{X}$ with range $[0, n]$, then we

[1] Finite domains can be characterized using binary encodings; infinite domains require an extension of our formalism in Section 5.1.2 with equality and is omitted for the sake of simplicity.

presume the existence of $n+1$ atomic propositions $[\mathrm{mut} = j]$ for $j \in [0, n]$ that fix the value of mut to j respectively.

For a symbolic trace $p = \langle (X_1, I_1, O_1), (X_2, I_2, O_2), \ldots \rangle \in \mathcal{ST}_{rc}(\mathcal{S})$ its *atomic proposition trace* is defined as $\mathsf{AP}(p) \stackrel{\text{def}}{=} \langle \mathsf{AP}(X_1) \cup \mathsf{AP}(I_1) \cup \mathsf{AP}(O_1), \mathsf{AP}(X_2) \cup \mathsf{AP}(I_2) \cup \mathsf{AP}(O_2), \ldots \rangle$. We lift this definition to sets of traces by defining $\mathsf{APTr}(\mathcal{S}) \stackrel{\text{def}}{=} \{\mathsf{AP}(p) \mid p \in \mathcal{ST}_{rc}(\mathcal{S})\}$.

5.1.2 HyperLTL

In the following, we provide an overview of HyperLTL, a logic for hyperproperties, in sufficient detail for our formalization of strong killing presented in Section 5.2. For further details on the logic, we refer the reader to [CFK+14]. HyperLTL is defined over atomic proposition traces as defined above.

Syntax. Let AP be a set of atomic propositions and let π be a *trace variable* from a set $\mathcal{V}_{tr}$ of trace variables. Formulas of HyperLTL are defined by the following grammar:

$$\begin{array}{lllllllll} \psi & ::= & \exists \pi.\psi & \mid & \forall \pi.\psi & \mid & \varphi & & \\ \varphi & ::= & a_\pi & \mid & \neg\varphi & \mid & \varphi \vee \varphi & \mid & \mathrm{X}\varphi \mid \varphi \,\mathrm{U}\, \varphi \end{array}$$

Connectives $\exists$ and $\forall$ are universal and existential trace quantifiers, read '*along some traces*' and '*along all traces*'. In our setting, atomic propositions $a \in \mathsf{AP}$ express facts about states or the presence of inputs and outputs. Each atomic proposition is sub-scripted with a trace variable to indicate the trace it is associated with. The Boolean connectives $\wedge$, $\rightarrow$, and $\leftrightarrow$ are defined in terms of $\neg$ and $\vee$ as usual. Temporal operators X and U read '*next*' and '*until*', respectively. Furthermore, we use the standard temporal operators '*eventually*' $\Diamond\varphi \stackrel{\text{def}}{=} \mathsf{true}\,\mathrm{U}\varphi$, and '*always*' $\Box\varphi \stackrel{\text{def}}{=} \neg\Diamond\neg\varphi$.

Semantics. $\Pi \vDash_{\mathcal{S}} \psi$ states that ψ is valid for a given mapping $\Pi : \mathcal{V}_{tr} \to \mathsf{APTr}(\mathcal{S})$ of trace variables to atomic proposition traces. Let $\Pi[\pi \mapsto p]$ be as Π except that π is mapped to p. We use $\Pi[i, \infty]$ to denote the trace assignment $\Pi'(\pi) = \Pi(\pi)[i, \infty]$ for all π. The validity of a formula is defined as follows:

$$\begin{array}{lll} \Pi \vDash_{\mathcal{S}} a_\pi & \text{iff} & a \in \Pi(\pi)[0] \\ \Pi \vDash_{\mathcal{S}} \exists \pi.\psi & \text{iff} & \text{there exists } p \in \mathsf{APTr}(\mathcal{S}) : \Pi[\pi \mapsto p] \vDash_{\mathcal{S}} \psi \\ \Pi \vDash_{\mathcal{S}} \forall \pi.\psi & \text{iff} & \text{for all } p \in \mathsf{APTr}(\mathcal{S}) : \Pi[\pi \mapsto p] \vDash_{\mathcal{S}} \psi \\ \Pi \vDash_{\mathcal{S}} \neg\varphi & \text{iff} & \Pi \nvDash_{\mathcal{S}} \varphi \\ \Pi \vDash_{\mathcal{S}} \psi_1 \vee \psi_2 & \text{iff} & \Pi \vDash_{\mathcal{S}} \psi_1 \text{ or } \Pi \vDash_{\mathcal{S}} \psi_2 \\ \Pi \vDash_{\mathcal{S}} \mathrm{X}\varphi & \text{iff} & \Pi[1, \infty] \vDash_{\mathcal{S}} \varphi \\ \Pi \vDash_{\mathcal{S}} \varphi_1 \,\mathrm{U}\, \varphi_2 & \text{iff} & \text{there exists } i \geq 0 : \Pi[i, \infty] \vDash_{\mathcal{S}} \varphi_2 \\ & & \text{and for all } 0 \leq j < i \text{ we have } \Pi[j, \infty] \vDash_{\mathcal{S}} \varphi_1 \end{array}$$

We write $\mathcal{S} \vDash \psi$ if $\Pi \vDash_{\mathcal{S}} \psi$ holds and Π is empty. We call $q \in \mathcal{ST}_{rc}(\mathcal{S})$ a π-witness of a formula $\exists\pi.\psi$ or $\forall\pi.\psi$, if $\Pi[\pi \mapsto p] \vDash_{\mathcal{S}} \psi$ and $\mathsf{AP}(q) = p$.

5.1.3 HyperCTL*

HyperCTL* is an extension of HyperLTL described in [CFK+14]. We recite the necessary concepts of the logic here and refer the reader to [CFK+14] for further details.

Syntax. HyperCTL* syntactically is a strict superset of HyperLTL. It allows free mixing temporal operators and path quantifiers. HyperCTL* formulas are defined by the following grammar:

$$\varphi ::= a_\pi \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathrm{X}\varphi \mid \varphi \,\mathrm{U}\, \varphi \mid \exists \pi \varphi$$

Further temporal operators, such as $\square$ and $\lozenge$, are defined as usual. Universal quantification in HyperCTL* is defined via negation and existential quantification: $\forall \pi \varphi \stackrel{\text{def}}{=} \neg \exists \pi \neg \varphi$.

Semantics. Quantification in HyperCTL* is over paths, which are sequences of system states and atomic proposition pairs, in contrast to HyperLTL where quantification is over sequences of atomic propositions only. In particular, paths assigned to path quantifiers within temporal operators, start in the respective system state currently reasoned over by the temporal operator.

In order to disambiguate the notions, we write $\Pi^* : \mathcal{V}_{tr} \to (\mathcal{Y}^{\mathcal{S}} \times \mathsf{AP})^\omega$ for path assignments, $\models^*$ for the HyperCTL* modeling relation and π^*-witness for witness paths of HyperCTL* formulas. Finally, for ease of presentation, when working with HyperCTL* formulas, we assume that STS have a single initial state. An arbitrary STS can easily be transformed into this form by introducing a unique initial state and transitions that lead to all initial states. The formal semantics of HyperCTL* are given as follows:

$\Pi^* \models^*_{\mathcal{S}} a_\pi$	iff	$a \in \mathsf{AP}(\Pi^*(\pi)[0])$
$\Pi^* \models^*_{\mathcal{S}} \exists \pi.\psi$	iff	there exists $y \in (\mathcal{Y}^{\mathcal{S}} \times \mathsf{AP})^\omega$ such that
		$y[0] = \Pi^*(\pi)[0]$ and $\Pi^*[\pi \mapsto y] \models^*_{\mathcal{S}} \psi$
$\Pi^* \models^*_{\mathcal{S}} \neg\varphi$	iff	$\Pi^* \not\models^*_{\mathcal{S}} \varphi$
$\Pi^* \models^*_{\mathcal{S}} \mathrm{X}\varphi$	iff	$\Pi^*[1,\infty] \models^*_{\mathcal{S}} \varphi$
$\Pi^* \models^*_{\mathcal{S}} \varphi_1 \,\mathrm{U}\, \varphi_2$	iff	there exists $i \geq 0 : \Pi^*[i,\infty] \models^*_{\mathcal{S}} \varphi_2$
		and for all $0 \leq j < i$ we have $\Pi^*[j,\infty] \models^*_{\mathcal{S}} \varphi_1$

To illustrate nested path assignments, consider the HyperCTL* formula with nested path quantifiers $\exists\pi\square((x' = x+1)_\pi \wedge \exists\pi' \mathsf{T}_{\pi'})$ evaluated over some symbolic transition system. Paths assigned to π, corresponding to the outer path quantifier, start in the initial state of the symbolic transition system. Furthermore, due to $\square(x' = x+1)_\pi$, along these paths, the value of x increases by one in each state. In contrast, paths assigned to π', corresponding to the path quantifier $\exists\pi'$ nested inside $\exists\pi\square$, start in the current state of the path assigned to the outer quantifier $\exists\pi$. For example, if the initial state of the symbolic transition system maps x to 0 then the paths assigned to π' start in states that map x to 0, 1, 2, etc.

5.1.4 Model Checking Complexity

The complexity of model checking HyperLTL as well as HyperCTL* formulas was studied in [CFK+14, FRS15, BF18]. The general problem of deciding $K \vDash \varphi$, i.e. model checking some HyperLTL, or HyperCTL* formula φ over some Kripke structure K is decidable, but non-elementary [CFK+14]. Both results use the fact that HyperLTL is subsumed by HyperCTL*. The decidability result is shown via a reduction of HyperCTL* model checking to deciding satisfiability of quantified propositional temporal logic QPTL. The non-elementary complexity result is shown via a reduction of model checking epistemic logic ETL to model checking HyperLTL.

While the general complexity of model checking HyperLTL and HyperCTL* formulas is very high, subsets of the general logic with lower model checking complexities have been identified. Foremost, the complexity of the problem heavily depends on the quantifier alternation depth of the checked formula (for example, $\forall\pi\forall\pi' a_\pi \wedge a_{\pi'}$ has quantifier alternation depth 0, whereas $\forall\pi\exists\pi'\forall\pi'' a_\pi \wedge a_{\pi'} \wedge a_{\pi''}$ has quantifier alternation depth 2). In particular, the complexity of model checking a HyperCTL* formula with quantifier alternation depth 0 is PSPACE-complete in the size of the formula as well as NLOGSPACE-complete in the size of the model [FRS15]. From there, the complexity grows exponentially in the quantifier alternation depth. In particular, the complexity of model checking a HyperCTL* formula φ with quantifier alternation depth k over a kripke structure K is in NSPACE($g(k, |\varphi|)$) and in NSPACE($g(k-1, |K|)$), where $g(k, n)$ is a tower of exponentiations of height k [FRS15]. The reason for these complexity jumps is the need for expensive model complementation of ω-automata.

It is interesting future work to identify under which conditions complementation of ω-automata is required to model check the properties capturing mutation killability introduced in the following section. Potentially, a tailored model checking method for these properties could be developed that does not suffer from this exponential blow-up due to complementation. A hint towards such an algorithm is given by [FHT19], presenting canonical automata expressing all prefixes of counter-examples of regular k-safety hyperproperties [CS10].

Finally, the complexity can be further reduced by considering acyclic or tree-shaped models [BF18]. In this work, we make no such assumptions, but they offer another path to scale the presented approach further in future work.

5.2 Killing with Hyperproperties

In this section, we provide a formalization of potential and definite killability in terms of HyperLTL, prove the correctness of our formalizations with respect to definitions given in Section 4.2, and explain how tests can be extracted by model checking these HyperLTL formulas. Furthermore, we present an encoding of locally adaptive tests in HyperCTL*. To this end, let us fix a symbolic conditional mutant $\mathcal{S}^{\bar{\mu}} \stackrel{\text{def}}{=} \langle \mathcal{I}, \mathcal{O}, \mathcal{X} \cup \{\mathrm{mut}\}, \alpha^{\bar{\mu}}, \delta^{\bar{\mu}} \rangle$ for a mutation setting $\{\mu_1, \ldots, \mu_n\}$.

All HyperLTL and HyperCTL* formulas presented in this section depend on a mutant id as well as on inputs and outputs of the model, but are model-agnostic otherwise. The idea of all presented formulas is to discriminate between traces of the model ($\square[\mathrm{mut} = 0]_\pi$) and traces of

the mutant ($\square[\text{mut} = j]_\pi$). Furthermore, we quantify over pairs (π, π') of traces with globally equal inputs ($\square(I_\pi \leftrightarrow I_{\pi'})$) and express that such pairs will eventually have different outputs ($\lozenge(O_\pi \not\leftrightarrow O_{\pi'})$), where for ease of presentation, we abbreviate $\bigwedge_{i \in \mathsf{AP}_\mathcal{I}}(i_\pi \leftrightarrow i_{\pi'})$ by $I_\pi \leftrightarrow I_{\pi'}$ and $\bigvee_{o \in \mathsf{AP}_\mathcal{O}} \neg(o_\pi \leftrightarrow o_{\pi'})$ by $O_\pi \not\leftrightarrow O_{\pi'}$. We start by showing some general properties used throughout the following HyperLTL formalizations of killability.

Lemma 5.2.1. Let Π be a trace assignment, let p, q be sequences of system states of $\mathcal{S}^{\bar{\mu}}$ with $\mathsf{AP}(p) = \Pi(\pi)$, $\mathsf{AP}(q) = \Pi(\pi')$, and let $j \in \{1, \dots, n\}$. All of the following statements are true:

1. $\Pi \vDash_{\mathcal{S}^{\bar{\mu}}} \square[\text{mut} = 0]_\pi$ then $p|_{\mathcal{I} \cup \mathcal{O} \cup \mathcal{X}} \in \mathcal{ST}_{rc}(\mathcal{S})$
2. $\Pi \vDash_{\mathcal{S}^{\bar{\mu}}} \square[\text{mut} = j]_\pi$ then $p|_{\mathcal{I} \cup \mathcal{O} \cup \mathcal{X}} \in \mathcal{ST}_{rc}(\mathcal{S}^{\mu_j})$
3. $\Pi \vDash_{\mathcal{S}^{\bar{\mu}}} \square\big(\bigwedge_{i \in \mathsf{AP}_\mathcal{I}}(i_\pi \leftrightarrow i_{\pi'})\big)$ then $p|_\mathcal{I} = q|_\mathcal{I}$
4. $\Pi \vDash_{\mathcal{S}^{\bar{\mu}}} \lozenge\big(\bigvee_{o \in \mathsf{AP}_\mathcal{O}} \neg(o_\pi \leftrightarrow o_{\pi'})\big)$ then $p|_\mathcal{O} \neq q|_\mathcal{O}$

Proof. The first two statements follow directly from the definition of symbolic conditional mutants. In particular, recall that conditionally mutated sub-formulas of $\alpha^{\bar{\mu}}$ and $\delta^{\bar{\mu}}$ have the form $(\text{mut} = j \wedge \gamma^{\mu_j}) \vee (\text{mut} \neq j \wedge \gamma)$. A trace that satisfies $\square[\text{mut} = 0]_\pi$ can never satisfy sub-formula $(\text{mut} = j \wedge \gamma^{\mu_j})$, while it satisfies $(\text{mut} \neq j \wedge \gamma)$ if and only if the respective trace of the model satisfies γ. Since this is true for every mutated sub-formula, clearly $\square[\text{mut} = 0]_\pi$ implies $p|_{\mathcal{I} \cup \mathcal{O} \cup \mathcal{X}} \in \mathcal{ST}_{rc}(\mathcal{S})$. The argument for $\square[\text{mut} = j]_\pi$ is symmetric. The latter two statements follow directly from the fact that $\mathsf{AP}_\mathcal{I}, \mathsf{AP}_\mathcal{O}$ uniquely characterize inputs and outputs. □

5.2.1 Deterministic Model and Mutant

To show killability (potential and definite) of a deterministic mutant for a deterministic model, one needs to find a trace of the model ($\exists\pi$) such that the trace of the mutant with the same inputs ($\exists\pi'$) eventually diverges in outputs, which is formalized via the hyperproperty ϕ_1 as follows:

$$\phi_1(\mathcal{I}, \mathcal{O}, j) \overset{\text{def}}{=} \exists\pi \exists\pi' \square\big([\text{mut} = 0]_\pi \wedge [\text{mut} = j]_{\pi'} \wedge (I_\pi \leftrightarrow I_{\pi'})\big) \wedge \lozenge\big(O_\pi \not\leftrightarrow O_{\pi'}\big)$$

Proposition 5.2.1. For a deterministic model $\mathcal{S}$ and mutant $\mathcal{S}^{\mu_j}$ it holds that

$$\mathcal{S}^{\bar{\mu}} \vDash \phi_1(\mathcal{I}, \mathcal{O}, j) \text{ iff } \mathcal{S}^{\mu_j} \text{ is killable.}$$

For every π-witness p of $\mathcal{S}^{\bar{\mu}} \vDash \phi_1(\mathcal{I}, \mathcal{O}, j)$ there is some $n \in \mathbb{N}$ such that the test $t \overset{\text{def}}{=} (p[1, n]|_{\mathcal{I} \cup \mathcal{O}})^{\mathcal{M}}$ kills $\mathcal{S}^{\mu_j}$.

Proof. We show that $\mathcal{S}^{\mu_j}$ is potentially killable *iff* $\mathcal{S}^{\bar{\mu}} \vDash \phi_1(\mathcal{I}, \mathcal{O}, j)$. This suffices, since $\mathcal{S}^{\mu_j}$ is deterministic and by the correspondence of symbolic traces to Mealy machine traces (Lemma 3.3.1), $\mathcal{S}^{\mu_j}$ is definitely killable *iff* $\mathcal{S}^{\mu_j}$ is potentially killable.

Assume that $\mathcal{S}^{\mu_j}$ is potentially killable. That is, there is a trace $q_1 \in \mathcal{T}_{rc}(\mathcal{S}^{\mu_j})$, such that $q^{\mathcal{M}}|_{I/O} \notin \mathcal{T}_{rc}(\mathcal{S})|_{I/O}$. Since $\mathcal{S}^{\mu_j}$ is input-enabled, there exists a trace $p \in \mathcal{ST}_{rc}(\mathcal{S})$, such that $p|_{\mathcal{I}} = q|_{\mathcal{I}}$. Together the correspondence of traces and symbolic traces (Lemma 3.3.1) it follows that $p|_{\mathcal{O}} \neq q|_{\mathcal{O}}$. Furthermore, from Lemma 5.2.1 follows that p and q are π- and π'-witnesses for $\square[\mathrm{mut} = 0]_{\pi}$ and $\square[\mathrm{mut} = j]_{\pi'}$ respectively. Therefore, p and q are satisfying assignments for $\phi_1(\mathcal{I}, \mathcal{O}, j)$ and π, π' respectively.

Conversely, assume $\mathcal{S}^{\bar{\mu}} \vDash \phi_1(\mathcal{I}, \mathcal{O}, j)$. Let p, q be a π, π'-witnesses of $\phi_1(\mathcal{I}, \mathcal{O}, j)$. From Lemma 5.2.1, we immediately get $p|_{\mathcal{I}} = q|_{\mathcal{I}}$, and $p|_{\mathcal{O}} \neq q|_{\mathcal{O}}$, $p|_{\mathcal{I}\cup\mathcal{O}\cup\mathcal{X}} \in \mathcal{ST}_{rc}(\mathcal{S})$, and $q|_{\mathcal{I}\cup\mathcal{O}\cup\mathcal{X}} \in \mathcal{ST}_{rc}(\mathcal{S}^{\mu_j})$, which shows $\mathcal{ST}_{rc}(\mathcal{S}^{\mu_j})|_{\mathcal{I}\cup\mathcal{O}} \notin \mathcal{ST}_{rc}(\mathcal{S})|_{\mathcal{I}\cup\mathcal{O}}$. Thus, together with Lemma 3.3.1 we showed that $\mathcal{S}^{\mu_j}$ is potentially killable.

Furthermore, since $p|_{\mathcal{O}} \neq q|_{\mathcal{O}}$, there exists an $n \in \mathbb{N}$ such that $p[1, n-1]|_{\mathcal{O}} = q[1, n-1]|_{\mathcal{O}}$ and $p[n]|_{\mathcal{O}} \neq q[n]|_{\mathcal{O}}$. Thus, the test $t \stackrel{\text{def}}{=} (p[1, n]|_{\mathcal{I}\cup\mathcal{O}})^{\mathcal{M}}$ kills $\mathcal{S}^{\mu_j}$. □

5.2.2 Non-deterministic Model and Mutant

For potential killability of non-deterministic models and mutants, we need to find a trace of the mutant ($\exists\pi$) such that all traces of the model with the same inputs ($\forall\pi'$) eventually diverge in outputs, which is formalized via the hyperproperty ϕ_2 as follows:

$$\phi_2(\mathcal{I}, \mathcal{O}, j) \stackrel{\text{def}}{=} \exists\pi\forall\pi' \square[\mathrm{mut} = j]_{\pi} \wedge \Big(\square\big([\mathrm{mut} = 0]_{\pi'} \wedge (I_{\pi} \leftrightarrow I_{\pi'})\big) \rightarrow \Diamond\big(O_{\pi} \not\leftrightarrow O_{\pi'}\big)\Big)$$

Proposition 5.2.2. For non-deterministic model $\mathcal{S}$ and mutant $\mathcal{S}^{\mu_j}$, it holds that

$$\mathcal{S}^{\bar{\mu}} \vDash \phi_2(\mathcal{I}, \mathcal{O}, j) \text{ iff } \mathcal{S}^{\mu_j} \text{ is potentially killable.}$$

If q is a π-witness for $\mathcal{S}^{\bar{\mu}} \vDash \phi_2(\mathcal{I}, \mathcal{O}, j)$, then for any trace $p \in \mathcal{ST}_{rc}(\mathcal{S})$ with $q|_{\mathcal{I}} = p|_{\mathcal{I}}$ there is $n \in \mathbb{N}$ such that the test $t \stackrel{\text{def}}{=} (p[1, n]|_{\mathcal{I}\cup\mathcal{O}})^{\mathcal{M}}$ potentially kills $\mathcal{S}^{\mu_j}$.

Proof. Assume that $\mathcal{S}^{\mu_j}$ is potentially killable. From the correspondence of traces and symbolic traces (Lemma 3.3.1) it follows that there is a symbolic trace $q \in \mathcal{ST}_{rc}(\mathcal{S}^{\mu_j})$, such that there is no symbolic trace $p \in \mathcal{ST}_{rc}(\mathcal{S})$ with $q|_{\mathcal{I}\cup\mathcal{O}} = p|_{\mathcal{I}\cup\mathcal{O}}$. Any trace assignment that maps π to q satisfies $\phi_2(\mathcal{I}, \mathcal{O}, j)$, since that assignment either violates the antecedent of the implication by mapping a trace $p \in \mathcal{ST}_{rc}(\mathcal{S})$ with different inputs than q to π', or it violates the consequent by mapping a trace $p \in \mathcal{ST}_{rc}(\mathcal{S})$ to π' with inputs $q|_{\mathcal{I}}$ and outputs that can only be different to $q|_{\mathcal{O}}$.

Conversely, assume $\mathcal{S}^{\bar{\mu}} \vDash \phi_2(\mathcal{I}, \mathcal{O}, j)$. Let p be a π-witness and q be a π'-witness for which the antecedent of the implication is satisfied, which is in fact satisfiable, since $\mathcal{S}^{\mu_j}$ is input-enabled. Clearly, p is a π-witness for $\square[\mathrm{mut} = j]_{\pi}$ and since q is chosen such that it satisfies the antecedent, q is a π'-witness for $\square[\mathrm{mut} = 0]_{\pi'}$. Thus, from Lemma 5.2.1, we get $p|_{\mathcal{I}\cup\mathcal{O}\cup\mathcal{X}} \in \mathcal{ST}_{rc}(\mathcal{S})$, $q|_{\mathcal{I}\cup\mathcal{O}\cup\mathcal{X}} \in \mathcal{ST}_{rc}(\mathcal{S}^{\mu_j})$, and $q|_{\mathcal{I}} = p|_{\mathcal{I}}$. Since $\Pi[\pi \mapsto p, \pi' \mapsto q]$ satisfies the antecedent of the implication and the whole formula, the trace assignment also satisfies the consequent of

the implication , i.e. $q|_{\mathcal{O}} \neq p|_{\mathcal{O}}$ (Lemma 5.2.1). Since q was chosen arbitrary (besides satisfying the antecedent), we can conclude $p|_{\mathcal{I}\cup\mathcal{O}} \notin \mathcal{ST}_{rc}(\mathcal{S}^{\mu_j})|_{\mathcal{I}\cup\mathcal{O}}$, which together with Lemma 3.3.1 shows that $\mathcal{S}^{\mu_j}$ is potentially killable.

Let $q \in \mathcal{ST}_{rc}(\mathcal{S}^{\mu_j})$ be a π-witness to $\mathcal{S}^{\bar{\mu}} \vDash \phi_2(\mathcal{I}, \mathcal{O}, j)$ and let $p \in \mathcal{ST}_{rc}(\mathcal{S})$ be any trace with $p|_{\mathcal{I}} = q|_{\mathcal{I}}$, which exists since $\mathcal{S}^{\mu_j}$ is input-enabled. Clearly, there exists an $n \in \mathbb{N}$ such that $q[1, n-1]|_{\mathcal{O}} = p[1, n-1]|_{\mathcal{O}}$ and $q[n]|_{\mathcal{O}} \neq p[n]|_{\mathcal{O}}$. Therefore, the test $t \stackrel{\text{def}}{=} (p[1,n]|_{\mathcal{I}\cup\mathcal{O}})^{\mathcal{M}}$ potentially kills $\mathcal{S}^{\mu_j}$. □

For definite killability one needs to find a sequence of inputs of the model ($\exists\pi$) and compare all non-deterministic outcomes of the model ($\forall\pi''$) to all non-deterministic outcomes of the mutant ($\forall\pi'$) for these inputs, which is formalized via the hyperproperty ϕ_3 as follows:

$$\phi_3(\mathcal{I}, \mathcal{O}, j) \stackrel{\text{def}}{=}$$
$$\exists\pi\forall\pi'\forall\pi''\,\square[\text{mut} = 0]_\pi \wedge \Big(\square\big([\text{mut} = j]_{\pi'} \wedge [\text{mut} = 0]_{\pi''} \wedge (I_\pi \leftrightarrow I_{\pi'}) \wedge (I_\pi \leftrightarrow I_{\pi''})\big) \rightarrow \Diamond\big(O_{\pi'} \not\leftrightarrow O_{\pi''}\big)\Big)$$

In Figure 5.1b, we present an instance of ϕ_3 for our running example.

Proposition 5.2.3. For non-deterministic $\mathcal{S}$ and $\mathcal{S}^{\mu_j}$, it holds that

$$\mathcal{S}^{\bar{\mu}} \vDash \phi_3(\mathcal{I}, \mathcal{O}, j) \text{ iff } \mathcal{S}^{\mu_j} \text{ is definitely killable.}$$

If p is a π-witness for $\mathcal{S}^{\bar{\mu}} \vDash \phi_3(\mathcal{I}, \mathcal{O}, j)$, then there exists $n \in \mathbb{N}$, such that the test $t \stackrel{\text{def}}{=} (p[1,n]|_{\mathcal{I}\cup\mathcal{O}})^{\mathcal{M}}$ definitely kills $\mathcal{S}^{\mu_j}$.

Proof. Let $\mathcal{S}^{\mu_j}$ be definitely killable, which, together the correspondence of traces and symbolic traces (Lemma 3.3.1), implies that there is a sequence of inputs $\vec{i} \in (\Sigma^{\mathcal{S}})^*$, such that for $P_{\vec{i}} \stackrel{\text{def}}{=} \{p \in \mathcal{ST}_{rc}(\mathcal{S}) \mid p|_{\mathcal{I}} = \vec{i}\}$ and $Q_{\vec{i}} \stackrel{\text{def}}{=} \{q \in \mathcal{ST}_{rc}(\mathcal{S}^{\mu_j}) \mid q|_{\mathcal{I}} = \vec{i}\}$ it is the case that $P_{\vec{i}}|_{\mathcal{O}} \cap Q_{\vec{i}}|_{\mathcal{O}} = \varnothing$. Since $\mathcal{S}$ and $\mathcal{S}^{\mu_j}$ are input-enabled, it is the case that $P_{\vec{i}} \neq \varnothing$ and $Q_{\vec{i}} \neq \varnothing$. We show that any $p \in P_{\vec{i}}$ is a π-witness to $\mathcal{S}^{\bar{\mu}} \vDash \phi_3(\mathcal{I}, \mathcal{O}, j)$. Let $q' \in \mathcal{ST}_{rc}(\mathcal{S}^{\mu_j})$ and $p'' \in \mathcal{ST}_{rc}(\mathcal{S})$ be arbitrary traces. Furthermore, consider a trace assignment that maps π to p, π' to q' and π'' to p'' and assume that it satisfies the antecedent (which is satisfiable, due to $P_{\vec{i}} \neq \varnothing$ and $Q_{\vec{i}} \neq \varnothing$). That is, $q' \in Q_{\vec{i}}$ and $p'' \in P_{\vec{i}}$. Since $P_{\vec{i}}|_{\mathcal{O}} \cap Q_{\vec{i}}|_{\mathcal{O}} = \varnothing$, it must be the case that $q'|_{\mathcal{O}} \neq p''|_{\mathcal{O}}$. Thus, in case q' and p'' are π'- and π''- witnesses that also satisfy the antecedent, any trace assignment that maps p to π satisfies the formula. Likewise, in case q' and p'' are π'- and π''-witnesses that do not satisfy the antecedent, then the whole formula is satisfied. Therefore, together with Lemma 3.3.1, we showed that $\mathcal{S}^{\bar{\mu}} \vDash \phi_3(\mathcal{I}, \mathcal{O}, j)$.

Conversely, assume $\mathcal{S}^{\bar{\mu}} \vDash \phi_3(\mathcal{I}, \mathcal{O}, j)$. Let p be a π-witness, and let q' and p'' be π' and π''-witnesses for which the antecedent is satisfied, which is in fact satisfiable, since $\mathcal{S}^{\mu_j}$ is input-enabled. Clearly, p is a π-witness for $\square[\text{mut} = 0]_\pi$ and since q' and p'' were chosen such

that they satisfy the antecedent, q' is a π'-witness for $\square[\mathrm{mut} = j]_{\pi'}$ and p is a π''-witness for $\square[\mathrm{mut} = 0]_{\pi''}$. Thus, from Lemma 5.2.1, we get $p|_{\mathcal{I}\cup\mathcal{O}\cup\mathcal{X}}, p''|_{\mathcal{I}\cup\mathcal{O}\cup\mathcal{X}} \in \mathcal{ST}_{rc}(\mathcal{S})$, $q'|_{\mathcal{I}\cup\mathcal{O}\cup\mathcal{X}} \in \mathcal{ST}_{rc}(\mathcal{S}^{\mu_j})$, and $p|_{\mathcal{I}} = q'|_{\mathcal{I}} = p''|_{\mathcal{I}}$.

Since the $\Pi[\pi \mapsto p, \pi' \mapsto q', \pi'' \mapsto p'']$ satisfies the whole formula and the antecedent, the trace assignment must also satisfy the consequent. That is, it must be the case that $q'|_{\mathcal{O}} \neq p''|_{\mathcal{O}}$ (Lemma 5.2.1). Since q' and p'' were chosen arbitrarily besides satisfying the antecedent and since symbolic traces that do not satisfy the antecedent are either not symbolic traces of the model or mutant, or have different input to p, we have shown $\{q \in \mathcal{ST}_{rc}(\mathcal{S}^{\mu_j}) \mid q|_{\mathcal{I}} = p|_{\mathcal{I}}\}|_{\mathcal{O}} \cap \{p'' \in \mathcal{ST}_{rc}(\mathcal{S}) \mid p''|_{\mathcal{I}} = p|_{\mathcal{I}}\}|_{\mathcal{O}} = \varnothing$, i.e. $\vec{i} \stackrel{\text{def}}{=} p|_{\mathcal{I}}$ is the input sequence which together with Lemma 3.3.1 shows that $\mathcal{S}^{\mu_j}$ is definitely killable.

Let $p \in \mathcal{ST}_{rc}(\mathcal{S})$ be a π-witness to $\mathcal{S}^{\bar{\mu}} \vDash \phi_3(\mathcal{I}, \mathcal{O}, j)$. First, we show that traces of $\mathcal{S}^{\mu_j}$ with inputs $p|_{\mathcal{I}}$ can not repeat before having a different output to p. Assume the contrary, i.e. there are $q \in \mathcal{ST}_{rc}(\mathcal{S}^{\mu_j})$ and $l < j \leq k$, such that $q|_{\mathcal{I}} = p|_{\mathcal{I}}, q[1,k]|_{\mathcal{O}} = p[1,k]|_{\mathcal{O}}$, and $q[l] = q[j]$. Trace q can be modified to a trace that loops between $q[l]$ and $q[j]$ indefinitely. This trace is a counter-example to $\mathcal{S}^{\mu_j}$ being definitely killable. Therefore, let $n-1$ be the finite length of the longest prefix of traces of $\mathcal{S}^{\mu_j}$ with equal output to p. Clearly, the test $t \stackrel{\text{def}}{=} (p[1,n]|_{\mathcal{I}\cup\mathcal{O}})^{\mathcal{M}}$ definitely kills $\mathcal{S}^{\mu_j}$. □

5.2.3 Mixed Determinism Model and Mutant

We now examine cases where the model is non-deterministic and the mutant is deterministic and vice versa. It should be noted that in practice it might not be known a-priori whether a model or mutant is really deterministic. In such cases, the hyperproperties $\phi_2(\mathcal{I}, \mathcal{O}, j)$ and $\phi_3(\mathcal{I}, \mathcal{O}, j)$ for non-deterministic mutants can be used to define and construct killing test cases, as their guarantees hold for deterministic mutants as well. Nevertheless, in this section, we present the weakest hyperproperties expressing potential and definite killability for mixed determinism cases.

To show potential killability of a non-deterministic mutant for a deterministic model, one needs to find a trace of the model ($\exists\pi$) such that there is a trace of the mutant with the same inputs ($\exists\pi'$) that eventually diverges in outputs, which is exactly formalized by the hyperproperty ϕ_1 above.

Proposition 5.2.4. Let the model $\mathcal{S}$ with inputs $\mathcal{I}$ and outputs $\mathcal{O}$ be deterministic and the mutant $\mathcal{S}^{\mu_j}$ be non-deterministic.

$$\mathcal{S}^{\bar{\mu}} \vDash \phi_1(\mathcal{I}, \mathcal{O}, j) \text{ iff } \mathcal{S}^{\mu_j} \text{ is potentially killable.}$$

Let p be a π-witness for $\mathcal{S}^{\bar{\mu}} \vDash \phi_1(\mathcal{I}, \mathcal{O}, j)$, then there exists $n \in \mathbb{N}$ such that the test $t \stackrel{\text{def}}{=} (p[1,n]|_{\mathcal{I}\cup\mathcal{O}})^{\mathcal{M}}$ potentially kills $\mathcal{S}^{\mu_j}$.

Proof. The proof can be conducted similar to the proof of Proposition 5.2.1. □

To show definite killing of a non-deterministic mutant of a deterministic model, one needs to find a trace of the model ($\exists\pi$) such that all traces of the mutant with the same inputs ($\forall\pi'$) eventually diverge in outputs, which is formalized via the hyperproperty ϕ_4 as follows:

$$\phi_4(\mathcal{I},\mathcal{O},j) \stackrel{\text{def}}{=} \exists\pi\forall\pi'\,\square[\text{mut} = 0]_\pi \wedge \Big(\square\big([\text{mut} = j]_{\pi'} \wedge (I_\pi \leftrightarrow I_{\pi'})\big) \rightarrow \Diamond\big(O_\pi \not\leftrightarrow O_{\pi'}\big)\Big)$$

Proposition 5.2.5. Let the model $\mathcal{S}$ with inputs $\mathcal{I}$ and outputs $\mathcal{O}$ be deterministic and the mutant $\mathcal{S}^{\mu_j}$ be non-deterministic.

$$\mathcal{S}^{\bar{\mu}} \vDash \phi_4(\mathcal{I},\mathcal{O},j) \text{ iff } \mathcal{S}^{\mu_j} \text{ is definitely killable.}$$

If p is a π-witness for $\mathcal{S}^{\bar{\mu}} \vDash \phi_4(\mathcal{I},\mathcal{O},j)$, then there exists $n \in \mathbb{N}$ such that the test $t \stackrel{\text{def}}{=} (p[1,n]|_{\mathcal{I}\cup\mathcal{O}})^{\mathcal{M}}$ definitely kills $\mathcal{S}^{\mu_j}$.

Proof. Assume that $\mathcal{S}^{\mu_j}$ is definitely killable. Since $\mathcal{S}$ is deterministic, for every input sequence, there is at most one trace with in $\mathcal{ST}_{rc}(\mathcal{S})$ with this input sequence. Therefore, from Lemma 3.3.1 it follows that there is an input sequence $\vec{i}$ and a unique trace $p \in \mathcal{ST}_{rc}(\mathcal{S})$ with $p|_{\mathcal{I}} = \vec{i}$ and $p|_{\mathcal{I}\cup\mathcal{O}} \notin \mathcal{ST}_{rc}(\mathcal{S}^{\mu_j})|_{\mathcal{I}\cup\mathcal{O}}$. Any trace assignment that maps π to p satisfies $\phi_4(\mathcal{I},\mathcal{O},j)$, since either the antecedent is violated by a trace $q \in \mathcal{ST}_{rc}(\mathcal{S}^{\mu_j})$ assigned to π' with different inputs, or the consequent is violated by a trace $q \in \mathcal{ST}_{rc}(\mathcal{S}^{\mu_j})$ assigned to π' with inputs $\vec{i}$ and outputs that can only be different to $p|_{\mathcal{O}}$.

Conversely, assume $\mathcal{S}^{\bar{\mu}} \vDash \phi_4(\mathcal{I},\mathcal{O},j)$. Let Π be a satisfying trace assignment that maps π to q and π' to p that also satisfies the antecedent, which is in fact satisfiable, since $\mathcal{S}^{\mu_j}$ is input-enabled. Clearly, p is a π-witness for $\square[\text{mut} = 0]_\pi$ and since q was chosen such that it satisfies the antecedent, q is a π'-witness for $\square[\text{mut} = j]_{\pi'}$. Thus, from Lemma 5.2.1, we get $p|_{\mathcal{I}\cup\mathcal{O}\cup\mathcal{X}} \in \mathcal{ST}_{rc}(\mathcal{S})$, $q|_{\mathcal{I}\cup\mathcal{O}\cup\mathcal{X}} \in \mathcal{ST}_{rc}(\mathcal{S}^{\mu_j})$, and $q|_{\mathcal{I}} = p|_{\mathcal{I}}$. Since Π satisfies the whole formula, it must be the case that Π also satisfies the consequent, i.e. $q|_{\mathcal{O}} \neq p|_{\mathcal{O}}$ (Lemma 5.2.1). Therefore, we can conclude $p|_{\mathcal{I}\cup\mathcal{O}} \notin \mathcal{ST}_{rc}(\mathcal{S}^{\mu_j})|_{\mathcal{I}\cup\mathcal{O}}$, which, as argued above, together with Lemma 3.3.1 is equivalent to definite killing in the deterministic model case.

The existence of a definitely killing test in case of finite $\mathcal{S}^{\mu_j}$ can be shown analogously to the proof of Proposition 5.2.3. □

Finally, to show killability of a deterministic mutant for a non-deterministic model, one needs to find a trace of the mutant ($\exists\pi$) such that all traces of the model with the same inputs ($\forall\pi'$) eventually diverge in outputs, which is already expressed via the hyperproperty ϕ_2 above.

Proposition 5.2.6. Let the model $\mathcal{S}$ with inputs $\mathcal{I}$ and outputs $\mathcal{O}$ be non-deterministic and the mutant $\mathcal{S}^{\mu_j}$ be deterministic

$$\mathcal{S}^{\bar{\mu}} \vDash \phi_2(\mathcal{I},\mathcal{O},j) \text{ iff } \mathcal{S}^{\mu_j} \text{ is killable.}$$

Let q be a π-witness for $\mathcal{S}^{\bar{\mu}} \vDash \phi_2(\mathcal{I},\mathcal{O},j)$, then there is $n \in \mathbb{N}$, such for the single trace $p \in \mathcal{ST}_{rc}(\mathcal{S})$ with $p|_{\mathcal{I}} = q|_{\mathcal{I}}$ the test $t \stackrel{\text{def}}{=} (p[1,n]|_{\mathcal{I}\cup\mathcal{O}})^{\mathcal{M}}$ kills $\mathcal{S}^{\mu_j}$.

Proof. Potential killing directly follows from the more restricted case in Proposition 5.2.2. Since $\mathcal{S}^{\mu_j}$ is deterministic, by Proposition 4.2.1 it is also definitely killable. The existence of a killing test can be shown analogously to the proof of Proposition 5.2.2. □

5.2.4 Locally Adaptive Tests

We can extend the hyperproperties presented above to force π-witnesses to have prefixes that are locally adaptive tests. Recall the definition of locally adaptive tests (Definition 3.2.3) that reasons over allowed outputs. Given a trace $\langle (s_1, i_1, o_1), \ldots, (s_n, i_n, o_n) \rangle \in \mathcal{T}_{rc}(\mathcal{M})$ of some model $\mathcal{M}$, an output a is allowed at step j if there is a trace $p \in \mathcal{T}_{rc}(\mathcal{M})$ that follows the test up to step j and then produces output a, i.e. $p = \langle (s_1, i_1, o_1), \ldots, (s_j, i_j, a), \ldots \rangle$.

In order to reason over allowed outputs, we introduce auxiliary indicator variables and restrict their values in π-witnesses of hyperproperties. Let Out be the set of all outputs. Remember that an output O of an STS is a mapping of output variables $\mathcal{O}$ to a range of output values. Therefore, Out is a set of mappings. We define locally allowed output indicators as the set of fresh Boolean variables $\mathcal{A} \stackrel{\text{def}}{=} \{a[O] \mid O \in Out\}$ as a subset of state variables $\mathcal{X}$ that are not used in the initial state or transition predicate.

We now strengthen the hyperproperties expressing killability, such that only assignments to these variables are allowed that reflect the semantics of allowed outputs. Unfortunately, these semantics are not expressible in HyperLTL, since they require to reason over all outgoing traces from intermediate states of an arbitrary trace. However, the property is expressible in HyperCTL*.

For an STS with a finite set of outputs Out and a HyperLTL formula ϕ of the form $\exists \pi \psi$, we define its locally adaptive test extension $\phi^{\mathcal{A}}$ as:

$$\phi^{\mathcal{A}} \stackrel{\text{def}}{=} \exists \pi \Big(\psi \wedge \bigwedge_{O \in Out} \square \mathrm{X} \Big(a[O]_\pi \leftrightarrow \big(\exists \pi' [\mathrm{mut} = 0]_{\pi'} \wedge (I_\pi \leftrightarrow I_{\pi'}) \wedge \bigwedge_{o \in \mathsf{AP}(O)} o_{\pi'} \wedge \bigwedge_{o \in \mathsf{AP}(\mathcal{O}) \setminus \mathsf{AP}(O)} \neg o_{\pi'} \big) \Big) \Big)$$

For example, for $\phi_3(\mathcal{I}, \mathcal{O}, j)$ the full extended formula is given as follows:

$$\phi_3(\mathcal{I},\mathcal{O},j)^{\mathcal{A}} \stackrel{\text{def}}{=}$$
$$\exists\pi\Big(\forall\pi'\forall\pi''\square[\text{mut}=0]_\pi \wedge \Big(\square\big([\text{mut}=j]_{\pi'} \wedge [\text{mut}=0]_{\pi''} \wedge (I_\pi \leftrightarrow I_{\pi'}) \wedge (I_\pi \leftrightarrow I_{\pi''})\big) \rightarrow$$
$$\Diamond\big(O_{\pi'} \not\leftrightarrow O_{\pi''}\big)\Big)\wedge$$
$$\bigwedge_{O\in Out} \square\mathrm{X}\Big(a[O]_\pi \leftrightarrow \big(\exists\pi'[\text{mut}=0]_{\pi'} \wedge (I_\pi \leftrightarrow I_{\pi'})\wedge$$
$$\bigwedge_{o\in\mathsf{AP}(O)} o_{\pi'} \wedge \bigwedge_{o\in\mathsf{AP}(\mathcal{O})\setminus\mathsf{AP}(O)} \neg o_{\pi'}\big)\big)\Big)$$

Likewise, this extension can be performed for $\phi_1(\mathcal{I},\mathcal{O},j)$ and $\phi_4(\mathcal{I},\mathcal{O},j)$. Note that the π path variable in $\phi_2(\mathcal{I},\mathcal{O},j)$ is constrained to evaluate to paths of the mutant. Thus, in order to leverage this transformation for $\phi_2(\mathcal{I},\mathcal{O},j)$, an additional existential quantifier picking one suitable trace of the original STS needs to be added to the formula.

We now show that models of these extensions contain locally adaptive tests.

Proposition 5.2.7. Let $\mathcal{S}^{\bar{\mu}}$ be a conditional mutant and let ϕ be a HyperLTL formula of the form $\exists\pi\psi$ such that $\mathcal{S}^{\bar{\mu}} \vDash \phi$ and some finite prefix of a π-witness to $\mathcal{S}^{\bar{\mu}} \vDash \phi$ is a linear test, then $\mathcal{S}^{\bar{\mu}} \vDash^* \phi^{\mathcal{A}}$ and some finite prefix of the trace component of a π^*-witness for $\mathcal{S}^{\bar{\mu}} \vDash^* \phi^{\mathcal{A}}$ is a locally adaptive test.

Proof. Let $p \in \mathcal{ST}_{rc}(\mathcal{S})$ be a π-witness to $\mathcal{S}^{\bar{\mu}} \vDash \phi$ and let $t \stackrel{\text{def}}{=} p[1,n]$ be the finite prefix that is a linear test. Since variables $a[O]$ are unconstrained by the STS, we can assume that the valuations of these variables in p are chosen such that p (together with its states) constitutes a π^*-witness for $\mathcal{S}^{\bar{\mu}} \vDash^* \phi^{\mathcal{A}}$. To show that t is a locally adaptive test, up to symbolic trace correspondence shown in Lemma 3.3.1, it needs to be the case that for every $j \in [1,n]$ and every $O \in Out$ it is the case that $t[j]$ at $a[O]$ evaluates to $\top$ if and only if there exists a trace $p' \in \mathcal{ST}_{rc}(\mathcal{S})$ with $p'[1,i-1] = t[1,i-1]$, $p'[j]|_{\mathcal{I}} = t[j]|_{\mathcal{I}}$, and $p'[j]|_{\mathcal{O}} = O$. Path p is chosen such that in every step $j-1$ and output O, p evaluates $a[O]$ to $\top$ in its successor state if and only if from the current state there a path of the original system p_j^O whose next state exactly has input $p[j]|_{\mathcal{I}}$ and output O. Therefore, paths p_j^O for every $O \in Out$, prepended with the prefix of p up to j, are witnesses to this property. □

Unfortunately, to the best of the knowledge of the authors, there currently does not exists a model checker for HyperCTL*. However, the problem was shown to be decidable in [FRS15], although its complexity grows exponentially in the number of quantifier alternations. Therefore, on top of providing formal semantics for locally adaptive tests, the encoding can be leveraged in practice soon as a HyperCTL* model checker emerges.

5.3 Non-deterministic Models in Practice

Checking the validity of the hyperproperties in Section 5.2 for a given model and mutant enables test-case generation. Unfortunately, the model checkers for HyperLTL or HyperCTL* are still in their infancy. To the best of our knowledge, MCHYPER [FRS15] is the only currently available HyperLTL model checker and there is no HyperCTL* model checker. The latter is hindered by the prohibitively expensive step of complementing ω-automata [FRS15]. Furthermore, HyperLTL formulas with quantifier alternation, such as killability defining formulas $\phi_2(\mathcal{I}, \mathcal{O}, j)$ and $\phi_3(\mathcal{I}, \mathcal{O}, j)$ for non-deterministic models, can currently not be handled with the available version of the tool. In a web-based version of MCHYPER such formulas can be handled via a combination with a reactive synthesis tool, as described in [CFST19]. To remedy this issue and to obtain test cases for non-deterministic systems, in this section, we propose two solutions.

Firstly, we present a transformation that makes non-determinism *controllable* by means of additional inputs and yields a deterministic STS. The quantifier alternation free formula $\phi_1(\mathcal{I}, \mathcal{O}, j)$ can be model checked over the transformed model. The result is an over-approximation of killability in the sense that the resulting test cases only kill some non-deterministic mutant if non-determinism can also be controlled in the system under test. However, if equivalence can be established for the transformed model, then the non-deterministic mutant is also equivalent. In Section 5.3.1 we define the transformation formally and prove its properties. In Section 5.3.2 we show how the transformation can be done syntactically in practice.

Secondly, we propose a bounded model checking approach for $\phi_2(\mathcal{I}, \mathcal{O}, j)$ and $\phi_3(\mathcal{I}, \mathcal{O}, j)$ via an encoding into a SMT satisfiability problem. This problem can be solved with off-the-shelf solvers such as the SMT solver Z3 [DMB08] or the first-order logic solver Vampire [KV13].

5.3.1 Controlling Non-determinism in Symbolic Transition Systems

The essential idea of our transformation is to introduce an additional *input* (represented by an auxiliary variable nd) that enables the control of non-deterministic choices in a conditional mutant $\mathcal{S}^{\bar{\mu}}$ with finite non-deterministic branching. The new input is used carefully to ensure that choices are consistent for the model and the mutant encoded in $\mathcal{S}^{\bar{\mu}}$. Without loss of generality, we assume that variable nd has a finite range sufficiently large to encode the non-deterministic choices in $\alpha^{\bar{\mu}}$ and $\delta^{\bar{\mu}}$. In particular, for every output O and successor state X', we assume that there exists a unique value $n_{O,X'}$ in the range of nd. Moreover, we add a fresh Boolean variable x^τ to $\mathcal{X}$ that we use to encode a fresh initial state. Furthermore, let $\psi(X)$, $\psi(X, I)$, and $\psi(O, X')$ be formulas that are uniquely satisfied by any state X, any state/input pair (X, I), and any output/successor state pair (O, X') respectively.

Given conditional mutant $\mathcal{S}^{\bar{\mu}} \stackrel{\text{def}}{=} \langle \mathcal{I}, \mathcal{O}, \mathcal{X} \cup \{\text{mut}\}, \alpha^{\bar{\mu}}, \delta^{\bar{\mu}} \rangle$, we define its controllable counterpart $D(\mathcal{S}^{\bar{\mu}}) \stackrel{\text{def}}{=} \langle \mathcal{I} \cup \{nd\}, \mathcal{O}, \mathcal{X} \cup \{\text{mut}\} \cup \{x^\tau\}, D(\alpha^{\bar{\mu}}), D(\delta^{\bar{\mu}}) \rangle$. We initialize $D(\delta^{\bar{\mu}}) \stackrel{\text{def}}{=} \delta^{\bar{\mu}}$ and incrementally add constraints as described below.

Non-deterministic initial state predicate: Let X be an arbitrary, fixed state. The unique fresh initial state is $X^\tau \stackrel{\text{def}}{=} X[x^\tau \mapsto \top]$, which, together with an empty output, we enforce by the new

initial conditions predicate:

$$D(\alpha^{\bar{\mu}}) \stackrel{\text{def}}{=} \psi(X^{\tau})$$

We add the conjunct $\neg\psi(X^{\tau}) \rightarrow \neg x^{\tau\prime}$ to $D(\delta^{\bar{\mu}})$, in order to force x^{τ} evaluating to $\bot$ in all states other than X^{τ}. In addition, we add transitions from X^{τ} to all pairs of initial states in $\alpha^{\bar{\mu}}$. For each state X such that $X \vDash \alpha^{\bar{\mu}}$, we add the following formula conjunctively to $D(\delta^{\bar{\mu}})$, introducing transitions from the new initial state for every input with empty output to every original initial state:

$$(\psi(X^{\tau}) \wedge nd = nd_{O_{\eta},X'}) \rightarrow \psi(O_{\eta}, X')$$

Non-deterministic transitions: For each transition $X \xrightarrow{I,O} X'$, we fix non-determinism by adding the following formula conjunctively to $\delta^{\bar{\mu}}$:

$$(\psi(X, I) \wedge nd = nd_{O,X'}) \rightarrow \psi(O, X')$$

The proposed transformation has the following properties:

Proposition 5.3.1. Let $\mathcal{S}^{\bar{\mu}} = \langle \mathcal{I}, \mathcal{O}, \mathcal{X} \cup \{\mathrm{mut}\}, \alpha^{\bar{\mu}}, \delta^{\bar{\mu}} \rangle$ be a conditional mutant for mutation setting $\{\mu_1, \ldots, \mu_n\}$.

1. $D(\mathcal{S}^{\bar{\mu}})$ is deterministic (up to mut).
2. $\mathcal{ST}_{rc}(\mathcal{S}^{\bar{\mu}})|_{\mathcal{X}\cup\{\mathrm{mut}\}\cup\mathcal{I}\cup\mathcal{O}} \subseteq \mathcal{ST}_{rc}(D(\mathcal{S}^{\bar{\mu}}))[1,\infty]|_{\mathcal{X}\cup\{\mathrm{mut}\}\cup\mathcal{I}\cup\mathcal{O}}$.
3. $D(\mathcal{S}^{\bar{\mu}}) \not\vDash \phi_1(\mathcal{I}, \mathcal{O}, j)$ then $\mathcal{S}^{\mu_j}$ is equivalent.

Proof. Statement 1: We show $D(\mathcal{S}^{\bar{\mu}})$ is deterministic (up to mut). $D(\mathcal{S}^{\bar{\mu}})$ has a unique (up to mut) initial state X^{τ}, since we fix $D(\alpha^{\bar{\mu}})$ to be satisfiable by exactly this state.

Let $X \xrightarrow{I,O_1} X_1'$ and $X \xrightarrow{I,O_2} X_2'$ be a non-deterministic transition of $\mathcal{S}^{\bar{\mu}}$, such that $X_1'(\mathrm{mut}') = X_2'(\mathrm{mut}')$. First, note that the value of $x^{\tau\prime}$ is fixed to $\bot$, both in X_1' and X_2'. Furthermore, we have that $X, I[nd \mapsto nd_{O_1,X_1'}], O_1, X_1' \vDash D(\delta^{\bar{\mu}})$, because $X, I, O_1, X_1' \vDash \delta^{\bar{\mu}}$ and both the antecedent as well as the consequent of the additional constraint $(\psi(X, I) \wedge nd = nd_{O_1,X_1'}) \rightarrow \psi(O_1, X_1')$ in $D(\delta^{\bar{\mu}})$ are satisfied. Furthermore, we have that $X, I[nd \mapsto nd_{O_1,X_1'}], O_2, X_2' \nvDash D(\delta^{\bar{\mu}})$, since the antecedent of the above constraint is satisfied, whereas the consequent is not. The argument for transition $X \xrightarrow{I,O_2} X_2'$ is symmetric. In summary, in the transformed symbolic transition system $D(\mathcal{S}^{\bar{\mu}})$, it is the case that $X \xrightarrow{I[nd\mapsto nd_{O_1,X_1'}],O_1} X_1'$ and $X \xrightarrow{I[nd\mapsto nd_{O_2,X_2'}],O_2} X_2'$ and neither $X \xrightarrow{I[nd\mapsto nd_{O_1,X_1'}],O_2} X_2'$ nor $X \xrightarrow{I[nd\mapsto nd_{O_2,X_2'}],O_1} X_1'$. Thus, we showed that non-determinism due to tuples (X, I, O_1, X_1) and (X, I, O_2, X_2) is resolved in the transformed system. This argument can be applied to any such pair, showing that $D(\mathcal{S}^{\bar{\mu}})$ is deterministic (up to mut).

Statement 2: To show this statement, we show that every initial state X_0 of $\mathcal{S}^{\bar{\mu}}$ is the target of a transition of the unique initial state X^τ of $D(\mathcal{S}^{\bar{\mu}})$ and that every transition $X \xrightarrow{I,O} X'$ is preserved in $D(\mathcal{S}^{\bar{\mu}})$.

The former point is true, due to constraint $(\psi(X^\tau) \wedge nd = nd_{O_\eta,X'_0}) \rightarrow \psi(O_\eta, X'_0)$ that is included in $D(\delta^{\bar{\mu}})$. For the latter point, consider a tuple $\langle X, I[nd \mapsto nd_{O,X'}], O, X' \rangle$ (with respect to $D(\mathcal{S}^{\bar{\mu}})$). This tuple satisfies both the antecedent and consequent of the newly added constraint $(\psi(X, I) \wedge nd = nd_{O,X'}) \rightarrow \psi(O, X')$. Furthermore, every other newly added constraint of the form $(\psi(X_1, I_1) \wedge nd = nd_{O_1,X'_1}) \rightarrow \psi(O_1, X'_1)$ is satisfied by the tuple, either because $X = X_1$, $I = I_1$, $O = O_1$, and $X' = X'_1$ (thus in particular $nd_{O_1,X'_1} = nd_{O,X'}$) such that both the antecedent or the consequent are satisfied, or any of the equalities does not hold and the antecedent is not satisfied.

Statement 3: $\not\models \phi_1(\mathcal{I}, \mathcal{O}, j)$ then $\mathcal{S}^{\mu_j}$ is equivalent is a direct consequence of the statements about symbolic traces, since $\not\models \phi_1(\mathcal{I}, \mathcal{O}, j)$ shows no trace in $\mathcal{ST}_{rc}(D(\mathcal{S}^{\bar{\mu}}))$ is a witness to killing the mutant. Since traces of $\mathcal{S}^{\bar{\mu}}$ are included in (the projection of) this set, there can not be a trace in $\mathcal{ST}_{rc}(\mathcal{S}^{\bar{\mu}})$ that is a witness to killing the mutant. □

In summary, the transformed model is deterministic, since we enforce unique initial valuations and make non-deterministic transitions controllable through input nd. Since we only add transitions or augment existing transitions with input nd, every transition $X \xrightarrow{I,O} X'$ of $\mathcal{S}^{\bar{\mu}}$ is still present in $D(\mathcal{S}^{\bar{\mu}})$ (when input nd is disregarded). The potential additional traces of Statement 2 originate from transitions added due to implicit input-enabling.

Statement 3 shows what can be achieved by model checking the quantifier alternation free formula ϕ_1 over the transformed controllable determinism STS $D(\mathcal{S}^{\bar{\mu}})$. Equivalent mutants of this system are also equivalent in the non-deterministic version. Killability purported by ϕ_1, however, could be an artifact of the transformation: determinization potentially deprives the model of its ability to match the output of the mutant by deliberately choosing a certain non-deterministic transition. Test cases can therefore only be considered killing under the assumption that non-determinism can be controlled by the tester. In Example 3.2.1, we present an equivalent mutant that serves only tea. This mutant is killable after transforming the mutant as well as the corresponding model, since upon fixing the non-deterministic outcome after the request to serve coffee, the model will respond with output `coffee`, whereas the mutant will reject the input corresponding this non-deterministic choice, i.e. respond with η. Therefore, our transformation only allows us to provide a lower bound for the number of equivalent non-deterministic mutants.

5.3.2 Controlling Non-determinism in Modeling Languages

The transformation outlined in Section 5.3.1 is purely theoretical and often infeasible in practice, since all states and inputs have to be enumerated. However, an analogous result can often be achieved by modifying the syntactic constructs of the underlying modeling language that introduce non-determinism, namely:

- *Non-deterministic assignments.* Non-deterministic choice over a finite set of elements $\{x'_1, \ldots x'_n\}$, as provided by SMV [McM92b], can readily be converted into a case-switch construct over nd. More generally, explicit non-deterministic assignments `x := *` to state variables `x` [Nel89] can be controlled by assigning the value of nd to `x`.

- *Non-deterministic schedulers.* Non-determinism introduced by concurrency can be controlled by introducing input variables that control the scheduler (as proposed in [LR09] for bounded context switches).

In case non-determinism arises through variables under-specified in transition relations, these variable values can be made inputs as suggested by Section 5.3.1. In general, however, identifying under-specified variables automatically is non-trivial.

Example 5.3.1. Consider again the SMV code in Figure 5.1a, for which non-determinism can be made controllable by replacing 1. with 2. and adding **`init`**`(nd):={0,1}`, where

1. **`if`**`(in=request&water>0):{coffee,tea}`
2. **`if`** `(nd=0&in=request&water>0):coffee`
   `elif``(nd=1&in=request&water>0):tea`

Similarly, the STS representation of the beverage machine, given in Example 3.3.1, can be transformed by replacing the first two rules by the following two rules:

$$(\texttt{nd=0} \wedge \texttt{water>0} \wedge \texttt{in=request} \wedge \texttt{out=coffee} \wedge \texttt{water'=water-1}) \vee$$
$$(\texttt{nd=1} \wedge \texttt{water>0} \wedge \texttt{in=request} \wedge \texttt{out=tea} \wedge \texttt{water'=water-1}) \vee$$

Finally, the results of the transformation is presented in Figure 5.5 on a case study.

5.3.3 Encoding Bounded Killability into SMT

Another way of solving killability properties with quantifier alternation is to leverage the first order expressibility of HyperLTL (proven in [FZ17]) and to encode the strong potential and definite killability problems into a suitable fragment of first order logic. Let $\mathcal{S}^{\bar{\mu}} \stackrel{\text{def}}{=} \langle \mathcal{I}, \mathcal{O}, \mathcal{X} \cup \{\text{mut}\}, \alpha^{\bar{\mu}}, \delta^{\bar{\mu}} \rangle$ be a conditional mutant for mutation setting $\{\mu_1, \ldots, \mu_n\}$, where for ease of presentation, we abbreviate $\alpha^{\bar{\mu}}$ with α and $\delta^{\bar{\mu}}$ with δ throughout this section. We describe an SMT encoding of potential and definite killability into a bounded (up to fixed bound k) satisfiability problem in a logic that contains the logic of the symbolic transition system, as well as quantification over the ranges of state and output variables. The idea is to create copies of each variable and each step and each mutant as well as the model. Furthermore, the transition relation is replicated for each step, using the respective step variables. Reachable states and outputs are expressed by universally quantifying over variable values and checking whether the respective variable assignment satisfies the initial state-, as well as, the step-wise transition relation- predicate.

Let $\mathcal{I} = \{i_0, \ldots, i_{m_i}\}$, $\mathcal{O} = \{o_0, \ldots, o_{m_o}\}$, and $\mathcal{X} = \{x_0, \ldots, x_{m_x}\}$. For each variable $i \in \mathcal{I}$ and $l = 0, \ldots, k$ we create new variables $i[l]$ with the same range as i. For each $j = 0, \ldots, n$, each variable $v \in \mathcal{O}$, and each $l = 0, \ldots, k$ as well as each $v \in \mathcal{X}$, and each $l = 0, \ldots, k+1$, we create new variables $v[l, j]$ with the same ranges as v. For a formula ψ and each $j = 0, \ldots, n$, let $\psi[l, j]$ be the formula that results from replacing each variable $v \in \mathcal{I} \cup \mathcal{O} \cup \mathcal{X} \setminus \{\mathrm{mut}\}$ with $v[l, j]$, each variable $v' \in \mathcal{X}'$ with $v[l+1, j]$, and mut as well as mut' with j. In particular, conditionally mutated sub-formulas of the form $(\mathrm{mut} = j_0 \wedge \gamma^{\mu_{j_0}}) \vee (\mathrm{mut} \neq j \wedge \gamma)$ are transformed to $(j = j_0 \wedge (\gamma^{\mu_{j_0}})[l, j]) \vee (j \neq j_0 \wedge \gamma[l, 0])$ for which clearly the mutated branch is satisfiable if and only if $j = j_0$.

We can encode trace prefixes up to k steps of the model respectively mutant j as satisfying assignments of the following formulas with free input-, output-, and state- variables (original respectively mutated versions):

$$\phi_0^{\mathcal{S}^{\bar{\mu}}} \stackrel{\text{def}}{=} \alpha[0,0] \wedge \bigwedge_{0 \leq l \leq k} \delta[l,0] \qquad \phi_j^{\mathcal{S}^{\bar{\mu}}} \stackrel{\text{def}}{=} \alpha[0,j] \wedge \bigwedge_{0 \leq l \leq k} \delta[l,j]$$

We encode potential killing linear tests for mutant j of length k as models of the following formula with free inputs, original output- and state variables, quantifying over all mutated outputs $o_0, \ldots, o_{m_o}$ and state variables $x_0, \ldots, x_{m_x}$ as well as steps $0, \ldots, k+1$:

$$\begin{aligned}\phi_{pk:j}^{\mathcal{S}^{\bar{\mu}}} \stackrel{\text{def}}{=} \forall\, & o_0[0,j], \ldots, o_s[t,j], \ldots, o_{m_o}[k,j], x_0[0,j], \ldots, x_s[t,j], \ldots, x_{m_x}[k+1,j]. \\ & \alpha[0,0] \wedge \bigwedge_{0 \leq l \leq k} \delta[l,0] \wedge \Big((\alpha[0,j] \wedge \bigwedge_{0 \leq l \leq k} \delta[l,j]) \rightarrow \bigvee_{0 \leq m \leq m_o} o_m[k,j] \neq o_m[k,0] \Big)\end{aligned}$$

Likewise, we encode definitely killing linear tests for mutant j of length k as models of the following formula with free inputs, quantifying over all model and mutated outputs $o_0, \ldots, o_{m_o}$ and state variables $x_0, \ldots, x_{m_x}$ as well as steps $0, \ldots, k$:

$$\begin{aligned}\phi_{dk:j}^{\mathcal{S}^{\bar{\mu}}} \stackrel{\text{def}}{=} \forall\, & o_0[0,0], \ldots, o_s[t,0], \ldots, o_{m_o}[k,0], x_0[0,j], \ldots, x_s[t,j], \ldots, x_{m_x}[k+1,j], \\ & o_0[0,j], \ldots, o_s[t,j], \ldots, o_{m_o}[k,j], x_0[0,j], \ldots, x_s[t,j], \ldots, x_{m_x}[k+1,j]. \\ & (\alpha[0,j] \wedge \bigwedge_{0 \leq l \leq k} \delta[l,j] \wedge \alpha[0,0] \wedge \bigwedge_{0 \leq l \leq k} \delta[l,0]) \rightarrow \bigvee_{0 \leq m \leq m_o} o_m[k,j] \neq o_m[k,0]\end{aligned}$$

In the following proposition, we prove the correctness of the encoding.

Proposition 5.3.2. Let $\mathcal{S}^{\bar{\mu}}$ be a conditional mutant for mutation setting $\{\mu_1, \ldots, \mu_n\}$ and let $j \in [1, n]$, then

1. $\phi_0^{\mathcal{S}^{\bar{\mu}}}$ is satisfiable *iff* there is trace $p \in \mathcal{ST}_{rc}(\mathcal{S})$ of length at least k.
2. $\phi_j^{\mathcal{S}^{\bar{\mu}}}$ is satisfiable *iff* there is trace $p \in \mathcal{ST}_{rc}(\mathcal{S}^{\mu_j})$ of length at least k.
3. $\phi_{pk:j}^{\mathcal{S}^{\bar{\mu}}}$ is satisfiable *iff* there is a linear test t for $\mathcal{S}$ of length k potentially killing $\mathcal{S}^{\mu_j}$.

4. $\phi^{\mathcal{S}^{\bar{\mu}}}_{dk:j}$ is satisfiable *iff* there is a linear test t for $\mathcal{S}$ of length k definitely killing $\mathcal{S}^{\mu_j}$.

Proof. Statement 1: A satisfying assignment for $\phi_0(\mathcal{S}^{\bar{\mu}})$ is an assignment of stepwise copies of input, the model's output and the model's state variables up to step k that satisfies the initial state predicate and the transition predicate in each step. Clearly, such an assignment corresponds to the prefix of trace $p \in \mathcal{ST}_{rc}(\mathcal{S})$ of length k.

Statement 2 can be shown analogously to Statement 1.

Statement 3: Since $\phi_0(\mathcal{S}^{\bar{\mu}})$ is a conjunct of $\phi_{pk:j}(\mathcal{S}^{\bar{\mu}})$ it needs to be satisfied by any satisfying assignment of $\phi_{pk:j}(\mathcal{S}^{\bar{\mu}})$. Thus, due to Statement 1, every satisfying assignment of $\phi_{pk:j}(\mathcal{S}^{\bar{\mu}})$ encodes a prefix t of some trace $p \in \mathcal{ST}_{rc}(\mathcal{S})$ of length k and every such prefix is encoded in a satisfying assignment. Furthermore, every extension of such a satisfying assignment via assignments of the universally quantified mutant output- and mutant state variables that encodes a prefix of some trace of $q \in \mathcal{ST}_{rc}(\mathcal{S}^{\mu_j})$ with the same input values as p in the first k steps satisfies the antecedent of the implication and thus needs to satisfy the consequent. Due to Statement 2 every trace of $\mathcal{S}^{\mu_j}$ is captured via such an assignment. Therefore, there exists a satisfying assignment corresponding to trace prefix t of the formula if and only if every trace $q \in \mathcal{ST}_{rc}(\mathcal{S}^{\mu_j})$ with $q|_{\mathcal{I}}[0,k] = t|_{\mathcal{I}}$ is such that $q|_{\mathcal{O}}[k] \neq t|_{\mathcal{O}}[k]$, which together with Lemma 3.3.1 is equivalent to t potentially killing $\mathcal{S}^{\mu_j}$.

Statement 4 can be shown analogously to Statement 3 with the exception that a satisfying assignment only encodes a sequence of inputs. Extensions of that satisfying assignment of universally quantified variables that satisfy the antecedent encode a trace in the model and a trace in the mutant with equal input. Since also the consequent needs to be satisfied by such extensions, their outputs must differ in step k, showing that every trace of the model with the sequence of inputs given by the satisfying assignment corresponds to a definitely killing linear test of length k (up to trace and symbolic trace correspondence shown in Lemma 3.3.1). □

Example 5.3.2. Consider again the transition relation $\delta^{\bar{\mu}}$ of the conditional mutant presented in Example 4.1.2, recited here:

$$
\begin{aligned}
&\big((\texttt{in=request} \wedge \texttt{water>0} \wedge \texttt{out=coffee} \wedge \texttt{water}' = \texttt{water-1}) \vee \\
&\ (\texttt{in=request} \wedge \texttt{out=tea} \wedge \texttt{water}' = \texttt{water-1} \wedge \\
&\qquad (\texttt{(mut=1} \wedge \texttt{false)} \vee \texttt{(mut}\neq\texttt{1} \wedge \texttt{water>0)))} \vee \\
&\ (\texttt{in=refill} \wedge \texttt{water=0} \wedge \texttt{out=full} \wedge \\
&\qquad (\texttt{(mut=2} \wedge \texttt{water}' = \texttt{1)} \vee \texttt{(mut}\neq\texttt{2} \wedge \texttt{water}' = \texttt{2))}) \vee \\
&\ (\texttt{in=request} \wedge \texttt{water=0} \wedge \texttt{out=}\eta \wedge \texttt{water}' = \texttt{water})\big) \wedge \\
&\texttt{mut}' = \texttt{mut}
\end{aligned}
$$

We present $\delta[l, 2]$, i.e. the encoding of the $l'th$ step of the transition predicate for mutant $\mathcal{S}^{\mu_2}$:

$$
\begin{aligned}
&\big(\texttt{in[l]=request} \wedge \texttt{water[l,2]>0} \wedge \texttt{out=coffee} \wedge \\
&\qquad \texttt{water[l+1,2]=water[l,2]-1}\big) \vee \\
&\big(\texttt{in[l]=request} \wedge \texttt{out=tea} \wedge \texttt{water[l+1,2]=water[l,2]-1} \wedge \\
&\qquad (\texttt{(2=1} \wedge \texttt{false)} \vee \texttt{(2}\neq\texttt{1} \wedge \texttt{water[l,2]>0))}\ \big) \vee \\
&\big(\texttt{in[l]=refill} \wedge \texttt{water[l,2]=0} \wedge \texttt{out=full} \wedge \\
&\qquad (\texttt{(2=2} \wedge \texttt{water[l+1,2]=1)} \vee \texttt{(2}\neq\texttt{2} \wedge \texttt{water[l+1,2]=2))}\big) \vee \\
&\big(\texttt{in[l]=request} \wedge \texttt{water[l,2]=0} \wedge \texttt{out=}\eta \wedge \texttt{water[l+1,2]=water[l,2]}\big) \wedge \\
&\texttt{2=2}
\end{aligned}
$$

In order to evaluate the scalability of this method, we encoded a parametrized version of the beverage machine, together with $\phi^{\mathcal{S}^{\bar{\mu}}}_{dk:j}$ in the SMTlib format and gave it to the Z3 SMT solver (version 4.8.7). The benchmark encoding is parametrized with the bound k for which a definitely killing test can be found (k = 5 in the running example instance, corresponding to input sequence $\langle$`request, request, refill, request, request`$\rangle$), which can be controlled via the capacity of the water tank (2 in the running example instance). The encoding and a script to create parametrized versions of it can be found in [ben]. We ran this proof of concept demonstration on a virtual machine with one Intel i7 core at 2.8 GHz and 10GB of RAM.

The instance with bound 12 is solved within 20 seconds. After that, there seems to be a steep increase in complexity. For the instance with bound 13, Z3 returns unknown after 7 minutes with an error indicating model-based quantifier instantiation did not find a model after 1000 attempts. Unsurprisingly, the large amount of universal quantification poses a challenging problem to Z3.

5.4 Mutation Testing with Hyperproperties Experiments

In this section, we present an experimental evaluation of the test generation via HyperLTL model checking method. We start by discussing the deployed tool-chain. Thereafter, we show a validation of the method on one case study via an action system formulation and the test case generation methods described in the later chapters of this thesis. Finally, we present quantitative results on a broad range of generic models.

5.4.1 Toolchain

Figure 5.2 shows the toolchain that we use to produce test suites for models encoded in the modeling languages Verilog and SMV. Verilog models are deterministic, whereas SMV models can be non-deterministic.

Variable Annotation

As a first step, we annotate variables as inputs and outputs. These annotations were added manually for Verilog, and heuristically for SMV (partitioning variables into outputs and inputs).

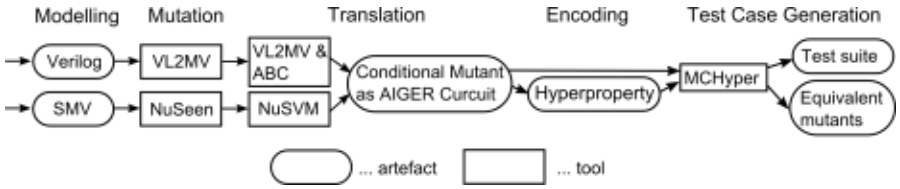


Figure 5.2: The tool pipeline of our experiments.

Mutation and Transformation

We produce conditional mutants via a mutation engine. For Verilog, we implemented our own mutation engine into the open source Verilog compiler VL2MV [CYB93]. We use standard mutation operators, such as replacing arithmetic operators, Boolean relations. The list of mutation operators used for Verilog can be found in the Table 4.1. For SMV models, we use the NuSeen SMV framework [AGR15, AGR17], which includes a mutation engine for SMV models. The mutation operators used for SMV are summarized in Table 4.2 and explained in detail in [AGR15]. We implemented the non-determinism controlling transformation presented in Section 6.1.4 into NuSeen and applied it to conditional mutants.

Translation

The resulting conditional mutants from both modeling formalisms are translated into AIGER circuits [BHW11]. AIGER circuits are essentially a compact representation for finite models. The formalism is widely used by model checkers. For the translation of Verilog models, VL2MV and the ABC model checker are used. For the translation of SMV models, NuSMV is used.

Test Suite Creation

We obtain a test suite via HyperLTL model checking $\neg\phi_1(\mathcal{I},\mathcal{O})$ on conditional mutants using the MCHYPER model checker. Tests are obtained as counter-examples, which are finite prefixes of π-witnesses to $\phi_1(\mathcal{I},\mathcal{O})$. In case we can not find a counter-example, and use a complete model checking method, the mutant is provably equivalent.

5.4.2 Car Alarm System (CAS) Case Study

Figure 5.3 depicts a model of a car alarm system. Variants of the model were studied in the model-based test case generation literature [AJT14, ABJ+15b, FKS+17] as well as in Section 6.3. For ease of presentation, we only depict the relevant inputs and omit loops for rejected inputs. The model includes timing sensitive transitions, which are modeled via an input wait and non-deterministic and discrete time ticks. The passed time of the respective transitions is written below the transition, which is an extra annotation that is there for readability, but does not have semantic meaning. Ranges of time sensitive transitions that lead to equivalent successor states (in terms of outgoing transitions) are subsumed for ease of presentation. For example, one state near the bottom right of the figure has 269 successor states and non-deterministic transitions for input wait that have empty output as well as one successor state for input wait that has output flash_off.

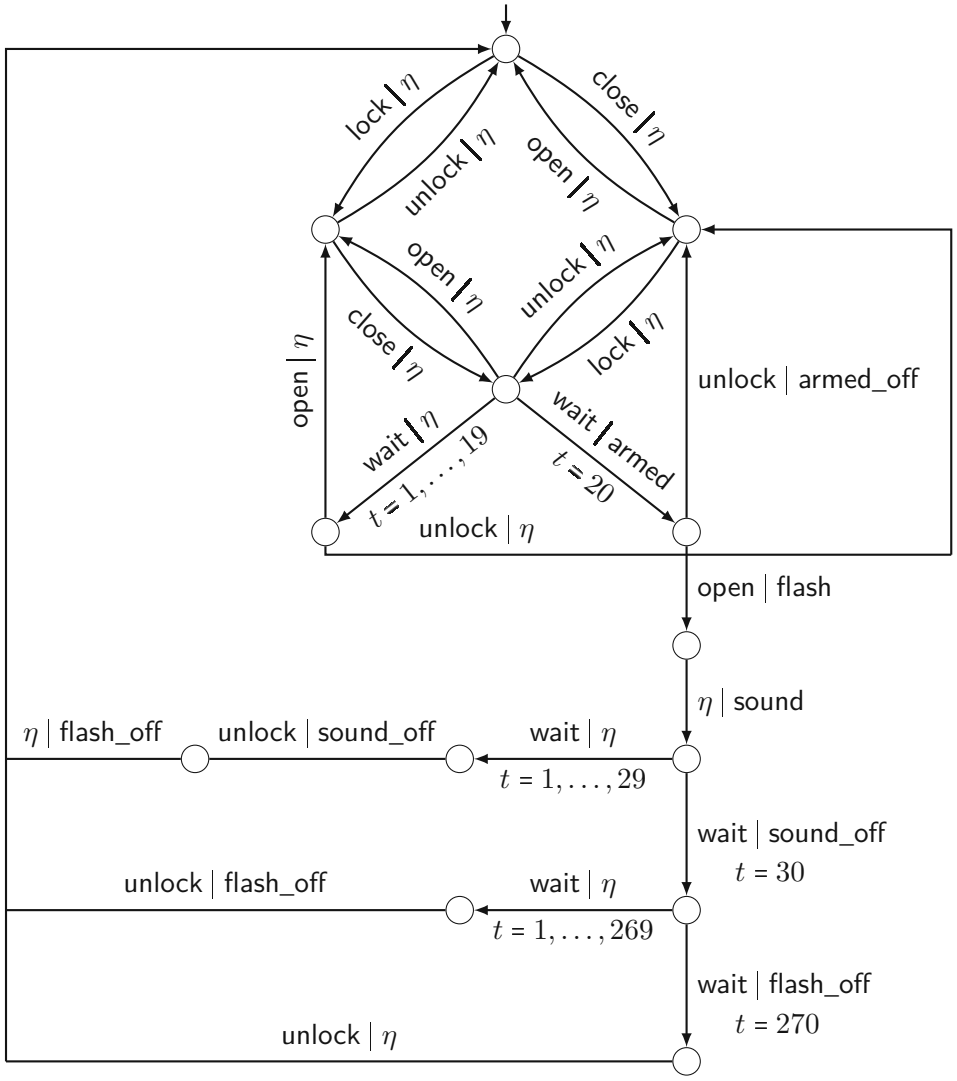
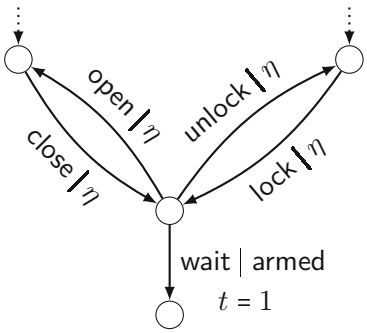


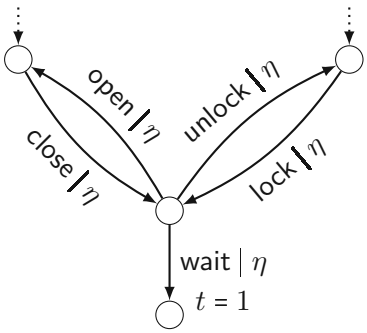
Figure 5.3: The non-deterministic timed car alarm system model.

The modeled car can be opened, closed, locked, and unlocked. Initially the car is open and unlocked. Once the car is closed and locked, after some time (20 clock ticks in the depicted instantiation) the car enters an armed state. In that armed state, if it is opened before it is unlocked, a visual (flash) and an acoustic (sound) alarm are triggered. After some specified time (30 clock ticks in the depicted instantiation) the visual alarm stops. Then after some more time (270 clock ticks in the depicted instantiation) the acoustic alarm also stops. At any time, the alarms can be turned off by unlocking the car.

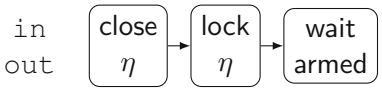
We can tune the degree of non-determinism of the model by adjusting the timers of time triggered events. In the following, we discuss mutants for deterministic and non-deterministic cases.



(a) A deterministic model.



(b) A deterministic mutant.



(c) A killing linear test.

(d) The spurious test response of the mutant.

Figure 5.4: A killing example for the deterministic case.

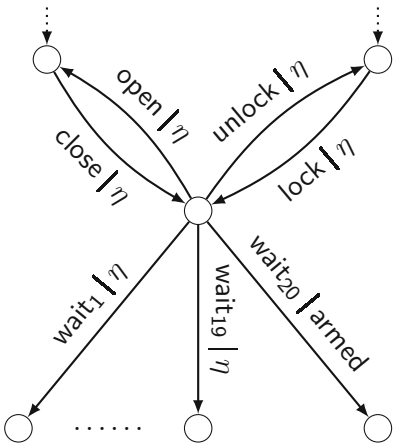
Deterministic Case

In case all timers of time triggered events are 1, the model is deterministic. In this case, we can study mutations on non time-triggered transitions. For example, we can introduce a mutation that disables some output. In the Mealy machine representation of the model, this amounts to replacing the output with η. In a syntactic representation of the model, this amounts to replacing the condition of a branch by false.

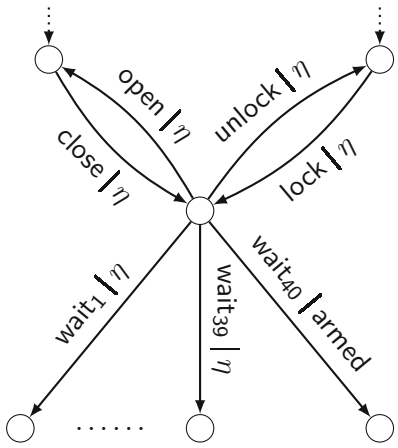
We depict the relevant parts of a deterministic version of the car alarm system model in Figure 5.4a and a mutant with a faulty arming mechanism Figure 5.4b. The mutant does not enter the armed state after the car is locked and closed. The test depicted in Figure 5.4c kills this mutant, as can be seen in the response of the mutant to the test in Figure 5.4d.

Non-deterministic Case

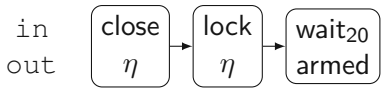
In case some timer of time triggered events is non-zero, the model is non-deterministic. As discussed in Section 6.1.4, in order to deal with non-deterministic models in practice, we need to make non-determinism controllable. We depict the relevant parts the transformed non-deterministic version of the car alarm system model in Figure 5.5a and a mutant that doubles the time trigger for entering the armed state in Figure 5.5b. Note that due to transformation making non-determinism controllable, in contrast to time manifesting in hidden non-determinism in Figure 5.3, time is explicitly controllable via inputs $\mathsf{wait}_1, \ldots, \mathsf{wait}_{40}$. The test depicted in Figure 5.5c kills that mutant, as can be seen in the response of the mutant to the test in Figure 5.5d, which controls timing. However, note that the original mutant with uncontrollable non-determinism is only potentially killable, since after inputs open and close, there is a non-deterministic transition in the mutant that produces the prescribed output armed. That is, the test only kills the mutant reliably when timing can be controlled. Although in practice it might be



(a) A model with controllable non-determinism.



(b) A mutant with controllable non-determinism.



(c) A killing linear test.

(d) Spurious test response of mutant

Figure 5.5: A killing example for the non-deterministic case.

difficult to execute a test that requires to wait an exact amount of time, it should be noticeable whether the time to enter the armed state is twice as much as the expected time.

The car alarm system model can easily be modified to illustrate the mixed determinism case. To this end, a model with all timers being 0 can be compared to one with some timer greater than 0.

Test Suite Evaluation.

We evaluated the strength and correctness of the test suite created using the methods and toolchain presented in this chapter via the model-based mutation testing tool MoMuT [ABJ+15a, FKS+17] on a non-deterministic version of the car alarm system. To this end, we manually formulated the model both in SMV and compared it to its action system formulation, which is the native modeling language for MoMuT. MoMuT can evaluate a test suite by computing its mutation score — the ratio of killed to the total number of mutants — with respect to action system mutations (see Table 4.3) on a given action system model.

This procedure evaluates our test suite in two ways. Firstly, it shows that the tests are well formed, since MoMuT does not reject them. Secondly, it shows that the test suite is able to kill mutants other than those it was created from, which is important, because it suggests that the test suite is also able to detect faults in implementations.

We created a test suite consisting of 61 tests, using the toolchain presented in Section 5.4.1, automatically mapping it to the test format accepted by MoMuT and removing redundant tests, where a test is redundant if its string representation is a prefix of another test. For the test suite,

MoMuT measures a mutation score of 91% on 439 action system mutants. In comparison, the test suite achieves a mutation score of 61% on 3057 SMV mutants for which it was created. These results highlight that the mutation score is relative to the mutation operators used. On this model, the SMV mutation operators produce a lot more equivalent mutants than the action system mutation operators. Further characteristics of the resulting test suite are presented in the following Section 5.4.1.

Finally, we created a separate test suite using MoMuT with its default settings on the action system model. The resulting test suite consisted of 6 tests that kill 90% of the action system mutants, which were 8 mutants less than the test suite created via hyperproperties. The test suite created by MoMuT is more compact, because it was created directly for action system mutants instead of the larger number of SMV mutants. This result shows that hyperproperty model checking based test generation is well suited to kill a large array of mutants, while mature mutation testing tools are able to create more compact test suites that kill a large proportion of the mutants. A combined use of both techniques is one interesting future direction of this work.

5.4.3 Quantitative Experiments.

We present experiments on a series of benchmark that demonstrate the versatility and scalability properties of generating test suites via hyperproperty model checking. The experiments were run in parallel on a machine with an Intel(R) Xeon(R) CPU at 2.00GHz, 60 cores, and 252GB RAM. We used 16 Verilog models which are presented in [FRS15], as well as models from opencores.org. Furthermore, we used 76 SMV models that were also used in [AGR15]. Finally, we used the SMV formalism of CAS. All models are available in [ben]. Verilog and SMV experiments were run using property driven reachability based model checking with a time limit of 1 hour. Property driven reachability based model checking did not perform well for CAS, for which we therefore switched to bounded model checking with a depth limit of 100. All reported values are rounded to the first significant digit.

Characteristics of Models. Table 5.1 presents characteristics of the models. For Verilog and SMV, we present average (μ), standard deviation (σ), minimum (Min), and maximum (Max) measures per model over the set of models. For CAS we report the values for that single model. We report the size of the circuits in terms of the number of **Input**-, **Output**-, **State** variables as well as the number of *And* **Gates**, which corresponds to the size of the transition relation of the model. Furthermore, in row Δ **Gates (%)**, we report the average absolute size difference (in % of number of Gates) of the conditional mutant and the original model, where the average is over all mutants. Finally, **Mutants** shows the number of the mutants that are generated and analyzed for the models.

We can observe that our method is able to handle models of respectable size, reaching thousands of gates. Furthermore, Δ Gates of the conditional mutants is relatively low. Thus, conditional mutants allow us to compactly encode the original and mutated model in one model. Hyperproperties enable us to refer to and juxtapose traces from the original and mutated model, respectively. Classical temporal logic does not enable the comparison of different traces. Therefore, mutation analysis by model checking classical temporal logic necessitates to represent all combined

Parameters	Verilog				SMV				CAS
	μ	σ	Min	Max	μ	σ	Min	Max	
Models	16				76				1
Input	186	310	4	949	9	13	0	88	58
Output	177	299	7	912	4	4	1	28	7
State	16	16	2	40	-	-	-	-	-
Gates	4207	8309	98	25193	189	210	7	1015	1409
Δ Gates (%)	3	3	0.1	10	8	8	0.3	35	0.9
Mutants	260	236	43	774	535	1042	1	6304	3057

Table 5.1: The characteristics of event structure experiments benchmarks.

original- and mutated model traces explicitly, which can be achieved by building the product of the two models, but which results in a quadratic blowup in the size of the input to the classical model checker in comparison to the size of the input to the hyperproperty model checker.

Model Checking Results. Table 7.2 summarizes the quantitative results of our experiments. Similarly to above, for Verilog and SMV, we present average (μ), standard deviation (σ), minimum (Min), and maximum (Max) measures per model over the set of models as well as the respective values for the single CAS model. The quantitative metrics we use for evaluating our test generation approach are **Killed (%)** the percentage of killed mutants, **Equivalent (%)** the percentage of equivalent mutants, **Avg. test length**, **Max test length**, the average respectively maximal test case length for tests produced for each killed mutant, as well as **Avg. Runtime (s)** the amount of model checking time per killed respectively equivalent mutant. Furthermore, we report **Timeout (%)** the percentage of mutants exceeding the time limit or BMC depth bound. For Verilog and SMV the time limit was 1 hour. For CAS the depth limit was 100 transitions.

Finally, we report **Total time (h/s)**- the total time for test suite creation per model, including timeouts, in hours or seconds for very small models. The total time is the sum of the per mutant model checking times, i.e. assumes sequential test suite creation. However, since mutants are model checked independently, the process can easily be parallelized, which drastically reduces the total time needed to create a test suite for a model, typically from hours to a few minutes. The times of the Verilog benchmark suite are dominated by two instances of the secure hashing algorithm (SHA), which are inherently hard cases for model checking.

We can see that the test suite creation times are in the realm of a few hours, which collapses to minutes when model checking instances in parallel. However, the timing measures really say more about the underlying model checking methods than our proposed technique of mutation testing via hyperproperties. Furthermore, we want to stress that our method is agnostic to which variant of model checking (e.g. property driven reachability, or bounded model checking) is used. As discussed above, for CAS switching from one method to the other made a big difference.

The mutation score average is around 60% for all models. It is interesting to notice that the scores of the Verilog and SMV models are similar on average, although we use a different mutation

Metrics	Verilog				SMV				CAS
	μ	σ	Min	Max	μ	σ	Min	Max	
Killed (%)	57	33	5	99	65	31	0	100	62
Avg. test length	4	2	2	8	15	58	4	462	6
Max test length	22	50	3	207	187	1279	4	10006	9
Avg. Runtime (s)	83	268	0.01	1068	1	5	-	47	8
Equivalent (%)	33	32	0	95	35	31	0	100	0
Avg. Runtime (s)	45	120	-	352	1	2	-	15	-
Timeout (%)	10	27	0	86	0	0	0	0	38
Total time (h/s)	69	169	3s	620	0.4	1	0.1s	7	1

Table 5.2: The event structure based language inclusion experiments results.

scheme for the types of models. Again, the mutation score says more about the mutation scheme than our proposed technique. Notice that we can only claim to report the mutation score, because, besides CAS, we used a complete model checking method (property driven reachability). That is, in case, for example, 60% of the mutants were killed and no timeouts occurred, then 40% of the mutants are provably equivalent. One very large Verilog model of a SHA encoding caused 86% of model checking runs to time out. In contrast, incomplete methods for mutation analysis can only ever report lower bounds of the mutation score. Furthermore, as discussed above, the 61% of CAS translate to 91% mutation score on a different set of mutants. This indicates that the failure detection capability of the produced test suites is well, which ultimately can only be measured by deploying the test cases on real systems.

5.5 Related Work

5.5.1 Hyperproperties

Hyperproperties were originally introduced to formally express security properties, such as non-interference, in [CS10]. The paper works out the theoretical foundations of hyperproperties and contrasts them to classical trace properties. Traditionally, hyperproperties were formulated and used in a case by case fashion, see for example [McL92, vdMZ07, KIN15].

In order to generalize these approaches and to enable rigorous study of hyperproperties, logics for hyperproperties were developed [CFK+14, CFHH19]. In particular, HyperLTL and HyperCTL*, which are hyperproperty sensitive extensions of classic temporal logics, are used in this work. The expressive power of HyperLTL was characterized to be equivalent to first order logic over disjoint copies of the natural numbers and a restricted type of quantification [FZ17].

Furthermore, the satisfiability-[FH16, FHS17], monitoring-[BSB17, FHST19], and model checking [FRS15] problems for HyperLTL were tackled. The initial lack of model checking techniques for HyperLTL formulas with quantifier alternation was recently addressed in [CFST19] via a combination of reactive synthesis and model checking of quantifier alternation free HyperLTL formulas. Unfortunately, the respective version of the HyperLTL model checker MCHy-

PER is currently only available in a web-based version. In addition, [HSB20] presents a bounded model checking algorithm for HyperLTL formulas with quantifier alternation, which can be understood as a generalization of the method presented in Section 5.3.3 and which, due to its boundedness, unfortunately suffers from the same problem of generally not being able to identify equivalent mutants.

5.5.2 Model Checking-based Test Case Generation

A number of test case generation techniques are based on model checking; a survey is provided in [FWA09a]. The approach has been demonstrated to scale to the industrial setting in [ECO+16]. In [FWA09b] a thorough evaluation and comparison of different model checkers applied to the test generation problem over multiple modeling formalisms is presented.

Most model checking based test generation target, in comparison to our work, different coverage metrics and/or abstraction levels, such as structural coverage criteria for Java programs [VPK04] and RSML models [RH01] or information/data flow criteria for extended finite state machines [HLSU02].

However, mutation testing via model checking has been explored as well. Perhaps the first approach is presented in [ABM98] for SMV models that contain a syntactical description of a transition system, such as the one presented in Figure 5.1a, as well as temporal logic specifications that are true over this transition system. The specification is mutated and inconsistencies are detected via model checking. A similar approach is followed in [BOY00], where the temporal logic specification reflects the transition system. In contrast to this approach, our approach does not require two separate artifacts, but instead works directly on the transition system representation of the model. In fact, in our experiments on SMV models, we never touch the specification parts, but simply use the formalism as a way to represent non-deterministic transition systems.

[GH99] presents an approach to formulate mutation killing via trap properties. Trap properties are conditions that, if satisfied, indicate a killed mutant. [FW08] leverages model checking based mutation testing for creating property relevant test suites via trap properties. A similar idea is presented in [XeAXW12], where model mutants are model checked against a set of pre-defined trap properties. In contrast, our approach directly targets the input-output behavior of the model and does not require formulation of model specific trap properties.

[OBY03] proposes model checking based mutation testing via explicit model duplication and mutation of one duplicate. Furthermore, the equality of output variables is asserted and model checked. This approach is similar to ours. However, in contrast to explicit syntax-based model duplication, the only syntactic change to the model applied in our method is the addition of the mutant flag as well as an encoding of the immediate effect of the respective mutant operator. Which parts of the mutated state space differ from the model and therefore need to be constructed is decided automatically by the model checker. Thus, our approach is more versatile and can be applied without much extra encoding effort.

Mutation based test case generation via module checking is proposed in [BPG07]. The theoretical framework of this work is similar to ours, but builds on module checking instead of

hyperproperties. Moreover, no implementation or experimental evaluation is provided, leaving the practical applicability of the approach open.

In [ALN13] an approach for mutation based test case generation of timed automata is presented. The test case generation problem is reduced to a language inclusion problem, which is solved via bounded SMT model checking. Similarly, [ESC+16] presents an approach to mutation-based test generation via model checking for embedded software. The authors combine the original model, mutants, as well as mutation detection monitors into one timed automaton model. A reachability property is then model checked over this combined model to generate killing test cases. Likewise, [RBF11] present bounded model checking-based mutation testing for programs given in the LLVM intermediate representation [LA04], which is an assembler-like low level language. A killing condition that is similar to potential killing is formulated and is handed to the solver together with bounded loop unrolling encodings of program and mutant. In contrast to our work, in presence of non-determinism, all these encodings of mutation killing can not differentiate between potential and definite killing.

In [FG09] model checking based test generation is used to check requirements property-focused coverage criteria, such as mutation of properties. Similarly, [BOY00] presents an approach to create mutated requirements and use model checking of SMV models to identify models that fulfill the faulty requirements. Our work is orthogonal to such approaches, since we consider test generation over mutations of the system instead of the property.

5.5.3 Symbolic Test Case Generation

The authors of [AJT14] present an approach to mutation-based test generation for action system models via a symbolic refinement condition. The refinement condition as well as sets of reachable states are iteratively computed by solving SMT problems. While this work offers an interesting practical solution for action systems, our approach targets a larger class of systems that can be encoded as symbolic transition systems. In a similar fashion, the MuAlloy [WSK18] framework enables model-based mutation testing for Alloy models using SAT solving. In this work, the model as well as killing conditions, are encoded into a SAT formula and solved using the Alloy framework. Likewise, in [AGV15] configurations of feature models that distinguish the correct model from mutants of it are automatically created via SMT solving. The resulting encoding contains a logical representation of the model, the mutant, and a distinguishing condition. In contrast to these approaches, we encode only the killing conditions into a formula and leave encoding of the transition system to the model checker. Therefore, our approach is more flexible and more likely to be applicable in other domains. We demonstrate this by producing test cases for models encoded in two different modeling languages.

Symbolic methods for weak mutation coverage are proposed in [BKC14] and [BDD+15]. The former work describes the use of dynamic symbolic execution for weakly killing mutants. The latter work describes a sound and incomplete method for detecting equivalent weak mutants. The considered coverage criterion in both works is weak mutation, which, unlike the strong mutation coverage criterion considered in this work, can be encoded as a safety trace-property. However, both methods could be used in conjunction with our method. Dynamic symbolic execution could

be used to first weakly kill mutants and thereafter strongly kill them via hyperproperty model checking. Equivalent weak mutants can be detected with the methods of [BDD+15] to prune the candidate space of potentially strongly killable mutants for hyperproperty model checking.

CHAPTER 6

Test Case Generation via Heuristic-guided Branching Search

Clearly, formal, logic-based approaches to solving some problem have their limitation, as we have seen in the previous chapter. In our case, the practical limitation was the absence of solver infrastructure. In addition, the high computational complexity of rigorously deciding strong killability is a theoretical limitation. In order to overcome these limitations practical and scalable methods are necessary that can handle large, real world problem instances.

In this chapter, we present such a model-based mutation-driven test generation method that is based on an explicit state branching search algorithm. The search algorithm is inspired by the rapidly exploring random trees [LaV98] path planning algorithm which organizes the exploration of the search space in many small sub-searches that all have their individual goals. We call this type of search "branching search". The method is parametrized not only by the depth of the search and the individual sub-searches, but also by various heuristics that direct sub-searches. The rich parameter space allows us to customize coverage goals to the computational and syntactic demands of models as well as the available computational resources. In addition to efficiently exploring the model's state space, mutants are analyzed lazily and on the fly, in order to explore their state spaces only in regions where a divergence from the model can be expected.

We start the chapter by introducing the algorithmic framework, its parameters, and how to do mutation based test case generation within this framework. We then present a range of heuristics that can be plugged into the algorithmic framework to adjust for different model characteristics or simply to perform a diverse kind of search. Towards our experimental evaluation, we present a series of models that are industry use cases and that range from small to large models. In our experiments, we evaluate our parameter space, show that different heuristics work well on different models, and that branching out the search is indeed beneficial.

6.1 Branching Search Algorithm

In this section, we present the branching search-based test case generation algorithm. The presented method explores the state space of a model in a heuristic-guided branching search and performs mutant killing analysis lazily on the fly. In a final step, search results are transformed into a test suite. Throughout this section, we present the main search algorithm, discuss how to process mutants within branching search, show how to maintain search results and finally show how to use these search results to create test cases. To this end, let us fix an action system conditional mutant $\mathcal{A}^{\bar{\mu}} = \langle \mathcal{V} \cup \{\mathrm{mut}\}, s_\iota{}^{\bar{\mu}}, Act^{\bar{\mu}}, A_\iota \rangle$ for mutation setting $\{\mu_1, \ldots, \mu_n\}$.

The main source of inspiration for the exploration algorithm procedure was the rapidly exploring random trees (RRT) path planning algorithm [LaV98]. An interesting property of RRT for our problem is that it is able to reach different areas of large state spaces quickly [KL00]. We assume that in different areas of the state space different actions are enabled and as a consequence different mutations can be found. RRT splits the search into small, independent sub-searches, which we call branching search. The (almost) independence of sub-searches allows branching search to be implemented in a parallel fashion. In our algorithm description, we refer to sub-searches as tasks and denote them by τ. Each task starts in some previously discovered state and explores the state space for a fixed number of steps, where one step corresponds to computing $vispath_{\mathcal{A}^{\bar{\mu}}}(j, s)$ for some mutant j and state s. Over the course of the search, we repeatedly create new tasks. In which state a task starts and which transition it takes is determined by heuristics that are presented in Section 6.2.

6.1.1 State Space Exploration

Algorithm 1 shows the pseudo code of our state space exploration algorithm. The algorithm has the following parameters that need to be instantiated: BRANCHLENGTH is the number of steps performed by each single task. MAXSTEPS is the total number of states explored by all tasks combined. To explore a state s, we calculate the predicate $vispath_{\mathcal{A}^{\bar{\mu}}}(0, s)$, i.e. compute all outgoing transitions and successor states of state s for the original model. CREATETASK is the heuristic to create new tasks, which essentially chooses the starting state of tasks, and possibly creates a goal for the task. SELECTSUCCESSOR is the heuristic to select which successor state should be explored next by the task. A task is a tuple $\tau = \langle \tau.state, \tau.goal, \tau.steps, \tau.propagate \rangle$, where $\tau.state \in visStates_{\mathcal{A}}$ is its current state, $\tau.goal$ is its goal state, $\tau.steps$ is its integer step counter, and $\tau.propagate$ is a set of mutants that should be propagated while exploring τ.

The algorithm executes two main blocks of instructions in a loop until the maximum number of steps has been performed. The first block (lines 6–8) starts new tasks. We want to start a new task initially ($stepCount = 0$) and whenever the number of steps between the current and the last time a task was created exceeds the branch length for each task divided by 4. We delay the creation of new tasks as opposed to eagerly starting as many tasks as possible, because start states of tasks are chosen among the previously discovered states. Therefore, starting many tasks at once would result in all of them starting in the same state, which is not the branching behavior we typically want. We found BRANCHLENGTH divided by 4 to be a good compromise between starting enough tasks to leverage branching search and its feature to parallelize the search and

Algorithm 1: The state space exploration algorithm.

```
   Global: Exploration model G
1  stepCount ← 0
2  lastTaskCount ← 0
3  activeTasks ← ∅
4  reached ← ∅
5  while stepCount < MAXSTEPS do
6      if stepCount − lastTaskCount ≥ BRANCHLENGTH/4 ∨ stepCount = 0 then
7          lastTaskCount ← stepCount
8          activeTasks ← activeTasks ∪ CREATETASK
9      for τ ∈ activeTasks do in parallel
10         if τ.steps < BRANCHLENGTH then
11             reached ← reached ∪ {π.s | π ∈ vispath_{A^ū}(0, τ.state)};
12             EXTENDMODEL(τ.state, vispath_{A^ū}(0, τ.state));
13             π′ ← SELECTSUCCESSOR(vispath_{A^ū}(0, τ.s));
14             PROCESSMUTANTS(τ, π′);
15             τ.state ← π′.s;
16             τ.steps ++;
17             stepCount ++;
18         else
19             activeTasks ← activeTasks ∖ {τ}
20 GENERATETESTS
```

delayed starting to benefit from prior exploration. Note that as we start more tasks, the global step counter increases faster. Thus, the number of active tasks grows exponentially with the number of global steps performed. Different start delaying mechanisms are conceivable, such as a strategy based on Luby sequences, which are popular within SAT-based restarting strategies [HH14]. When creating a new task, we heuristically choose a start state among all states in the *reached* set as well as a goal state. Note that when a task reaches its goal state, the task does not stop, but explores successor states close to its goal state. An alternative approach would be to terminate the task and start a new one instead.

The second block (lines 9–19) performs the main work of the search by computing visible successor paths, extending an exploration model, choosing a successor path, and processing mutants. The latter three concepts are explained in detail in the following. Visible successor path computation is implicitly done in the algorithm by evaluating $vispath_{\mathcal{A}^{\bar{\mu}}}(0, s)$, which we assume to be computed the first time it occurs. In order to compute $vispath_{\mathcal{A}^{\bar{\mu}}}(0, s)$ the underlying do-od loop of the model is unrolled, guards are evaluated, and assignments are applied to the state. Across the whole test case generation method these steps are by far the computationally most expensive part. Therefore, it is beneficial to calculate successor paths in parallel. All visible successor states are added to the *reached* set of states for possible further exploration. Finally, the task's state and step counter as well as the global step counter are updated.

6.1.2 Exploration Model

We record search results in a labeled Mealy machine that we call *exploration model*, from which test cases are extracted in the final step of our test generation method. The exploration model is a Mealy machine $G = \langle S, \{s_0\}, \Sigma, \Lambda, \delta, lsk, lwk, Vert\rangle$ together with strong- respectively weak- kill annotations $lsk : \delta \to 2^{\{1,\ldots,n\}}$ respectively $lwk : \delta \to 2^{\{1,\ldots,n\}}$ and a function $Vert : visStates_{\mathcal{A}} \to S$ that translates visibly reachable action system states to states of the exploration model that we initialize with $Vert(s_\iota) \stackrel{\text{def}}{=} s_0$. The exploration model is a subset of the Mealy machine representation of the analyzed action system (as presented in Section 3.4) in the sense that if $\mathcal{M}^{\mathcal{A}} \stackrel{\text{def}}{=} \langle S^{\mathcal{A}}, S^{\mathcal{A}}_\iota, \Sigma^{\mathcal{A}}, \Lambda^{\mathcal{A}}, \delta^{\mathcal{A}}\rangle$ is the full Mealy machine representation of the analyzed action system model, then at every state of the exploration, we have $S \subseteq S^{\mathcal{A}}$ and $\delta \subseteq \delta^{\mathcal{A}}$ (up to isomorphism). Function EXTENDMODEL adds the visible successor paths Π of action system state s to the exploration model starting at $Vert(s)$, as given by the Mealy machine translation of action systems in Section 3.4, and updates $Vert(.)$ accordingly for newly visited successor states. Furthermore, functions ANNOTATESTRONGKILL and ANNOTATEWEAKKILL (used during mutant processing) annotate transitions corresponding to visible successor paths with the strongly and weakly killed mutant respectively.

6.1.3 Lazy Mutant Processing

Mutants are processed lazily via the functions PROCESSMUTANTS and STRONGKILLCHECK presented in Algorithm 2. Every task τ has a set $\tau.propagate$ of mutants and their current states. The attached mutants traverse the same paths through the action system as its propagating task and visible successor paths are compared to the model in each step.

Function PROCESSMUTANTS first gathers the mutants in successor paths of the model that were not previously weakly or strongly killed in Line 3. Thereafter, the set of mutants that should be propagated along the task is updated to all currently propagating mutants that are not strongly killed yet. Because we check that new mutants were not killed before, it is guaranteed that the sequence of states and paths in the model that leads to $\tau.state$ is also present in the mutant.

Thereafter, visible successor paths for attached mutants are computed implicitly by evaluating $vispath_{\mathcal{A}^{\bar{\mu}}}(j, s_{\mu_j})$ in Line 5. Similarly to computing the successor paths of the model, this is a computationally expensive step and it is beneficial to perform these computations in parallel. The successor paths of the model and mutant are analyzed for a potential strong kill, using function STRONGKILLCHECK that checks whether some visible successor path of the mutant is not present in the model. Mutants that are strongly killed are added to the global set of strongly killed mutants and are removed from the task's propagating set of mutants.

In case no strong kill was found, Line 10 checks whether the mutant is weakly killed by the current successor states. If the mutant is weakly killed, we add it to the set of weakly killed mutants and propagate it further along the paths chosen by the task, since the weak kill can induce a strong kill in further steps. To this end, we store the successor state of the mutant s'_{μ_j} corresponding to the chosen successor path labels $\pi'.l$. This state can but does not need to be different from the task's next state. However, since mutants follow paths explored by tasks, it is not guaranteed that the state is different.

Algorithm 2: The Lazy Mutant processing functions.

```
   Global: Exploration model G
   Global: Weakly killed mutants wk
   Global: Strongly killed mutants sk
1  Function ProcessMutants (Task τ, Successor path π')
2      newMutants ← (⋃_{π∈vispath_{A^μ}(0,τ.state)} ⋃_{ℓ∈π.l} m(ℓ)) ∖ (wk ∪ sk);
3      τ.propagate ← (τ.propagate ∪ {(μ_j, τ.state) | μ_j ∈ newMutants}) ∖ sk;
4      for (μ_j, s_{μ_j}) ∈ τ.propagate do in parallel
5          π_sk ← StrongKillCheck(vispath_{A^μ}(0, τ.state), vispath_{A^μ}(j, s_{μ_j}));
6          if π_sk ≠ ⟨⟩ then
7              e ← AnnotateStrongKill(τ.s, π_sk.l, j);
8              sk ← sk ∪ {(μ_j, e)};
9              τ.propagate ← τ.propagate ∖ {(μ_j, s_{μ_j})};
10         else if π'.s ∉ succ_{A^μ}(j, s_{μ_j}) then
11             AnnotateWeakKill(τ.state, π'.l, j);
12             wk ← wk ∪ {μ_j};
13             s'_{μ_j} ← π_{μ_j}.s where π_{μ_j} ∈ vispath_{A^μ}(j, s_{μ_j}) such that π_{μ_j}.l = π'.l;
14         else
15             τ.propagate ← τ.propagate ∖ {(μ_j, s_{μ_j})};
16 Function StrongKillCheck(Model paths Π, Mutant paths Π^μ)
17     if Π^μ ⊈ Π then
18         return π^μ ∈ Π^μ ∖ Π;
19     return ⟨⟩;
```

Finally, mutants that are neither weakly nor strongly killed in the current step are dropped from the task's propagating mutants in Line 15.

Mutant processing is structured such that the mutant's state space is not explored in common regions of model and mutant before its weak kill as well as after its strong kill. Therefore, mutation processing is lazy in the sense that the state space of mutants is only partially explored in regions where they potentially diverge from the model.

We analyze potential killing as opposed to definite killing in this test case generation algorithm, since definite killing requires to explore and compare multiple traces through the model and mutant. Such a trace comparison is orthogonal to the main focus of our search algorithm which is to explore large parts of the state space of large model. Definite killing is indeed better achieved via symbolic methods, such as those presented in Chapter 5.

Example 6.1.1. Consider again our running example and its mutants present in Example 3.4.1 and Example 4.1.3 respectively. Our exploration starts with a task $\tau = \langle [2], \tau.goal, 0, \varnothing \rangle$, where we leave $\tau.goal$ abstract for this example. We have $vispath_{\mathcal{A}^{\bar{\mu}}}(0, [2]) = \{(\langle \mathbf{ctr}\,\mathsf{request} \mid \mathbf{obs}\,\mathsf{coffee} \rangle, [1]), (\langle \mathbf{ctr}\,\mathsf{request} \mid \mathbf{obs}\,\mathsf{tea} \rangle, [1]), (\langle \mathbf{ctr}\,\mathsf{refill} \mid \mathbf{obs}\,\eta \rangle, [2])$. The first path does not have any mutants attached, whereas for the second path, we have $\mu_1 \in m(\mathsf{tea})$. Thus, mutant μ_1 is weakly and strongly kill checked. However, since it is an equivalent mutant

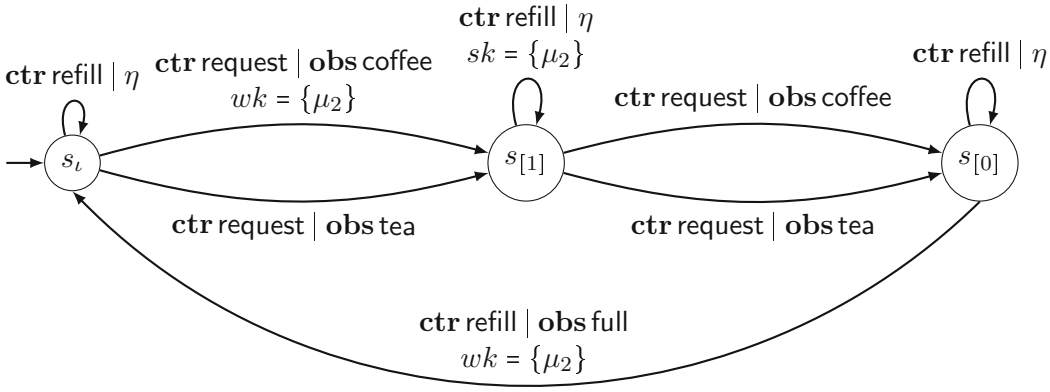


Figure 6.1: The exploration model of running example and mutants.

both checks fail and the mutant is not propagated further. After the next state $[1]$, the same mutant is found and analyzed. For the following state $[0]$, we have $vispath_{\mathcal{A}^{\bar{\mu}}}(0,[0]) = \{(\langle \mathbf{ctr}\,\mathsf{refill} \mid \mathbf{obs}\,\mathsf{full}\rangle,[2]),(\mathsf{request} \mid \mathbf{obs}\,\eta,[0])\}$ and $\mu_2 \in m(\mathsf{refill})$. For this mutant, we have $vispath_{\mathcal{A}^{\bar{\mu}}}(2,[0]) = \{(\langle \mathbf{ctr}\,\mathsf{refill} \mid \mathbf{obs}\,\mathsf{full}\rangle,[1])$. The path labels are the same as in the model, but the state is different. Therefore, we annotate the exploration model with a weak kill for mutant μ_2 and propagate it further. In the following exploration step, we reach state $[1]$ in the model and state $[0]$ in mutant μ_2, while the labels are still the same. Therefore, we again annotate the exploration model with a weak kill. Finally, in the step thereafter, we have $\langle \mathbf{ctr}\,\mathsf{refill}, \mathbf{obs}\,\mathsf{fill}\rangle \in vispath_{\mathcal{A}^{\bar{\mu}}}(2,[0])$ and $\langle \mathbf{ctr}\,\mathsf{refill}, \mathbf{obs}\,\mathsf{fill}\rangle \notin vispath_{\mathcal{A}^{\bar{\mu}}}(0,[1])$. Therefore, we found a strong kill and annotate the transition corresponding to input $\mathbf{ctr}$ refill with a strong kill for μ_2. The resulting exploration model is depicted in Figure 6.1.

6.1.4 Test Case Generation

We now show how extract locally adaptive strongly and potentially killing test cases from an exploration model $G = \langle S, S_\iota, \Sigma, \Lambda, \delta, lsk, lwk\rangle$. Linear tests can be constructed the same way by leaving out allowed outputs. Partially adaptive tests potentially require full knowledge of the model, i.e. transitions outside of the exploration model, which is outside of the scope of our test case generation algorithm.

In the exploration phase, the set sk is constructed, which contains tuples of mutants and transitions that indicate where strong kills end in the exploration model. In the test case generation phase, depicted in Algorithm 3 and triggered by calling CREATETESTS(sk), sequences of input, output and sets of allowed outputs tuples are constructed and attached together to form a test. A test case that kills some mutant μ corresponds to a path in the exploration model that starts in the initial vertex, followed by a (potentially empty) sequence of transitions without a killing annotation for μ, constructed via function PREFIX, a (potentially empty) sequence of transitions with μ- weak killing annotations, constructed via function WEAKKILLSEQUENCE, and ending in a transition with a μ- strong killing annotation, constructed via function STRONGKILLSEQUENCE.

Algorithm 3: The test case generation algorithm.

Global: Exploration model G

Function CREATETESTS *(Strongly killed mutants sk)*
  $tests \leftarrow \emptyset$;
  for $(\mu, e) \in sk$ **do**
    $(\langle (i,o,O)_{n_s}, \ldots, (i,o,O)_n \rangle, e_s) \leftarrow$ STRONGKILLSEQUENCE(e, μ);
    $(\langle (i,o,O)_{n_w}, \ldots, (i,o,O)_{n_s} \rangle, e_w) \leftarrow$ WEAKKILLSEQUENCE(e_s, μ);
    $\langle (i,o,O)_0, \ldots, (i,o,O)_{n_w} \rangle \leftarrow$ PREFIX(e_w);
    $tests \leftarrow tests \cup \{\langle (i,o,O)_0, \ldots, (i,o,O)_{n_w}, \ldots, (i,o,O)_{n_s}, \ldots, (i,o,O)_n \rangle\}$;
  REMOVEREDUNDANTTESTS$(tests)$;
  return $tests$;
Function STRONGKILLSEQUENCE *(Transition $e = (v,i,o,v')$, Mutant μ)*
  $O \leftarrow \{o_{post} \mid \exists v'_{post} : (v, i, o_{post}, v'_{post}) \in \delta\}$;
  if $\exists\, e_{pre} = (v_{pre}, i_{pre}, o_{pre}, v)$ *with* $(\mu \in lsk(e_{pre}))$ **then**
    $(seq, e_{start}) \leftarrow$ STRONGKILLSEQUENCE(e_{pre}, μ);
    return $((seq \circ (i,o,O)), e_{start})$;
  else
    return $(\langle (i,o,O) \rangle, e)$;
Function WEAKKILLSEQUENCE *(Transition $e = (v,i,o,v')$, Mutant μ)*
  $O \leftarrow \{o_{post} \mid \exists v'_{post} : (v, i, o_{post}, v'_{post}) \in \delta\}$;
  if $\exists e_{pre} = (v_{pre}, i_{pre}, o_{pre}, v)$ *with* $(\mu \in lwk(e_{pre}))$ **then**
    $(seq, e_{start}) \leftarrow$ WEAKKILLSEQUENCE(e_{pre}, μ);
    return $((seq \circ (i,o,O)), e_{start})$;
  else
    return $(\langle (i,o,O) \rangle, e)$;
Function PREFIX *(Transition $e = (v,i,o,v')$)*
  $O \leftarrow \{o_{post} \mid \exists v'_{post} : (v, i, o_{post}, v'_{post}) \in \delta\}$;
  if $v = v_\iota$ **then**
    return $\langle (i,o,O) \rangle$;
  else
    $e = \arg\min_{(v_{pre}, i_{pre}, o_{pre}, v) \in \delta} (depth(v_{pre}))$;
    return PREFIX$(e) \circ (i,o,O)$;

The respective functions transform transitions of the exploration model into tuples of inputs, outputs, and allowed outputs. Function CREATETESTS constructs and assembles these sequences in reverse order. Functions STRONGKILLSEQUENCE and WEAKKILLSEQUENCE return, in addition to the respective sequence, its first transition in the exploration model, which is the beginning of the construction of the next part of the sequence. For prefix construction, we assume the existence of a function $depth : S \to \mathbb{N}$ that gives the depth of vertices, i.e. the length of the shortest paths to v_ι. Finally, the function REMOVEREDUNDANTTESTS removes tests that are the prefix of another test.

6.2 Branching Search Heuristics

In this section, we present heuristics we developed for the branching search-based test case generation algorithm presented in the previous section. As noted above, the algorithm is inspired by the rapidly exploring random trees (RRT) path planning algorithm, which heavily relies on distance metrics.

Therefore, we start our presentation of heuristics with a discussion of distance metrics for action systems. We then present heuristics for two parameters of our search procedure: CREATETASK and SELECTSUCCESSOR. While some of the presented heuristics are distance-based and thus resemble the classic RRT framework, we also developed non-distance based heuristics that work well on some models. Finally, in this section we show how the heuristics can be adjusted in order to perform an exhaustive and distributed breath first search over the model.

6.2.1 Distance Metric

Many of our heuristics use a distance metric between states, abstractly denoted by $d(s_1, s_2)$, to guide the search. The notion of distance is also a key concept of rapidly exploring random trees. RRTs are usually applied to path planning in the plane, where the notion of distance is naturally given by the Euclidean distance.

Since action systems have many non-linear data types, such as objects or enumerate values, Euclidean distance is not an appropriate choice. Instead, we opted to use the Hamming Distance, a distance metric that defines distance 0 for equal values and distance 1 for unequal values. We calculate the distance between two lists as the average number of elements differing. We calculate the distance between two states as the average Hamming Distance between all its variables. Note that two states always have the same variables as their number is fixed by the action system. The intuition is that most actions manipulate only a small subset of the variables and executing the action that sets most of the variables to the desired value is beneficial to reaching our goal.

6.2.2 CREATETASK Heuristics

CREATETASK heuristics create fresh tasks by picking a $start$ state from the previously reached states $reached$, and a $goal$ state from the set of all possible states. The fresh task is the tuple $\langle start, goal, 0, \varnothing \rangle$.

Table 6.1 provides an overview of the CREATETASK heuristics described in more detail below. We assume a function $\mathsf{rand}(.)$ that returns a uniformly random element of its input set. $\mathcal{S}$ denotes the set of all possible states. That is, if we have a system with two integers with values between 0 and 10, $\mathcal{S}$ is the set $[0, 10] \times [0, 10]$. It is not guaranteed that all of these states are reachable. To generate random values for list variables, we first create a random list length and then create random values recursively.

Init The `Init` heuristic always chooses the initial state as the start state. Therefore, it is more likely to make different choices on early branches in the action system. Its downside is that is

Name	Description
`Init`	$goal \leftarrow \mathsf{rand}(\mathcal{S})$ $start \leftarrow s_\iota$
`RandCT`	$goal \leftarrow \mathsf{rand}(\mathcal{S})$ $start \leftarrow \mathsf{rand}(reached)$
`RGoal`	$goal \leftarrow \mathsf{rand}(\mathcal{S})$ $start \leftarrow \arg\min_{s \in reached}(d(s, goal))$
`CGoal`	$goal \leftarrow \mathsf{combine}(reached)$ $start \leftarrow \arg\min_{s \in reached}(d(s, goal))$
`PGoal` ($\mathcal{U}/\mathcal{R}$)	$start \leftarrow \mathsf{rand}(\mathcal{U}/\mathcal{R})$ $goal \leftarrow$ perturb $start$
`RoRoCT`	Round robin of other heuristics

Table 6.1: The CREATETASK heuristics.

Name	Description
`RandSS`	$s' = \mathsf{rand}(states)$
`Dist`	$\Lambda = \{(t, \lambda) \mid t \in states, \lambda = d(t, \tau.goal)\}$ $s' = \mathsf{select}(\Lambda)$
`Part`	$\Lambda = \{(t, \lambda) \mid t \in states, \lambda = \alpha(t)\}$ $s' = \mathsf{select}(\Lambda)$
`LocalBFS`	if max depth not reached add all unexplored new states to queue s' = queue.pop
`RoRoSS`	Round robin of other heuristics

Table 6.2: The SELECTSUCCESSOR($states$) heuristics.

might not penetrate the model as deeply as other heuristics and might repeat work previously performed when there are no unexplored options close to the initial state. The goal state is chosen randomly among all possible states.

RandCT The `RandCT` heuristic simply chooses one random state in $reached$. The goal state is chosen randomly among all possible states.

RGoal The `RGoal` heuristic resembles classic RRT search. A goal state is chosen randomly among all possible states. The start state is then chosen as the closest state in $reached$ according to distance d.

CGoal The `CGoal` heuristic works similar to `RGoal`. However, a goal state is not chosen at random, but is created as a random combination of previously reached states, abbreviated via the function $\mathsf{combine}(.)$. For every variable in the model, a value is picked from a randomly selected state of $reached$. The idea of this heuristic is to create goal states that are more likely to be reachable than purely random states.

PGoal The `PGoal` heuristic exists in two flavors called `PGoal`($\mathcal{U}$) and `PGoal`($\mathcal{R}$). Both have in common that they turn around the selection process used by RRT by first picking a start state and afterwards producing the goal from that state. The state to start from is randomly chosen from the set of *unique states* $\mathcal{U}$ or *rare value states* $\mathcal{R}$. These sets are defined as follows: $\mathcal{U} = \{s \in reached \mid s$ was inserted into $reached$ at most once$\}$, $\mathcal{R} = \{s \in reached \mid \exists v \in \mathcal{V}$ such that $|\{t \in reached \mid$ variable v has the same value in s and $t\}| < |reached|/10\}$. After picking a start state, a goal state is created by perturbing a fraction of the variable values of the chosen start state. By perturbing, we mean modifying a value, either by selecting a different value at random, or choosing integer values close to original integer values. The idea of these heuristics is to explore regions of the state space that have yet been sparsely explored.

RoRoCT The `RoRoCT` heuristic is a meta-heuristic that creates tasks according to multiple CREATETASK heuristics in a round robin fashion. For the experiments presented in this paper, `RoRoCT` chooses among all other heuristics. The idea of round robin is to combine the strengths of multiple heuristics. If one task gets stuck due to a bad heuristic decision, the next one is not likely to get stuck in the same way.

6.2.3 SELECTSUCCESSOR Heuristics

SELECTSUCCESSOR heuristics choose one successor state out of a given set successor states of *states* to explore next. Table 6.2 shows the different heuristics and gives a short overview.

We assume the existence of a helper selection function select(.) which picks one element out of a set of successor states according to provided state evaluations. As instantiations of this function, we experimented with `greedy`, simply taking the best evaluation; `weighted`, picking probabilistically according to the distribution given by the evaluations; and `bucket`, which first groups paths into buckets of equal evaluation, then picks a bucket probabilistically according to the distribution given by the evaluations, and finally picks a state randomly within the bucket.

RandSS The `Rand` heuristic randomly picks a state in *states*.

Dist The `Dist` heuristic assigns to each successor state in *states* its distance to the task's goal state and picks a successor state via one of the above select(.) functions. This heuristic emulates classic RRT search.

Part The `Part` heuristic assigns to each successor state s in *states* the *new object states* evaluation $\alpha(s)$ and picks a successor path via one of the above select(.) functions. The function $\alpha(.)$ returns the number of object values of a state, but not in the global set of previously reached states *reached*. An object value is the projection of the state to the values of one particular object instance. When checking whether we have encountered an object value before, we check whether that particular object value has been encountered in any instance of the object's class before.

The idea of this heuristic is to have a fine grained characterization of new information in states. For large models, the search algorithm constantly finds new states, since the number of combinations of variable valuations is high. Objects of the same type typically have a symmetric role, in the sense that it does not matter which particular instance is part of a state that might trigger new behavior. The `Part` heuristic exactly tries to capture and quantify the novelty of information a state supplies.

LocalBFS The `LocalBFS` heuristic works a bit different than the other heuristics. The idea of this heuristic is to combine heuristic-driven exploration with complete local search via depth bounded breadth-first search. Heuristic-driven exploration is supposed to quickly expand the search space into diverse areas, while bounded breadth-first search is supposed to uncover rare, bottleneck transitions that open up new regions of the state space. Since many of our target

models are too large for full breadth-first search, we limit the depth bound of local BFS searches to a fixed bound. Furthermore, in order to balance between heuristic-driven exploration and complete local search, we perform local BFS searches with a given frequency, e.g. every fourth task performs bounded BFS, while the others perform distance-based exploration.

RoRoSS The `RoRoSS` heuristic is a meta-heuristic that selects successors according to multiple SELECTSUCCESSOR heuristics in a round robin fashion. The heuristic is fixed for one task for its whole lifetime. Similarly to `RoRoCT` for task creation, we want to benefit from the advantages of all other heuristics and avoid to get stuck during the search.

6.2.4 Full Breadth-first Search

Using our search framework, we can also instantiate the heuristics to perform full breadth-first search, exploring all reachable states in the process. This strategy is only viable for small models. However, on models that are small enough to be explored fully, it produces interesting insights. For example, we are able to determine the state space and the number of reachable mutants. These measurements serve as a baseline for comparison of heuristics.

In order to perform full breadth-first search in our framework, in addition to the data kept in Algorithm 1, we maintain a queue of states. Tasks add successor states to this queue. The condition to create new tasks is replaced by a check whether there are still unexplored states in the queue. The CREATETASK heuristic simply picks the first unexplored state of the queue. Finally, branch length is set to 1, therefore the SELECTSUCCESSOR does not have an effect. The difference to `LocalBFS` is that full breadth-first search operates not on the task heuristic level, but on a global search level. That is, multiple tasks correspond to one full breadth-first search, whereas only one single task corresponds to one `LocalBFS` search. Furthermore, with full BFS we do not mix the exploration with other heuristics. Instantiating our algorithmic framework this way allows us to use the parallelization framework for breadth-first search.

6.3 Models

In this section, we present a series of models that we use for our experimental evaluation in the following section. The models were collected from several industrial use cases. We start by presenting characteristics of the models and describe their use-cases thereafter.

Table 6.3 provides a comparison of some of the key properties of the models. The use cases for AlarmSystem, PartCountUML and Loader are described in [ABJ+15a, AAJ+14, ABJ+15b]. The models were automatically translated to action systems by MoMuT from three different modeling languages: UML, Event-B and a textual domain specific language (DSL). This source modeling language is given in the **Source** column. In this work we consider only the resulting action systems. Therefore all properties of the models described here relate to the action system representation. The **Obj** column shows the number of objects in the action system. Each object has its own independent do-od loop and can act concurrently to all other objects. Although not every object is always allowed to make a transition in every iteration, this property gives an

Model	Source	Obj	Ins	ℓ- Act	Vars	States	S Size	LoC
ChassisDyno	DSL	1	9	5	6	14	0.02	124
PartCountDSL	DSL	1	19	7	11	222	0.03	391
DemoTopology	Event-B	1	120	9	25	TO	4.9	451
AlarmSystem	UML	3	4	53	42	76	0.3	967
Debounce	UML	3	30	34	29	560	0.1	655
PartCountUML	UML	3	18	103	55	12138	0.5	1834
Defibrillator	UML	4	17	140	67	TO	0.4	2949
Loader	UML	4	1692	105	98	MO	0.8	2009
MMS	UML	125	86	1023	1490	MO	8.2	8281
LBT	UML	2373	826	1763	27959	MO	182.4	33289

Table 6.3: The properties of the test models.

indication of the amount of concurrent actors in the respective model. The **Ins** column lists the number of inputs the action system can accept. For controllable actions with parameters, the Cartesian product of the parameter ranges has been used, over-estimating in some cases. The **ℓ-Act** column shows the number of labeled actions in the action system. The **Vars** column shows the total number of variables of the action system. The **States** column shows the result of full BFS in terms of number of states found, if successful, timeout (TO) after 24 hours, or memory out (MO) when having 378 GB RAM available. The **S Size** column shows the size of a single state given in kilobytes, i.e. the maximally required memory to represent one state of the action system, assuming all lists are of maximal length (except for LBT where we chose to report the minimal length lists as the maximum would have been too much of an overestimation). Finally, the **LoC** column shows the number of lines of code of the action system model.

We provide a brief introduction of the models and references with more information about them when available.

AlarmSystem AlarmSystem is a simple model of a car alarm system. Previous results with earlier generations of MoMuT and the model have been described in, [ABJ+15a, ABJ+15b], where coverage numbers are not directly comparable to this work, because the set of mutations has been extended since then.

Debounce – Signal Debouncing Algorithm Debounce models a debouncing algorithm used in the domain of safety critical industrial control. It is counting time ticks depending on changes of an input value in order to decide if the changes shall be considered transient, either caused by bounces of a switch contact or by electro magnetic interference.

PartCountUML – Particle Counter (UML) PartCountUML is a model of a remote control protocol of an exhaust measurement device. In terms of complexities posed, the model is slightly more complex than AlarmSystem. Our initial findings of test case generation for PartCountUML

in an industrial context have been published previously [AAJ+14]. Here, we use the same model with our new test case generation engine.

PartCountDSL – Particle Counter (DSL) PartCountDSL is derived from a domain specific language (DSL) model reproducing the functionality of PartCountUML. The model translated from the DSL is leaner than the one coming from UML, the resulting action system models are differently structured. Due to this, despite expressing the same functionality, it contributes to the diversity of the models used in the experiments.

ChassisDyno – Chassis Dyno Controller (DSL) ChassisDyno is a very simple second measurement device, written in the same DSL as the model above.

Defibrillator – Automated External Defibrillator Defibrillator models the diagnostic logic of an automated external defibrillator device.

Loader – Loader Bucket Implement Loader models the control loop (including user feedback and error handling) of a bucket loader implement controller. The controller receives joystick deflection values as inputs and computes output values that will drive valves controlling the movements of the bucket. Although Loader is a rather small model, it is highly complex, as can be seen in Table 6.3: due to heavily parametrized actions, it requires the highest number of traces for one iteration of the do-od block. Initial findings with previous versions of MoMuT and the use case can be found in [ABJ+15b], which reports findings on a partial model of the system.

MMS, LBT – Railway Interlocking Systems MMS and LBT are instantiations of a railway interlocking system. The original UML models consist of two parts each: one shared general model that defines all classes and data structures, and one that instantiates the objects needed for the station. While MMS represents a minimal station that allows trains to pass one another, LBT is a model of a mid sized, real-life railway station. Its layout comprises 37 track sections, 56 track relays, 34 switches, 22 main signals, and 145 train routes the operator can select from. MMS, in contrast, only comprises 10 track sections, 4 track relays, 2 switches, 6 main signals, and 10 train routes. Both models are highly non-deterministic due to 2373 (LBT), and 125 (MMS) concurrently running objects. The objects are used to model both physical and logical entities, such as train routes. Both models make extensive use of lists and forall/exists quantifiers. For example, LBT includes more than 9000 lists in the state, has more than 50 exists quantifiers, and over 100 forall quantifiers that have a maximum nesting depth of five.

DemoTopology – Interlocking Logic DemoTopology models generic rules for safe operation of a railway interlocking system. As such, its functionality is a subset of MMS and LBT . The used station layout is slightly more complex than that of MMS . In contrast to the UML based interlocking system models, this model uses only a single instance and no concurrent objects. The original model is expressed in Event-B [Abr10a]. Event-B models make heavy use of sets and maps, which are not supported natively in action systems and have to be emulated using

lists. This emulation identifies two lists whose elements are permutations of the other as being unequal, i.e. adding two elements in different order would yield the same state in the Event-B semantics, but in our semantics the resulting states are different. That causes our full BFS exploration to time out. DemoTopology is a variant of the model described in [RFT16].

The UML models have been built by researchers in close cooperation with industry partners for their real use cases. The DSL and Event-B models have been built by industry partners. As can also be seen in Table 6.3, the models vary substantially in their characteristics, not only in size itself. The selected models provide different challenges for our test case generator. In terms of complexity, LBT is the biggest example because of its high number of concurrent objects, with MMS following at some distance. Debounce is the simplest, but still uses an integer to model discrete time. The latter is used in all models except the railway interlocking ones. Loader's challenging complexity stems from both its use of many concurrent timers and the large allowed value ranges for its input parameters, increasing the state space.

6.4 Branching Search Experiments

In this section, we present an experimental evaluation of the branching search-based test case generation algorithm presented in Section 6.1, its heuristics, and its parameters presented in Section 6.2 on the models presented in Section 6.3. We start by discussing some relevant implementation aspects and present our experimental setup. Finally, we present and discuss the obtained results, evaluating various aspects of our algorithm.

6.4.1 Implementation

The algorithms and heuristics presented in this work are implemented in the MoMuT tool [ABJ+15a]. In order to cope with the larger models, the tool is tailored towards scalability. To this end, the tool is written in C++ and the original and mutated action systems are just-in-time-compiled [LA04, Ayc03] to machine code, which allows us to execute transitions fast and leverage compiler optimizations.

We implemented the search procedure in an asynchronous parallelized way. There is a central scheduler, which accumulates and distributes data and performs the CREATETASK as well as SELECTSUCCESSOR computations. Additionally, there is a set of workers that perform the labor intensive $vispath_{\mathcal{A}^{\bar{u}}}(.,.)$ computations as well as kill-checking. Workers store gathered data in buffers, which are repeatedly harvested by the central scheduler. Workers request new tasks from the scheduler actively. Therefore, our parallel computation scheme has two synchronization points between scheduler and workers: the harvesting of buffers and obtaining new tasks. In between these synchronization points, all threads run asynchronously.

The implementation currently does not support propagation of weakly killed mutants. That is, we implemented a variant of PROCESSMUTANTS in Algorithm 2 that also removes weakly killed mutants from $\tau.propagate$ in the same way that strongly killed- and non killed- mutants are removed from this set. Interestingly, as can be seen below, a considerable amount of mutants can be strongly killed without propagation.

6.4.2 Experimental Setup

We used three different machines for our experiments. Machine 1 has an Intel(R) Xeon(R) CPU at 3.47GHz, 24 cores, and 189GB RAM. Machine 2 has an Intel(R) Xeon(R) CPU at 2.80GHz, 40 cores, and 378GB RAM. Machine 3 has an Intel(R) Xeon(R) CPU at 2.00GHz, 60 cores, and 252GB RAM.

We have four parameters in Algorithm 1 that can be instantiated: CREATETASK, SELECTSUCCESSOR, MAXSTEPS, and BRANCHLENGTH. There is a hidden fifth parameter: the random seed. Since a lot of our CREATETASK and SELECTSUCCESSOR heuristics involve randomness a pseudo-random number generator is used to ensure reproducibility of the results. Furthermore, we can control the maximum amount of cores used for parallel processing. We use all available cores if not otherwise specified.

6.4.3 Results Summary

We start the presentation of our results by showing measurements averaged over all runs per model, shown in Table 6.4. This should provide an overview of the results achievable with our techniques and display model characteristics. In total, we performed 6483 runs across all models in 564 hours of wallclock time. A run means that we started test case generation on a model with a specific set of parameters.

We report the number mutations introduced per model (**Mutants**). Further we present the number of mutants reached (**Reach**), the number of mutants strongly (**SKill**) killed, the number of mutants weakly, but not strongly killed (**WKill**), the number of tests (**# Tests**), and average test length (**TestL**), as well as the wallclock runtimes (**Time**) in seconds, averaged over the all runs performed per model. Finally, the last column (**Runs**) shows the overall number of runs we performed for the model.

Model	Mutants	Reach	SKill	WKill	# Tests	TestL	Time	Runs
ChassisDyno	184	178.9	117.8	49.1	9.6	1.2	0.6	722
PartCountDSL	589	451.4	320.4	118.0	47.7	2.9	1.9	1089
DemoTopology	171	154.4	67.7	40.9	7.4	4.5	1303.3	158
AlarmSystem	818	646.6	210.0	215.8	16.8	3.8	5.1	1122
Debounce	1015	922.5	122.9	185.6	18.8	4.8	30.5	1262
PartCountUML	2257	2144.0	711.6	437.3	74.9	225.8	201.9	1262
Defibrillator	3186	1907.9	613.7	959.8	35.0	899.2	258.2	440
Loader	3291	1995.5	1143.0	300.3	34.7	64.2	4801.8	154
MMS	6391	2686.2	885.5	994.8	39.1	88.7	1032.1	258
LBT	1884	214.1	48.5	65.6	10.9	41.9	25189.2	16

Table 6.4: A summary of experimental results.

6.4.4 Breadth-First Search

We started full breadth-first search on all models. Small models work fine and provide useful bounds on the number of mutants that can be reached, as well as how many steps are maximally necessary to do so. For larger models, full breadth-first search did not terminate. Either we reached more and more new states and aborted search after 24 hours (TO) or we ran out of memory (MO). These experiments were performed on Machine 2.

We present the relevant measurements for breadth-first searches in Table 6.5. We report times for runs that finished and TO or MO for unfinished runs. The measurements of unfinished runs represent the results before termination. The column **States** shows the number of states reached and the column **Depth** shows the maximal shortest path from the initial state to some state in terms of number of exploration steps. Columns **Reach** and **Time** report the number of mutants reached and runtime in seconds, as in the previous Table 6.4. For unfinished runs we report the results of the maximal depth fully explored and denote the results with ≥ symbols.

Model	**States**	**Depth**	**Reach**	**Time**
ChassisDyno	14	3	184	0.6
PartCountDSL	222	10	588	1.7
DemoTopology	≥ 13 129	≥ 3	≥ 160	TO
AlarmSystem	76	12	810	78
Debounce	560	9	978	215
PartCountUML	12 138	12	2230	1 719
Defibrillator	≥ 18 580 118	≥ 299	≥ 1 428	TO
Loader	≥ 1702	≥ 2	≥ 676	MO
MMS	≥ 266 112	≥ 4	≥ 1 275	MO
LBT	≥ 118 413	≥ 2	≥ 365	MO

Table 6.5: The results of the full breadth-first search experiments.

We use the results of full breadth-first search to classify our models into small, large, and LBT. Small models are those where BFS finished, i.e. models ChassisDyno, PartCountDSL, AlarmSystem, Debounce, and PartCountUML. Large models are all the others, except LBT. For such models, runs with realistic values for MAXSTEPS terminate within a few hours. LBT is special in its computational demands. We needed to set MAXSTEPS value to 500 and limit the mutants to 500 in order to get runs that terminate within reasonable time. We did not extensively evaluate heuristics on LBT, but demonstrate that our algorithm is able to process this huge model within reasonable time (7 hours per run on Machine 3). Note that we used partial order reduction, which is described in Chapter 7, to cope with the high concurrency of LBT.

6.4.5 Heuristic Evaluation on Small Models

For small models, we tested the whole cross product of all choices for CREATETASK and SELECTSUCCESSOR, 4 different random seeds, three values for MAXSTEPS, as well as nine values for BRANCHLENGTH. We chose the `bucket` *select*(.) function for all experiments, since it

outperformed the other two options in preliminary experiments. For the MAXSTEPS parameter we chose the number of states $|S|$, $|S|/2$, and $|S|/10$, where $|S|$ was determined with full BFS. For the BRANCHLENGTH parameter for each MAXSTEPS value, we tested MAXSTEPS, MAXSTEPS/5, and MAXSTEPS/20, resulting in a total of nine parameter combinations. These experiments were performed on Machine 2.

Model	MAXSTEPS = $\mathbf{|S|}$	MAXSTEPS = $\mathbf{|S|/2}$	MAXSTEPS = $\mathbf{|S|/10}$
ChassisDyno	99.4%	97%	-
PartCountDSL	86.6%	81.6%	61.9%
AlarmSystem	93.0%	86.5%	47.1%
Debounce	100.0%	100%	82.4%
PartCountUML	98.9%	97.3%	92.3%

Table 6.6: The mutants reached on small models in relation to MAXSTEPS and relative to full breadth first search.

In Table 6.6 we show the mean results in terms of number of mutants reached achieved by the heuristics with the different parameter values for MAXSTEPS in relation to the results of full BFS. We can see that the heuristics often manage to come close to the BFS results, even for smaller MAXSTEPS values. The last column for ChassisDyno is empty, because the model is too small for sensible experiments with MAXSTEPS = $\mathbf{|S|/10}$. Similarly, the last column in AlarmSystem corresponds to runs with just 7 steps, explaining the relatively poor performance. If we take the maximum, instead of average results, all table entries but three are 100% for MAXSTEPS = $\mathbf{|S|/10}$, where AlarmSystem, PartCountDSL, and PartCountUML have values 79.4%, 74.7%, and 99.8% respectively. This shows that the heuristics can achieve good results with low effort.

CREATETASK				**SELECTSUCCESSOR**			
	Mean	**# Best**	**# Bad**		**Mean**	**# Best**	**# Bad**
`CGoal`	96.9 %	1	0	`Dist`	93.1 %	1	1
`RGoal`	94.9 %	2	1	`RoRoSS`	92.7 %	2	1
`P(Rare)`	94.4 %	2	1	`RandSS`	92.6 %	2	1
`RandCT`	94.3 %	1	0	`Part`	92.5 %	3	2
`P(Uniq)`	94.2 %	2	1	`LocalBFS`	92.4 %	2	3
`RoRoCT`	90.6 %	1	3				
`Init`	86.3 %	2	3				

Table 6.7: The performance of heuristics on small models with MAXSTEPS = $\mathbf{|S|/2}$.

In Table 6.7 we report the performance of individual heuristics for MAXSTEPS = $\mathbf{|S|/2}$. We chose MAXSTEPS = $\mathbf{|S|/2}$, because we want to evaluate how well heuristics perform on a limited budget. The relative results for MAXSTEPS = $\mathbf{|S|/10}$ are similar, but the variability increases. Column **Mean** shows the average percentage of mutants reached in comparison to the number of mutants reached by BFS. The other two columns are the result of a statistical analysis, iteratively filtering the data set. We evaluate performance using a one-sided Mann-Whitney U test [MW47]

with a p-value $p = 0.05$. The test compares two sets of data and estimates how likely it is that the two sets are equally distributed, in our case with respect to the number of mutants reached. Starting from the full set of runs, we iteratively filter out runs corresponding to significantly under-performing heuristics. In every iteration and for every heuristic that remains in the data set, we split it into runs using that heuristic and all other runs. We record which heuristic was least likely to be as good as its complement-set and proceed with the respective complement-set. Columns **# Best** show for how many models the respective heuristic was among the remaining heuristics when filtering out until all heuristics are equally likely to perform as well as the others. Columns **# Bad** show for how many models the respective heuristic was thrown out during filtering by the statistical test.

We can see that the impact of CREATETASK heuristics is larger than the impact of SELECTSUCCESSOR heuristics, as the mean results of the best and worst heuristics are further apart. `CGoal` seems to be the best choice in the CREATETASK category. `RGoal`, `P(Uniq)`, `P(Rare)`, and `RandCT` perform roughly equally well. On the other hand `Init` and `RoRoCT` are significantly worse than the other heuristics on the small models. All heuristics are among the best ones in at least one model, showing that all heuristics can be useful on different models. The fact that `CGoal` performs better than `RGoal` shows that careful goal selection is important for good search results.

For the SELECTSUCCESSOR category, all the heuristics perform more or less equally well overall. However, `Part` and `LocalBFS` are significantly worse than other heuristics on more models. The fact that `Dist` cannot beat random based strategies more decisively, as would be predicted by the results of RRT in the path planning domain, indicates that a more accurate distance metric is necessary to better reflect mutation reaching capability.

6.4.6 Heuristic Evaluation on Large Models

As mentioned above, testing all combinations of heuristics is infeasible for large models. Therefore, we made a selection of heuristic combinations, based on the data of the small models. To this end, we ranked each combination of CREATETASK and SELECTSUCCESSOR with their mean result and selected combinations that were performing particularly badly (`Init` × `Part`, `Init` × `RandSS`, and `RGoal` × `LocalBFS`), averagely (`RandCT` × `RandSS`, `RandCT` × `Part`, and `RoRoCT` × `Dist`), and well (`RGoal` × `Dist` and `CGoal` × `Dist`). We fixed the MAXSTEPS parameter to 100000, 3500, 2000 and 400 for Defibrillator, MMS, Loader, and DemoTopology respectively. BRANCHLENGTH was set to MAXSTEPS/5 and MAXSTEPS/20. These values were determined from the incomplete full BFS runs over these models as a trade-off between runtime and number of mutants reached. These experiments were performed on Machine 2.

Figure 6.2 shows the result of large models as a box-plot of the number of mutants reached in relation to the combination of heuristics. Note that the plot scales from minimum number of mutants reached to the maximum number of mutants reached. The DemoTopology model was excluded from this figure, because we found that every heuristic is able to reach all mutants

within 400 exploration steps. In that sense, it classifies as a small model. However, as we see in the full BFS results, it has many states, making it a large model.

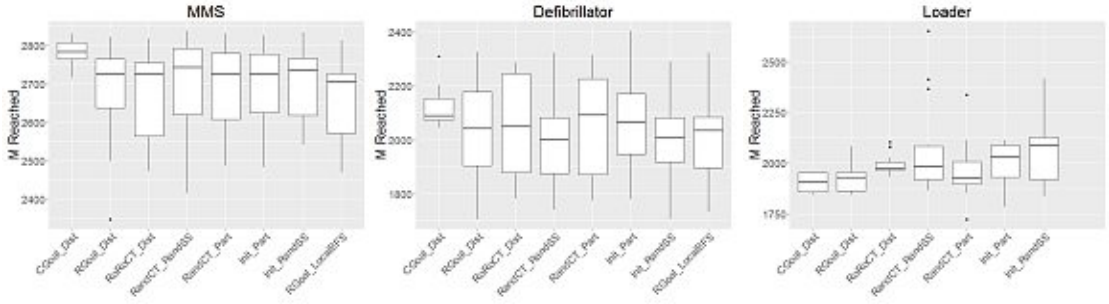


Figure 6.2: An evaluation of SELECTSUCCESSOR and CREATETASK heuristics on large models.

Like on small models, `CGoal` performs well on MMS and Defibrillator, not only producing the best mean result, but also showing much less variability than the other heuristics. Interestingly, the heuristic performs worst on Loader in terms of mean result. This underlines again, that multiple heuristics have merit, and there is no perfect heuristic for every model. `Init` and `Part` seem to perform better than predicted on small models. For `Part` this makes sense, since the number of objects increases in larger models, which naturally increases the power of the heuristic. `Init` performs reasonably on larger models. It seems that when branches are long enough, it can make sense to restart the search at the initial state. For `LocalBFS`, we had to reduce the depth from 4 to 2 of each local BFS search, in order not to blow up the search space. We had to exclude `LocalBFS` for Loader altogether, because even a breadth-first search with depth 2 is not feasible within reasonable time on this model.

6.4.7 Branching

We evaluated the influence of the BRANCHLENGTH parameter on search results, by an experiment with different values on two large models, MMS and Defibrillator. We chose these models, because they are big enough so there is a lot of variation in the number of mutants reached, but the runs are fast enough so we are able to obtain many data points.

To test the parameter, we set up the following experiment. We set MAXSTEPS to 3500 and 100000 for MMS and Defibrillator respectively. We used `RandCT` and `RandSS`, as well as 4 different random seeds for each value for BRANCHLENGTH. The experiment was performed on Machine 1. Figure 6.3 shows the result of the experiments. For MMS medium branch lengths work best. For Defibrillator short branch lengths work best. Both results suggest that branching search is indeed justified.

6.4.8 Test Case Generation

We evaluate the test case generation algorithm and the produced tests with increasing graph size. To this end, we ran an experiment using the Defibrillator model on Machine 1. We iteratively increase MAXSTEPS to produce exploration models of increasing size and record test suite characteristics. The results of this experiment are shown in Table 6.8. Column **Graph**

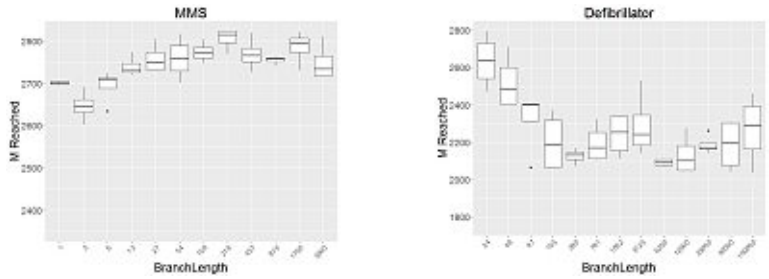


Figure 6.3: The mutants reached with multiple BRANCHLENGTH values.

Trans shows the size of the final exploration model in number of transitions. Unsurprisingly, the size of the exploration model increases as we increase the MAXSTEPS parameter. Column **Test Trans** shows the number of transitions of the final test suite. The number of test transitions does not grow as much as the size of the exploration model. This can be explained by the fact that after some point, no more mutants can be killed. However, it also shows that doing more exploration beforehand does not harm the final test suite by introducing unnecessary transitions. Column **Duplicate Trans** shows the number of transitions in the test suite that are taken at least twice. Initially, the ratio between duplicate transitions and test transitions is quite high. This ratio goes down with higher MAXSTEPS values, showing again that exploring more is better. Column **Time** shows the time in seconds needed for test case generation, once the exploration model is finalized. We can see that test case generation on large graphs takes a non-negligible time. For example for MAXSTEPS = 100 000 the search took 13 minutes and the test case generation took almost 4 minutes. However, for smaller graphs the amount of time spent is low.

MAXSTEPS	Graph Trans	Test Trans	Duplicate Trans	Time
100	683	334	107	0.002
200	1290	467	143	0.006
1 000	4342	659	138	0.042
10 000	21013	1581	260	0.524
20 000	28772	1918	271	2.420
50 000	49252	2117	320	17.217
100 000	63001	1904	291	231.032

Table 6.8: The test characteristics on Defibrillatorin relation to MAXSTEPS.

6.4.9 Parallelization

We evaluated how much speedup we could achieve by performing exploration steps in parallel. We define speedup as the fraction of average wall-clock time of parallel runs and the average wall-clock time of sequential runs. More specifically, we compare total runtimes of executions with 1 thread (sequential) to threads with 20 threads (parallel). We evaluated the speedup on all models but LBT, as running LBT on a single core is infeasible within a reasonable amount of time. For each model we fix MAXSTEPS and set BRANCHLENGTH to MAXSTEPS/20. Furthermore, we repeat the runs on each model using 4 different random seeds. The experiments were

performed on Machine 1.

Unsurprisingly, the larger models benefit more from parallelization. More precisely, models where one exploration step is computationally expensive benefit more. The model where a single exploration step takes the longest is Loader with 415 seconds per step on average. For Loader we get a speedup of 11.3x. On the other extreme, the average exploration step on PartCountDSL takes only 0.06 seconds. This model in fact suffers from the synchronization overhead of the parallel implementation and has a speedup of 0.9x. For large models we have an average speedup of 5.98x. Small models do not benefit much from parallelization, having an average speedup of 1.82x.

6.4.10 Finding Best Parameters

Based on our experiments, we describe an approach to finding the best parameters for our algorithm on a new model. First, the time budget that is acceptable for test case generation has to be set. Whether seconds, hours or days are acceptable makes a significant difference in what can be expected from the algorithm. Second, full BFS should be performed on the model to estimate the size of the model. If full BFS terminates, no other search option has to be tried. Otherwise, third, the amount of successful steps of the full BFS within the desired time budget should be recorded, which serves as a baseline for the MAXSTEPS parameter of further runs. Fourth, a few runs should be performed using the `CGoal` / `Dist` heuristic pair with different BRANCHLENGTH parameters to see whether more mutants can be found with small, medium, or long branch parameters. Fifth, different heuristic pairs should be evaluated. `RandCT` / `RandSS` can work well when not much structure of the model can be used. `CGoal` / `Part` can work well when the model uses much object orientation.

6.5 Related Work

6.5.1 Search- and Exploration-Based Testing

In this chapter we presented test case generation via branching search, which algorithmically is between search- and exploration-based testing. Classical search-based testing is defined as a search over the input space, whereas we define search over the state space. A comprehensive overview of search-based testing is provided in [McM04], both for the white-box and for black-box setting. A popular approach within search-based testing is casting test case generation as an optimization problem [Kor90, GN97, TCMM98, VMGF13, ZC05]. Another popular technique within search-based testing is genetic programming for gradually improving test suite quality, starting from a random test suite [WBP02, FA11, MF11]. Both techniques require executing and evaluating test cases multiple times, which is prohibitively expensive on our models.

An exploration-based mutation testing approach for pushdown automata has been presented in [BBTF11, BBTF12]. In the approach, tests are created from mutants via an exploration algorithm aiming to cover as many faulty transitions as possible. Killing analysis is performed a-posteriori by executing the resulting tests on the original model. Similarly, [KPLV+03] proposes exploration-based mutation testing for SDL specifications based on complete executions of test

suites on model and mutant. In contrast, our exploration algorithm performs lazy and on-the-fly mutation analysis to reduce mutation exploration costs. Our work extends previous work on exploration-based test case generation [ABJ+15a, ABJ+15b] with a richer search framework.

6.5.2 Guided Random Testing

In addition to search-based testing, our test case generation algorithm is similar to guided random testing. Guided random testing is performed in [MAZ+15] by introducing multiple techniques to enhance random testing with static and dynamic analysis information. A classic automated white-box test case generation tool is DART [GKS05], which combines random testing with symbolic execution-based guidance. Furthermore, Randoop [PE07, PLEB07] uses feedback from test executions to guide random automatic generation of Java unit tests. Whereas the above ideas are applied to code, we perform test case generation on models. REDIRECT [SYR08] is a model-based testing approach that applies guided random techniques to Simulink/Stateflow models. Apart from using a different class of models, the authors of [SYR08] target state and transition coverage, in contrast to the mutation coverage considered in this work.

Adaptive random testing [CLOM08, CKMT10] aims to distribute test inputs uniformly across the input data space. This is a similar idea to RRT, which tries to uniformly cover state spaces. However, the inherent difference to our work is that in contrast to simple input values, inputs of reactive models are sequences of actions. Therefore, we work with an infinite input space that is not easily classified via some simple metric.

6.5.3 Rapidly Exploring Random Trees

In terms of structuring our search, the main inspiration was the rapidly exploring random trees (RRT) algorithm [LaV98], which is a well established planning algorithm. RRT is mostly used in continuous domains [FKS06, JCS08, AS11], since it heavily depends on distance metrics, which naturally arise in such domains. However, RRT was used in testing of hybrid systems [DDD+15] and to solve well defined discrete planning tasks in [BCLM03]. We propose an RRT inspired search algorithm for a general purpose discrete setting and a novel set of heuristics, some of which are not distance-based.

CHAPTER 7

Event Structure-Based Test Case Generation

The complexity of modern systems often stems from the amount of independent actors contributing to it. Each individual actor can be simple, but by interacting there is a combinatorial explosion of possible global system behaviors. Envisioned applications using the internet of things are basically the embodiment of this explosion in complexity, where simple sensors and devices are supposed to form complex systems through collaboration and interaction. Managing this complexity is an important future challenge that is tackled in a large body of research on concurrent and distributed systems. Unfortunately, abstraction through modeling is only of limited use in this respect. While the functionality of individual actors can be simplified through abstraction, emerging behavior achieved through interaction of a large number of actors does not lend itself to this approach.

Fortunately, the interactions are seldom global in the sense that an action of one actor influences all other actors. In contrast, the actions of actors are largely and purposefully independent of each other. There is a long tradition in concurrency theory to exploit this independence by subsuming execution paths that correspond to different orderings of independent transitions of concurrent actors. Such subsuming traces are typically called Mazurkiewicz traces [Maz86]. Trace theory [Maz86] is a rich theoretical body of research on the topic. Furthermore, algorithms like partial order reduction [God96, CGMP99] and Lipton reduction [Lip75] offer practical solutions to the problem.

In order to cope with models of highly concurrent systems, we want to apply such techniques to our model-based mutation testing scenario. However, existing solutions typically lack an explicit representation of independent traces. Mazurkiewicz traces are typically represented as one representative trace, together with an independence relation. During exploration of Mazurkiewicz traces, only one representative trace is explored, while all others are ignored, since they provably lead to the same state or they provably satisfy the same logical formulas. Of course this is the

whole point of these techniques, but representations via representatives plus independence relation are impractical for mutation-driven test case generation. In particular, the strong killcheck requires to compare Mazurkiewicz traces of the model and the mutant. It is not obvious how to do this comparison while taking into account potentially differing independence relations. Furthermore, it is desirable also to retain independence information in the produced tests, i.e. to understand tests as Mazurkiewicz traces. To this end, an explicit and easy to interpret representation of such traces is desirable.

Event structures [Win88] are a representation of Mazurkiewicz traces as an explicit graph structure that lend themselves well both to comparing two systems as well as an explicit representation of tests as Mazurkiewicz traces. These structures stem from the Petri nets community, which is deeply involved in the study of concurrent systems. Event structures were used as a basis for testing concurrent systems before, see for example [KSH12, KSH15, KH18]. However, whereas classic event structure-based test case generation approaches construct the partial order semantics of the analyzed system on demand (e.g. via a SAT encoding provided in [EH08]), leveraging event structures for mutation testing requires to compare the complete partial order semantics of the model and mutant. Recently, [RSSK15] published a paper that describes how to obtain event structures from simple labeled transition systems and provides an algorithm that computes the complete partial order semantics of the transition system by constructing the set of all maximal configurations, which enabled us to integrate partial order semantics into our test case generation framework.

In this chapter, we recite parts of [RSSK15] and show how to apply it to action systems. We present a novel type of tests that we call concurrent tests, which essentially is the Mazurkiewicz trace for which a classic linear test is one representative. By representing sets of linear tests that are equivalent under a given independence relation, concurrent tests are strictly more powerful than classic linear tests. Concurrent tests induce less inconclusive test executions that occur when a test execution corresponds to a member of- instead of the representative of- the respective Mazurkiewicz trace. Furthermore, we show how to perform the strong killcheck using these structures. It turns out that this problem can be formulated as a language inclusion problem over event structures. The main contribution of this chapter is our proof of the computational complexity of this problem as well as a decision algorithm for it. Finally, we conduct an experimental evaluation of our language inclusion algorithm, comparing it to an explicit trace, automaton based language inclusion approach.

7.1 Event Structures and Configurations

In this section we introduce labeled prime event structures. Throughout this chapter, we assume that every set of labels $\mathcal{X}$ contains a distinct label ε, which denotes the empty symbol. Concatenation of ε to a word does not change the word.

Definition 7.1.1 (FLES)**.** Given a set of labels $\mathcal{X}$, a finite, $\mathcal{X}$-labeled prime event structure (FLES) is a tuple $\mathcal{E} \stackrel{\text{def}}{=} \langle E, <, \#, h \rangle$ where E is a finite set of events, $< \subseteq E \times E$ is a strict partial order on E, called *causality relation*, $h : E \to \mathcal{X}$ labels every event with an element of $\mathcal{X}$, and

$\# \subseteq E \times E$ is the symmetric, irreflexive *conflict relation* that is *closed under* $<$, i.e. for all $e, e', e'' \in E$, if $e \# e'$ and $e' < e''$, then $e \# e''$.

For an event e, we use $\lceil e \rceil$ to denote the history of e as the set of events that must happen before e according to $<$, formally $\lceil e \rceil \stackrel{\text{def}}{=} \{e' \in E \mid e' < e\}$. We require that there is a special event $\bot \in E$, such that $\lceil \bot \rceil = \emptyset$, for all events $e \in E : \bot < e$, and $h(\bot) = \varepsilon$. We define the direct successors dsucc of event e as the set of events that depend on e without there being another event in-between, formally $\mathsf{dsucc}(e) = \{e' \in E \mid e < e' \wedge \nexists e'' : e < e'' < e'\}$. We say that two events $e, e' \in E$ are *concurrent* if $e \neq e'$, not $(e < e')$, not $(e > e')$, and not $(e \# e')$. Two events e, e' are in *immediate conflict* denoted by $e \#^i e'$ if $e \# e'$ and both $\lceil e \rceil \cup \lceil e' \rceil$ and $\lceil e \rceil \cup \lceil e' \rceil$ are configurations, as defined in the following. An event $e \in E$ is *maximal* respectively *minimal* in $\mathcal{E}$, if there is no event $e' \in E$ such that $e < e'$ respectively $e' < e$.

A central concept in assigning event structures a semantic is the notion of configurations:

Definition 7.1.2 (Configuration)**.** For a FLES $\mathcal{E} \stackrel{\text{def}}{=} \langle E, <, \#, h \rangle$, a *configuration* of $\mathcal{E}$ is a set of events $C = \{e_1, \ldots, e_n\} \subseteq E$ that is both

- Left closed: $\forall e \in C : \forall e' \in E$ such that $e' < e \implies e' \in C$, and
- Conflict free: $\forall e, e' \in C : \neg(e \# e')$

A configuration C is *maximal*, if there is no configuration C' such that $C \subseteq C'$ and $C \neq C'$. We denote by $conf(\mathcal{E})$ the set of configurations of $\mathcal{E}$ and by $mconf(\mathcal{E})$ the set of all maximal configurations of an event structure $\mathcal{E}$. A *trace* tr of C is a sequence of events $\langle e_1, \ldots, e_n \rangle$, where every event $e \in C$ occurs exactly once in the sequence and for all $e_i, e_j \in \mathsf{tr} : e_i < e_j \implies i < j$. We denote the set of all traces of a configuration C with $T(C)$. Let $f : C \to X$ be a mapping on C to some set X. For a trace tr of C, we denote by $f(\mathsf{tr})$ the sequence resulting from point-wise application of f on the elements of tr. Finally, we extend T to event structures by defining it as the union of traces over all maximal configurations. That is, $T(\mathcal{E}) \stackrel{\text{def}}{=} \bigcup_{C \in mconf(\mathcal{E})} T(C)$.

A finite, labeled prime event structure $\mathcal{E}$ represents a finite set of bounded words over an alphabet $\mathcal{X}$, where the bound for the length of words is given by the size of the largest maximal configuration. We call this set the language $\mathcal{L}(\mathcal{E})$.

Definition 7.1.3 (Language of C and $\mathcal{E}$)**.** The language of configuration C of $\mathcal{E}$ is $\mathcal{L}(C) \stackrel{\text{def}}{=} \{h(\mathsf{tr}) \mid \mathsf{tr} \in T(C)\}$. The language of $\mathcal{E}$ is $\mathcal{L}(\mathcal{E}) \stackrel{\text{def}}{=} \{h(\mathsf{tr}) \mid \mathsf{tr} \in T(\mathcal{E})\}$.

To illustrate this definition we give a small example.

Example 7.1.1 (Event structure and configurations)**.** We show two event structures in Figures 7.1a and 7.1b. Boxes depict events. Inside every box is its event's identifier, above or below the box is its event's label. If there is no label we implicitly assume the label to be ε. Solid arrows depict direct successors of an event. Dashed lines depict immediate conflicts. Two

(a) An event structure with one maximal configuration.

(b) An event structure with conflicts.

Figure 7.1: Event structures examples.

events e, e' are in immediate conflict if $e \# e'$ and there are no $e_1, e_2 \in E$ such that $e_1 < e \wedge e_1 \# e'$ or $e_2 < e' \wedge e \# e_2$. For better readability, we omit all other causalities and conflicts.

Figures 7.1a and 7.1b both represent the language $\{AB, BA\}$. The event structure in Figure 7.1a has a single maximal configuration consisting of events $\{\perp, e_1, e_2\}$. The event structure in Figure 7.1b has two maximal configurations: $\{\perp, e_1, e_2\}$ and $\{\perp, e_3, e_4\}$ (due to the conflict between e_1 and e_3 these two events cannot appear in the same configuration).

Configuration as an event structure

Given an event structure $\mathcal{E} = \langle E, <, \#, h \rangle$ and a configuration C, we denote its corresponding event structure as $\mathcal{E}^C \stackrel{\text{def}}{=} \langle C, <_{\lceil C \times C}, \varnothing, h_{\lceil C} \rangle$, where $X_{\lceil Y}$ denotes the restriction of X to Y. For ease of presentation, throughout this chapter, we abuse notation and do not differentiate between a configuration and its corresponding event structure.

A note on cut-offs

Cut-offs are a popular method for unfoldings (such as event structures) to represent cyclic domains [McM92a]. Roughly speaking, one event is a cut-off of another event, if the former represents the same state as the latter with a smaller sub-structure. The latter event is then omitted from the structure. Cut-offs preserve reachability properties, but significantly increase the (computational) complexity of problems defined over unfoldings. For example, [Hel00] shows that the complexity of multiple model checking problems (including the language membership problem studied below) over unfoldings increase from NP-complete to PSPACE-complete when cut-offs are considered. An alternative approach to model checking over unfoldings with cut-offs is to expand them into exponentially larger structures and performing model checking over this structure [EH00, EH08]. Our application of event structures does not require cut-offs. Therefore, we do not consider them in this work. In particular, we do not use the results of [RSSK15] on cut-offs. However, the extension of our results to include cut-offs is interesting future work.

7.2 Unfolding Based Partial Order Reduction

In this section we show how to transform action systems to event structures on the fly and thereby perform unfolding based partial order reduction. The main concepts presented in this

section were developed in [RSSK15]. For the sake of self-containedness, we recite the relevant concepts of [RSSK15] in the following Section 7.2.1, where we only modify some notation in order to remove conflicts with the rest of this work. Thereafter, we show how to apply the procedure to our setting by concretizing aspects that are left abstract in [RSSK15].

7.2.1 Execution Model and Partial Order Reduction

An *abstract transition system* is a tuple $\mathcal{ATS} \stackrel{\text{def}}{=} \langle S, T, \tilde{s} \rangle$, where S is a set of *global states*, T is a set of *transitions*, and $\tilde{s} \in S$ is the *initial global* state. A transition is *enabled* at a state s if $t(s)$ is defined and $enabl(s)$ denotes the set of transitions enabled at s. Given two states $s, s' \in S$ and sequence of transitions $\sigma \stackrel{\text{def}}{=} \langle t_1, t_2, \ldots, t_n \rangle \in T^*$, we denote by $s \xrightarrow{\sigma} s'$ the fact that there exists states $s_0, \ldots, s_n \in S$ such that $s_0 = s, s_n = s'$ and for $i \in [1, n] : s_i = t(s_{i-1})$. Note that since transitions of abstract transition systems are deterministic, the successor state after a sequence of transitions is uniquely given. We denoted by $state(\sigma)$ the successor state of the initial state $\tilde{s}$ after σ and by $reach(\mathcal{ATS})$ the set of successor states of the initial state $\tilde{s}$ after any sequence of transitions.

Given two transitions $t, t' \in T$ and state $s \in S$, we say that t, t' *commute at* s iff

- if $t \in enabl(s)$ and $s' = t(s)$, then $t' \in enabl(s)$ iff $t' \in enabl(s')$; and
- if $t, t' \in enabl(s)$, then there is a state s' such that $s \xrightarrow{\langle t, t' \rangle} s'$ and $s \xrightarrow{\langle t', t \rangle} s'$.

An *unconditional independence relation* on $\mathcal{ATS}$ is a symmetric and irreflexive relation $\diamond \subseteq T \times T$ such that $t \diamond t'$, then t and t' commute at every state $s \in reach(\mathcal{ATS})$. If t, t' are not independent according to $\diamond$, then they are dependent, denoted by $t \circledast t'$.

In [RSSK15], partial order reduction semantics of an abstract transition system $\mathcal{ATS}$ together with an independence relation $\diamond$ are given by a canonical event structure $\mathcal{U}_{\mathcal{ATS},\diamond}$. This event structure is termed an unfolding of $\mathcal{ATS}$ and represents all valid transition sequences as well as reachable states. Furthermore, [RSSK15] provide an algorithm for computing the set of all maximal configurations from an event structure, which can also include cut-offs. This algorithm is optimal in the sense that it visits every maximal configuration exactly once, which is akin to classical partial order reduction algorithms being optimal when exploring each Mazurkiewicz trace exactly once.

7.2.2 Unfolding Action Systems

We now show how to apply the partial order semantics of [RSSK15] to action systems. To this end, we show how action systems can be interpreted as abstract transition systems and we provide an independence relation for action systems. We use these semantics and the configuration exploration algorithm given in [RSSK15] to construct canonical event structures $\mathcal{U}_{\mathcal{ATS},\diamond}$ on the fly, by replacing queries for given extensions of a configuration (denoted by $ex(C)$ in [RSSK15]) with extended successor path computation $extpath_{\mathcal{A}}(.,.)$ and insertion of new events from the results of this computation.

Action System as Abstract Transition System

Clearly, action systems can be interpreted as abstract transition systems, where transitions are given by the small step semantics presented in Section 3.4. Formally, let $\mathcal{A} = \langle \mathcal{V}, s_\iota, Act, A_\iota \rangle$ be an action system. Its corresponding abstract transition system is $\mathcal{ATS}^{\mathcal{A}} \stackrel{\text{def}}{=} \langle S^{\mathcal{A}}, T^{\mathcal{A}}, s_\iota^{\mathcal{A}} \rangle$. Note that the translation (unsurprisingly) proceeds analog to the Mealy machine translation presented in Section 3.4.3. However, in contrast to the Mealy machine translation, which is defined over visible successor paths and visible labels, the abstract transition system corresponding to an action system is defined over extended successor paths and the full set of action labels. We include internal labels to the abstract transition system, because they are necessary to correctly reconstruct effects on states and to resolve non-determinism.

Every visibly reachable state of the action system induces a state in the abstract transition system. In addition, we introduce states to the abstract transition system that correspond to intermediate steps of extended successor paths. Formally, every visibly reachable state $s \in visStates_{\mathcal{A}}$ induces the following set of abstract transition system states:

$$S_s^{\mathcal{A}} \stackrel{\text{def}}{=} \{s^{\mathcal{A}}\} \cup \{s_1^\pi, \ldots, s_{n-1}^\pi \mid \pi \in extpath_{\mathcal{A}}(s), |\pi.l| = n\}$$

The set of Mealy machine states is the union over induced states of all visibly reachable states:

$$S^{\mathcal{A}} \stackrel{\text{def}}{=} \bigcup_{s \in visStates_{\mathcal{A}}} S_s^{\mathcal{A}}$$

The initial state of the abstract transition system simply is the translation of the action system initial state $s_\iota^{\mathcal{A}}$. For every action label ℓ, there is a transition function $t_\ell \in T^{\mathcal{A}}$ that correspond to executing the respective action. That is, $t_\ell(s) = s'$ if and only if there is a visibly reachable state $s_0 \in visStates_{\mathcal{A}}$ and an extended successor path $\pi \in extpath_{\mathcal{A}}(s_0)$ such that s' corresponds to the successor of s after label ℓ, i.e. formally such that for some $i \in [1, |\pi.l| - 1] : s = s_{i-1}^\pi, \pi.l[i] = \ell$, and $s' = s_i^\pi$, or $s = s_{n-1}^\pi, \pi.l[n] = \ell$, and $s' = \pi.s^{\mathcal{A}}$.

An Independence Relation for Action Systems

In order to perform unfolding based partial order reduction on action systems, we define an independence relation $\diamond$ based on variable reads and writes (defined in Section 3.4). To this end, we define the variables written and read by a transition t_ℓ as $w(t_\ell) \stackrel{\text{def}}{=} \{v \mid \ell \text{ writes } v\}$ respectively $r(t_\ell) \stackrel{\text{def}}{=} \{v \mid \ell \text{ reads } v\}$. We say that transitions t and t' are in a *read/write conflict* if and only if

$$(w(t) \cap w(t')) \cup (w(t) \cap r(t')) \cup (r(t) \cap w(t')) \neq \emptyset.$$

Due to the sequential and prioritized composition in action systems, the absence of read/write conflicts does not fully capture independence of action system induced transitions. We say that two transitions t_{ℓ_1} and t_{ℓ_2} are *sequentially dependent*, if the respective actions ℓ_1 and ℓ_2 are composed sequentially in the action system. To see how prioritized composition influences transition dependence, suppose some transition t writes a variable and thereby enables another transition t', i.e. some guard inside t' is now satisfied. Transition t therefore disables all transitions over which transition t' is prioritized. Therefore, we define the set of *prio reads* as

$pr(t_\ell) \stackrel{\text{def}}{=} \bigcup_{\ell' \text{ is prioritized over } \ell} r(\ell')$. We say that transitions t and t' are in a *prio sensitive read-/write conflict* if and only if

$$(w(t) \cap w(t')) \cup (w(t) \cap (r(t') \cup pr(t'))) \cup ((r(t) \cup pr(t)) \cap w(t')) \neq \varnothing.$$

Finally, we define the independence relation $t \diamond t'$ if and only if t and t' are not sequentially dependent and not in a prio sensitive read/write conflict.

Lemma 7.2.1. $\diamond \subseteq T^{\mathcal{A}} \times T^{\mathcal{A}}$ is an unconditional independence relation. That is, $\diamond$ is symmetric and irreflexive, and if $t \diamond t'$ then t and t' commute at every state $s \in S^{\mathcal{A}}$.

Proof. The definition of $t \diamond t'$ is symmetric in t and t'. Therefore, $\diamond$ is symmetric. Furthermore, if $t = t'$, then $(w(t) \cap w(t')) \cup (w(t) \cap (r(t') \cup pr(t'))) \cup ((r(t) \cup pr(t)) \cap w(t')) = \varnothing$, i.e. $t \circledast t'$. Therefore, $\diamond$ is irreflexive.

The fact that independent transitions commute follows from the definition of the small step semantics of action systems (Table 3.2). In particular, whether and at which positions action system labels appear in extended successor paths solely depends the evaluation of guards in the current state (i.e. the values of variables in that state), the position in a sequential composition, and whether the respective action is blocked by an enabled action that is prioritized over it. Transitions that are not in a prio sensitive read/write conflict do not change any variable appearing in the guards of the other transition, are not sequentially composed, and do not enable or disable actions that are prioritized over the action underlying the other transition. Therefore, transitions that are not in a prio sensitive read/write conflict must commute.

□

7.3 Event Structure Based Test Case Generation

In the previous section, we discussed how to perform unfolding based partial order reduction to explore action systems. In this section, we show how to embed this procedure into the branching search based test case generation method presented in Chapter 6.

Partial Unfolding

The key idea of branching search is to break the search apart into small steps in order to handle very large models and to parallelize the exploration. We want to retain this property in event structure based test case generation by unfolding action systems stepwise. In Section 3.4, we noted that unrolling paths until they can only be extended by controllable actions is the right granularity to explore action systems in a mutation-driven test case generation context. Therefore, instead of unfolding $\mathcal{ATS}^{\mathcal{A}^{\mu_j}}$ fully at once, we instead unfold abstract transition systems that correspond to the extended paths for a single state s. Formally, let us define for a visible reachable state s, its abstract transition system $\mathcal{ATS}_s^{\mathcal{A}^{\mu_j}} \stackrel{\text{def}}{=} \langle S_s^{\mathcal{A}^{\mu_j}} \cup \{s' \in S^{\mathcal{A}^{\mu_j}} \mid \exists \pi \in extpath_{\mathcal{A}^{\mu_j}}(s) : \pi.s = s'\}, T_s^{\mathcal{A}^{\mu_j}}, s^{\mathcal{A}^{\mu_j}} \rangle$, where $S_s^{\mathcal{A}^{\mu_j}}$ is defined above in the general translation of action systems

to abstract transition systems and where $T_s^{\mathcal{A}^{\mu_j}}$ is defined as $T^{\mathcal{A}^{\mu_j}}$ restricted to $S_s^{\mathcal{A}^{\mu_j}}$. We denote the event structure constructed resulting from unfolding $\mathcal{ATS}_s^{\mathcal{A}^{\mu_j}}$ by $unfold(j, s)$.

We embed stepwise unfolding into the branching search algorithm by replacing $vispath_{\mathcal{A}}(j, s)$ with $unfold(j, s)$. In contrast to $vispath_{\mathcal{A}}(j, s)$ that contains tuples of paths and successor states, $unfold(j, s)$ is an event structure that can be interpreted as a set of configurations. Since every configuration corresponds to a set of paths that are equivalent with respect to commutativity according to the independence relation as well as a unique state, $unfold(j, s)$ is strictly more general than $vispath_{\mathcal{A}}(j, s)$. However, since paths are represented in a bundled form as configurations in the result of $unfold(j, s)$, kill checking and test case generation have to be adapted to handle event structures. The naive approach is to simply unpack the paths, i.e. calculate all interleavings represented by the event structure, and apply the path-wise methods. However, this approach (partially) defeats the purpose of performing partial order reduction, as the number of paths represented by an event structure in general is exponential in its size. Therefore, in the following we show how test case generation and kill checking can be done explicitly on the event structure.

Internal Action Labels

Event structures serve two main functions during test case generation. Firstly, they are used to drive unfolding based partial order model exploration. This requires to calculate states from the event labels. In particular, internal labels are important to determine the effect on states and to resolve non-determinism. Secondly, as we will discuss next, event structures express the input-output sequences of the model. This requires to skip over internal labels. In order to serve both functions, during event structure based test case generation, we maintain two types of labels for each constructed event structure. One version includes internal labels and is used to construct states, the other version replaces internal labels with ε and is used to represent input-output sequences via the language of the event structure. In the rest of this work, if not specified otherwise, we understand the language of an event structure as being defined over labels with ε representation of internal labels. The following proposition states the correctness of the approach:

Proposition 7.3.1. For every state s and mutant j, we have $\mathcal{L}(unfold(j, s)) = vispath_{\mathcal{A}}(j, s)$.

Proof. The proposition follows from the completeness and correctness of the unfolding semantics, (Theorem 5 as well as Lemma 16 of [RSSK15]) the completeness of the unfolding algorithm (Theorem 11 of [RSSK15]), the fact that the unfolded transition system is defined exactly over the extended successor paths, and the reduction of internal actions to ε event structure labels.

□

Language Inclusion Based Strong Killcheck

The classic strong killchecking with explicit paths boils down to a simple subset check of the form $\Pi^{\mu} \not\subseteq \Pi$. However, by using unfolding based partial order reduction during exploration,

instead of explicit paths, for killchecking during event structure based model exploration, we need to compare an event structure $\mathcal{E}$ that is a partial unfolding of the model and an event structure $\mathcal{E}^{\mu}$ that is a partial unfolding of some mutant. The key insight to performing this task on the event structures directly is that input-output sequences can be represented as words in the language of an event structure. Therefore, the killcheck can be done via a language inclusion check $\mathcal{L}(\mathcal{E}^{\mu}) \subseteq \mathcal{L}(\mathcal{E})$.

7.3.1 Concurrent Exploration Model

In order to record search results that are produced via unfolding based partial order reduction in the test case generation framework presented in Chapter 6, we replace the exploration model with a *concurrent exploration model*. In contrast to representing the exploration model as an extended Mealy machine, a concurrent exploration model is a tuple $G = \langle V, v_0, E, lconf, lsk, lwk, Vert \rangle$, where (V, E) is a directed labeled graph with set of vertices V and set of edges $E \subseteq (V \times V)$, v_0 is the initial vertex, $lconf$ labels vertices with maximal configurations of partial unfoldings of the model, lsk, lwk label vertices with strong and weak kills respectively, and a function $Vert : visStates_{\mathcal{A}} \to V$ that translates visibly reachable action system states to vertices in the concurrent exploration model. The concurrent exploration model is initialized as $\langle \{v_0\}, v_0, \varnothing, [v_0 \mapsto \varnothing], [v_0 \mapsto \varnothing], [v_0 \mapsto \varnothing], [s_\iota \mapsto v_0] \rangle$.

Let $\mathcal{C}$ be the set of maximal configurations of $unfold(0, s)$. For each $C \in \mathcal{C}$, we add a fresh vertex v_C to V, set its configuration label $lconf(v_C) \stackrel{\text{def}}{=} C$, add an edge $(Vert(s), v_C)$, initialize the killing labels $lsk(v_C) \stackrel{\text{def}}{=} \varnothing$, $lwk(v_C) \stackrel{\text{def}}{=} \varnothing$. Furthermore, in case $state(C)$ (the state obtained by executing any transition interleaving represented by C) is not in the domain of $Vert$, we update the state to vertex map $Vert(state(C)) \stackrel{\text{def}}{=} v_C$.

7.3.2 Concurrent Tests

Event structure-based model exploration enables a novel type of test cases that captures multiple interleavings of paths reaching some test goal. In contrast, classic linear tests represent only a single execution interleaving. Linear tests are not ideal for concurrent systems, because the exact interleaving of concurrent actions can not always be controlled. This may lead to unnecessary inconclusive or even failing test executions. A concurrent test essentially is a conflict free slice of the complete event structure of a model.

Definition 7.3.1 (Concurrent test)**.** A *concurrent test* for some action system $\mathcal{A}$ is a conflict free finite labeled prime event structure $\mathcal{E} = \langle E, <, h \rangle$, such that its input-output interleaved words are tests, i.e. $io(\mathcal{L}(\mathcal{E})) \subseteq \mathcal{T}_{st}(\mathcal{A})$. Every test execution $io(\omega)$ for $\omega \in \mathcal{L}(\mathcal{E})$ *passes* and every test execution $io(\omega)$ for $\omega \notin \mathcal{L}(\mathcal{E})$ *fails*.

In essence, a concurrent test is a configuration such that all its words represent passing tests. Similar to linear tests, we can represent inconclusive output information in concurrent tests to make them partially adaptive. Inconclusive output information for concurrent tests correspond to events that are in conflict with the configuration that represent passing tests and are thus failing.

Definition 7.3.2 (Partially adaptive concurrent test)**.** A *partially adaptive concurrent test* for some action system $\mathcal{A}$ is a finite labeled primed event structure $\mathcal{E} = \langle E, <, \#, h\rangle$, together with a maximal configuration C of $\mathcal{E}$ that represents passing tests executions, such that its input-output interleaved words are tests, i.e. $io(\mathcal{L}(\mathcal{E})) \subseteq \mathcal{T}_{st}(\mathcal{A})$. Every test execution $io(\omega)$ for $\omega \in \mathcal{L}(C)$ *passes*, every test execution $io(\omega)$ for $\omega \in \mathcal{L}(\mathcal{E}) \setminus \mathcal{L}(C)$ is *inconclusive*, and every test execution $io(\omega)$ for $\omega \notin \mathcal{L}(\mathcal{E})$ *fails*.

Concurrent tests can be extracted from the concurrent exploration model. To this end, let us define the concatenation $C_1 \circ C_2$ of two configurations C_1 and C_2 as the conflict free event structure $\mathcal{E} = \langle E, <, h\rangle$, where $E = C_1 \cup C_2$, $<$ is the union of causalities in C_1, C_2 as well as $\{(e_1, e_2) \mid e_1 \text{ is maximal in } C_1, e_2 \text{ is minimal in } C_2\}$, and h is the union of labels of C_1 and C_2.

Let $G = \langle V, v_0, E, \mathit{lconf}, \mathit{lsk}, \mathit{lwk}, \mathit{Vert}\rangle$ be a concurrent exploration model and let $\langle v_0, v_1, \ldots, v_n\rangle$ be a path in the graph, i.e. $(v_i, v_{i+1}) \in E$ for every $i \in [0, n-1]$. The concurrent test corresponding to this sequence is defined as $C_1 \circ C_2 \circ \ldots \circ C_n$. If $\mu \in \mathit{lsk}(v_i)$ respectively $\mu \in \mathit{lwk}(v_i)$ for some $i \in [0, n]$, the test potentially strongly- respectively weakly- kills mutant μ.

Partially adaptive concurrent test are extracted from the concurrent exploration model similarly, where, in addition to the concurrent test, immediate conflicts are extracted from the neighboring vertices of the test defining sequence $\langle v_0, v_1, \ldots, v_n\rangle$.

Example 7.3.1. Consider again the coffee brewing machine model presented in *Example* 3.4.3. The event structure representation of this example is depicted in *Figure* 7.2. In the figure, for internal actions labels, the respective two types of event structure labels are separated with a dash. Transitions $t_{\mathbf{obs}\,\mathsf{beans}}$, $t_{\mathbf{obs}\,\mathsf{sugar}}$, and $t_{\mathbf{obs}\,\mathsf{water}}$ are pairwise independent, because they are not in a prio sensitive read/write conflict, nor sequentially dependent. Since there are no conflicts in the event structure, it directly represents a concurrent test. In contrast to linear tests, the interleaving of **obs** beans, **obs** sugar, and **obs** water is not fixed in the concurrent test.

In order to demonstrate conflicts, we extend the example in *Figure* 7.3 with a non-deterministic choice to add coffee beans or tea leafs to the beverage. The respective events e_4 and e_3 are in conflict, depicted with a dashed line. We assume that the underlying action system is given in such a way that these options are mutually exclusive (for example by checking and writing some variable indicating whether beans or leafs were already added to the beverage). As a result, the event structure represents two concurrent tests, corresponding to $[e_{19}]$ and $[e_{20}]$. The respective test can be extended to partially adaptive concurrent tests by including the event e_4 respectively e_3 as well as its conflict.

7.4 Language Inclusion Problem and Complexity Results

In the previous section, we discussed how to leverage unfolding based partial order reduction for test case generation and cast the strong mutant killcheck as a language inclusion problem over event structures. In the remainder of this chapter, we discuss the theory behind this problem. In particular, we define it formally, prove its complexity, and provide a decision algorithm for it.

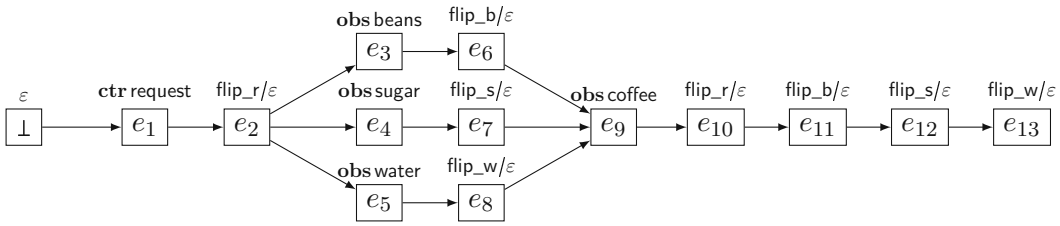


Figure 7.2: The event structure representation of a coffee brewing machine.

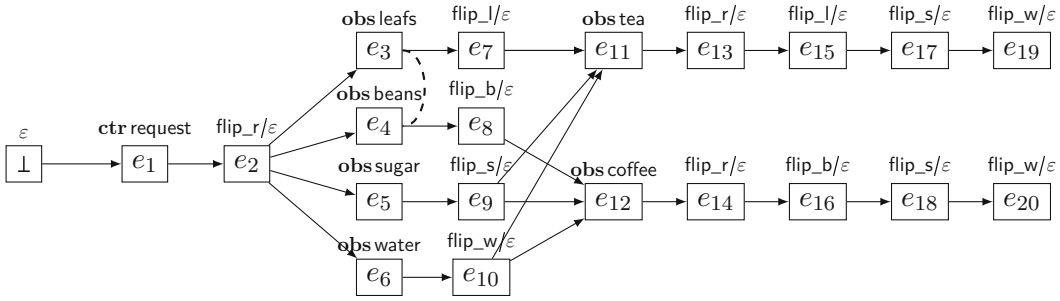


Figure 7.3: The event structure representation of a coffee and tea brewing machine with conflicting events.

In this section, we prove the computational complexity for the language inclusion problem. As an intermediate step we look at the membership problem.

7.4.1 Language Membership is NP-complete

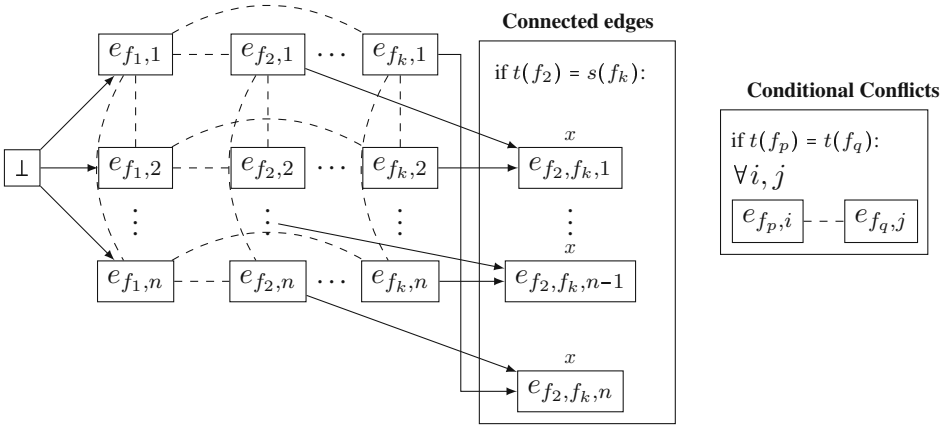
The *finite prime event structure language membership problem* for word w and FLES $\mathcal{E}$ is the problem of deciding whether $w \in \mathcal{L}(\mathcal{E})$. Surprisingly, deciding membership is NP-complete. In contrast, trace membership $\mathtt{tr} \in T(\mathcal{E})$ can be decided in polynomial time. Trace membership can be decided simply by verifying that the set of events of $\mathtt{tr}$ forms a maximal configuration of $\mathcal{E}$, which requires to verify left-closure, conflict-freedom, and maximality. All of those can be checked in polynomial time (linear time, assuming linear conflict lookup).

Intuitively, the hardness of language membership comes from the fact that the labeling function does not need to be injective and the role of conflicts, which together rule out a greedy algorithm that consumes the word in question symbol by symbol in a unique way.

Theorem 7.4.1. Finite prime event structure language membership is in NP.

Proof. Let $\mathcal{E} = \langle E, <, \#, h\rangle$ be an $\mathcal{X}$-labeled FLES and $w = \langle \sigma_1, \ldots, \sigma_n\rangle \in \mathcal{X}^*$ be a word. A trace $\mathtt{tr}$ is a polynomially sized certificate for $w \in \mathcal{L}(\mathcal{E})$. Checking that $\mathtt{tr} \in T(\mathcal{E})$ can be done in polynomial time, and checking whether $h(\mathtt{tr}) = w$ can be done in linear time. □

To prove NP-hardness we reduce the Hamiltonian cycle (HC) problem to the membership problem. HC is known to be NP-hard [Kar72]. It is the problem of deciding whether for a directed



All $e_{f_1,1}, \ldots, e_{f_k,1}, e_{f_1,n}, \ldots, e_{f_k,n}$ are causally related to $\perp$

Figure 7.4: $\mathcal{E}^G$ for Theorem 7.4.2.

graph there exists a path that visits all vertices once and that ends in the vertex it started. We use $s(f)$ and $t(f)$ to denote the source and target of a directed edge f.

Theorem 7.4.2. Finite prime event structure language membership is NP-hard.

Proof. We proof by reduction of HC to FLES language membership.

Let $G = (V, F)$ be a directed graph and define $n \stackrel{\text{def}}{=} |V|$. We assume that G does not contain any self loops, i.e. edges f such that $s(f) = t(f)$, and that $n > 1$. Apart from a graph with only one vertex, a graph contains a Hamiltonian cycle if and only if the same graph without self-loops contains a Hamiltonian cycle. The case $n = 1$ can be trivially decided and is therefore not considered in our reduction. A graph with self-loops can be converted into one without self-loops in linear time by removing self-loops from F.

Given an integer $j \in [1, n]$, we abbreviate $(j \bmod n) + 1$ by $\mathrm{su}(j)$. We say that f is connected to f' if $t(f) = s(f')$. We say that a sequence of edges $\langle f_1, \ldots, f_n \rangle$ is a cycle if for every $j \in [1, n] : f_j$ is connected to $f_{\mathrm{su}(j)}$. A Hamiltonian cycle of G is a cycle $\langle f_1, \ldots, f_n \rangle$ of F, such that $\{t(f_j) \mid j \in [1, n]\} = V$. The decision problem HC: "Does there exist a Hamiltonian cycle of G" is NP-hard [Kar72].

We provide a polynomially sized (in $|V| + |F|$) $\{\varepsilon, x\}$-labeled event structure $\mathcal{E}^G \stackrel{\text{def}}{=} \langle E, <, \#, h \rangle$, such that $x^n \in \mathcal{L}(\mathcal{E}^G)$ (n times the letter x concatenated) if and only if there exists a Hamiltonian cycle of G. The main idea is that configurations of the event structure translate to connected sequences of edges via events that each represent the assignment of an edge to a position in the sequence and events that express connectedness of two successive edges in the sequence. $\mathcal{E}^G$ is presented in Figure 7.4 as a visual aid for this proof.

The set of events E contains exactly the following **events**:

- The initial event $\perp$

- For every edge $f \in F$ and $j \in [1,n]$, there is an event $e_{f,j} \in E$ representing that f is the j'th element of a sequence of edges.
- For every pair of edges $f, f' \in F$ with $t(f) = s(f')$ and $j \in [1,n]$ there is an event $e_{f,f',j} \in E$ representing that f and f' are successive elements of a sequence of edges.

Formally, we define E as follows:

$$E \stackrel{\text{def}}{=} \{\bot\} \cup \bigcup_{j=1}^{n} \left(\{e_{f,j} \mid f \in F\} \cup \{e_{f,f',j} \mid f, f' \in F \wedge t(f) = s(f')\} \right)$$

Clearly, $|\mathcal{E}^G| = |E|$ is polynomial in $|F| \leq |G|$. Note that causality, conflict relation, and labeling are always of at most quadratic size in $|E|$.

The causality relation $<$ is the smallest partial order relation containing the following **causalities**:

- For every edge $f \in F$ and $j \in [1,n]$: $\bot < e_{f,j}$ representing that every assignment of a single edge to a position in a sequence of edges is allowed.
- For every pair of edges $f, f' \in F$ with $t(f) = s(f')$, and every $j \in [1,n]$: $e_{f,j} < e_{f,f',j}$ and $e_{f',\mathrm{su}(j)} < e_{f,f',j}$ representing that successive edges assigned in the represented sequence are connected.

Formally, we define $<$ as follows, where X^+ denotes the transitive closure of relation X:

$$< \stackrel{\text{def}}{=} \left(\bigcup_{j=1}^{n} \left(\{(\bot, e_{f,j}) \mid f \in F\} \cup \{(e_{f,j}, e_{f,f',j}), (e_{f',\mathrm{su}(j)}, e_{f,f',j}) \mid e_{f,f',j} \in E\} \right) \right)^+$$

The conflict relation $\#$ is the smallest conflict relation closed under $<$ (i.e. $e \# e' \wedge e' < e'' \Rightarrow e \# e''$) containing the following **immediate conflicts**:

C.1 For every edge $f \in F$ and $j, k \in [1,n], j \neq k$: $e_{f,j} \# e_{f,k}$ representing that every edge can only be assigned to one position in a sequence.

C.2 For every pair of edges $f, f' \in F, f \neq f'$ and every $j \in [1,n]$: $e_{f,j} \# e_{f',j}$ representing that every position in the sequence can only be assigned once.

C.3 For every pair of edges $f, f' \in F, f \neq f'$, such that $t(f) = t(f')$ and every $j, k \in [1,n]$: $e_{f,j} \# e_{f,k}$ representing that edges assigned to a sequence must have non-overlapping target vertices.

Formally, $\#$ has the following set of immediate conflicts:

$$\#^i \stackrel{\text{def}}{=} \bigcup_{j=1}^{n} \left(\bigcup_{i=1, i\neq j}^{n} \left(\{(e_{f,j}, e_{f,i}) \mid f \in F\} \cup \{(e_{f,j}, e_{f',i}) \mid f, f' \in F \wedge f \neq f' \wedge t(f) = t(f')\} \right) \right.$$
$$\left. \cup \{(e_{f,j}, e_{f',j}) \mid f, f' \in F \wedge f \neq f'\} \right)$$

The **labeling function** h is given as follows:

- Every event of the form $e_{f,f',j}$ has label $h(e_{f,f',j}) \stackrel{\text{def}}{=} x$.
- Every other event in E has label ε.

Formally, h is defined as follows:

$$h(e) \stackrel{\text{def}}{=} \begin{cases} x & \text{for } e = e_{f,f',j} \in E \\ \varepsilon & \text{otherwise} \end{cases}$$

We say that a configuration of $\mathcal{E}^G$ represents a sequence of edge $\langle f_1, \ldots, f_m \rangle$ if it includes events $e_{f_1,i_1}, \ldots, e_{f_m,i_m}$, where $\{i_1, \ldots, i_m\} \subseteq [1, n]$, for every $j \in [1, m-1] : i_j < i_{j+1}$.

Configurations represent sequences of edges: We claim that every configuration (besides $\{\perp\}$) of $\mathcal{E}^G$ represents a sequence of $m \leq n$ edges with pairwise different targets.

The claim follows from the structure of the immediate conflicts: Due to conflicts **C.1**, a configuration cannot contain events $e_{f,j}$ and $e_{f,k}$ for $j \neq k$. Therefore, for every edge, a configuration includes one such event or none. Furthermore, the indices j of events $e_{f,j}$ give rise to a sequence of represented edges. Due to conflicts **C.2**, every index can only be assigned to one position in the sequence. There are at most n possible positions. Therefore, every configuration (besides $\{\perp\}$) represents a sequence of $m \leq n$ edges. Furthermore, the edges must pairwise different targets due to conflicts **C.3**.

Configurations with events $e_{f,f',j}$ represent sequences of (partially) connected edges:

Due to the causes of events $e_{f,f',j}$, we have that a configuration contains $e_{f,f',j}$ if and only if it represents a sequence of edges $f_1, \ldots, f_j = f, f_{\text{su}(j)} = f', \ldots, f_m$.

Hamiltonian cycle $\Rightarrow x^n \in \mathcal{L}(\mathcal{E}^G)$:

Assume G has a Hamiltonian cycle $\langle f_1, \ldots, f_n \rangle$. We claim that $\perp \cup \{e_{f_j,j}, e_{f_j,f_{\text{su}(j)},j} \mid j \in [1, n]\}$ is a maximal configuration. The set is causally closed, since the edges are connected. The set is conflict free, because every position is assigned exactly once (no conflicts among **C.1** and **C.2**) and since the sequence is a Hamiltonian cycle, the targets are not overlapping (no conflict among **C.3**). The configuration is maximal, since clearly no further event of the form $e_{f,i}$ can be

added, since the assignment of edges to positions is fixed, and no event of the form $e_{f,f',i}$ can be added, since these events are directly induced by the assignment of edges to positions in the sequence. Furthermore, this maximal configuration contains exactly n events labeled by x and all other events are labeled by ε, showing $x^n \in \mathcal{L}(\mathcal{E}^G)$.

$x^n \in \mathcal{L}(\mathcal{E}^G) \Rightarrow$ **Hamiltonian cycle:**

If $x^n \in \mathcal{L}(\mathcal{E}^G)$, then $\mathcal{E}^G$ has a maximal configuration C that includes n events of the form $e_{f,f',j}$.

Assume that C does not represent a Hamiltonian cycle. That is, two successive edges in the represented sequence of edges that are not connected, or not all vertices are visited by the sequence.

The former case cannot be true, due to presence of events $e_{f,f',j}$ in C, showing that f is connected to f' and the causalities of such events ($e_{f,j}$ and $e_{f',\mathrm{su}(j)}$), implying that f and f' are successive edges in the sequence of edges represented by C.

To see why the latter case cannot be true, consider that in order for a connected sequence of n edges not to visit one of the n vertices of the graph, it needs to visit some vertex twice. That is, it needs to include edges that have the same target vertex. C can not represent two different edges with the same target due to conflicts **C.3**. Furthermore, C can not represent the same edge twice, due to conflicts **C.1**. Therefore, the sequence of edges represented by C can not contain two edges with the same target.

Therefore, the maximal configuration C represents a Hamiltonian cycle in G. □

7.4.2 Language Inclusion is Π_2^p-complete

The *finite prime event structure language inclusion problem* for FLES $\mathcal{E}_1$ and $\mathcal{E}_2$ is the problem of deciding whether $\mathcal{L}(\mathcal{E}_1) \subseteq \mathcal{L}(\mathcal{E}_2)$.

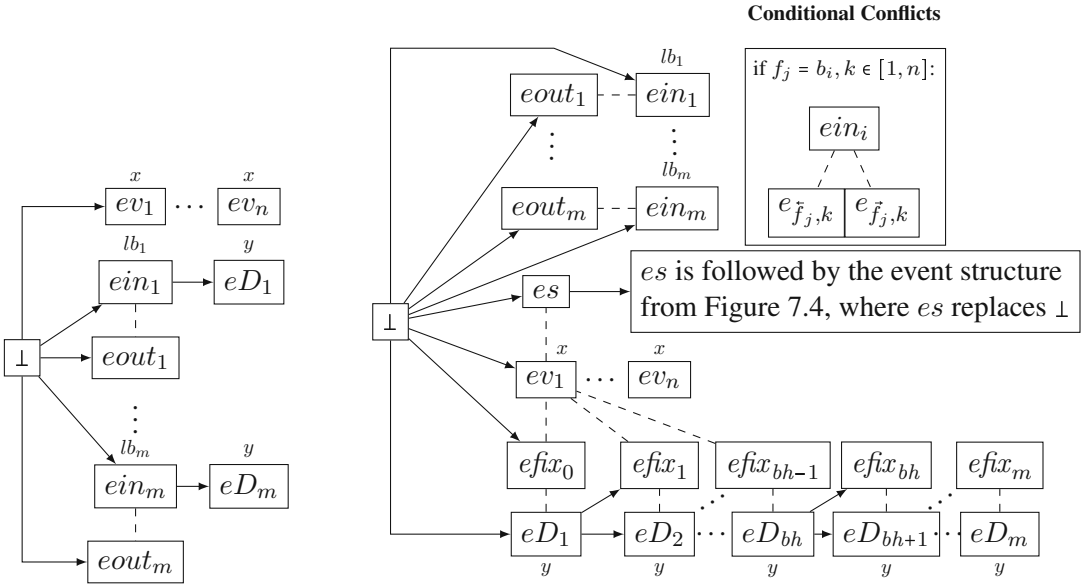
Π_2^p is a complexity class from the polynomial hierarchy. It intuitively represents a $\forall\exists$ quantifier alternation. To show inclusion, we use the definition of Π_2^p given by Wrathall [Wra76], providing semantics for the complexity class in terms of formal languages. These languages should not be confused with the particular type of languages we discuss in this work. In contrast, such languages encode problem instances and candidate witnesses.

Formally, a language L is in Π_2^p iff there exists a polynomially decidable language L', such that $x \in L \Leftrightarrow \forall y_1 \exists y_2 [\langle x, y_1, y_2\rangle \in L']$. A language L' is polynomially decidable if $w \in L'$ can be decided in polynomial time. The x represents an encoding of the problem instance as a string. The y_1 and y_2 represent string encodings of witnesses to some sub-problems.

We fix two $\mathcal{X}$-labeled FLES $\mathcal{E}_1 = \langle E_1, <_1, \#_1, h_1\rangle$ and $\mathcal{E}_2 = \langle E_2, <_2, \#_2, h_2\rangle$.

Theorem 7.4.3. Finite prime event structure language inclusion is in Π_2^p.

Proof. Language inclusion $\mathcal{L}(\mathcal{E}_1) \subseteq \mathcal{L}(\mathcal{E}_2)$ amounts to checking whether $\forall w \in \mathcal{L}(\mathcal{E}_1) \Rightarrow w \in \mathcal{L}(\mathcal{E}_2)$. In terms of traces this can be expressed as $\forall \mathsf{tr}_1 \in T(\mathcal{E}_1).\ \exists \mathsf{tr}_2 \in T(\mathcal{E}_2).\ h_1(\mathsf{tr}_1) = h_2(\mathsf{tr}_2)$, meaning that for every trace in $\mathcal{E}_1$ there has to be a trace in $\mathcal{E}_2$ corresponding to the same word in the common alphabet $\mathcal{X}$.



(a) $\mathcal{E}_1^{G,B}$. (b) $\mathcal{E}_2^{G,B}$.

Figure 7.5: The event structures for the language inclusion hardness proof.
We use ... to indicate omitted events.

We define $L \stackrel{\text{def}}{=} \{\langle \mathcal{E}_1, \mathcal{E}_2 \rangle \mid \mathcal{L}(\mathcal{E}_1) \subseteq \mathcal{L}(\mathcal{E}_2)\}$ and $L' \stackrel{\text{def}}{=} \{\langle\langle \mathcal{E}_1, \mathcal{E}_2 \rangle, \mathsf{tr}_1, \mathsf{tr}_2 \rangle \mid \mathsf{tr}_1 \in T(\mathcal{E}_1) \Rightarrow \big(h_1(\mathsf{tr}_1) = h_2(\mathsf{tr}_2) \wedge \mathsf{tr}_2 \in T(\mathcal{E}_2)\big)\}$. By the argument above, we obtain the desired form $x \in L$ iff $\forall y_1 \exists y_2 [\langle x, y_1, y_2 \rangle \in L']$ to show Π_2^p inclusion. Furthermore, L' can be decided deterministically in polynomial time, because trace membership, as well as label equality, can be decided in polynomial time. □

To show Π_2^p hardness, we present a reduction from the Dynamic Hamiltonian Cycle (DHC) problem to the finite prime event structure language inclusion problem. Given an undirected graph $G = (V, F)$ and a set $B \subseteq F$, graph G and B form a DHC if for every set $D \subseteq B$ with $|D| \leq |B|/2$, the graph $G_D = (V, F \setminus D)$ has a Hamiltonian cycle. We define $n \stackrel{\text{def}}{=} |V|$, $k \stackrel{\text{def}}{=} |F|$, $m \stackrel{\text{def}}{=} |B|$, and $bh \stackrel{\text{def}}{=} \lfloor |B|/2 \rfloor$. Essentially DHC, in comparison to HC, has an additional universal quantifier over subsets of B. DHC is known to be Π_2^p-complete [KL95].

Theorem 7.4.4. Finite prime event structure language inclusion is Π_2^p-hard.

Proof. We prove the claim by reduction of DHC to FLES language inclusion. Let $G = (V, F)$ be a finite, undirected graph and let $B \subseteq F$. Let $V = \{v_1, \ldots, v_n\}$, $F = \{f_1, \ldots, f_k\}$, and $B = \{b_1, \ldots, b_m\}$. Let $bh \stackrel{\text{def}}{=} \mathsf{floor}(\frac{|B|}{2})$. Using the same arguments of generality as in the proof of Theorem 7.4.2, we assume that G does not contain any self loops, i.e. edges f such that $s(f) = t(f)$, and that $n > 1$.

For this proof, we will use the same method to encode Hamiltonian cycles of directed graphs into event structures as in the proof of Theorem 7.4.2. Therefore, the first step of our reduction is to encode G as a directed graph $\vec{G} = (V, \vec{F})$ with $\vec{F} \stackrel{\text{def}}{=} \{\vec{f}, \overleftarrow{f} \mid f = (u, v) \in F \wedge \vec{f} = \langle u, v\rangle \wedge \overleftarrow{f} = \langle v, u\rangle\}$, where (u, v) and $\langle u, v\rangle$ denote undirected, respectively directed, edges between vertices u and v. A self loop-free graph G has an undirected Hamiltonian cycle if and only if $\vec{G}$ has a directed Hamiltonian cycle. Intuitively this is true, because we introduce for each undirected edge an edge in each direction that connect the vertices in either direction, just as the undirected edge does.

We provide $\{\varepsilon, x, y, lb_1, \ldots, lb_m\}$-labeled event structures $\mathcal{E}_1^{G,B} = \langle E_1, <_1, \#_1, h_1\rangle$ and $\mathcal{E}_2^{G,B} \stackrel{\text{def}}{=} \langle E_2, <_2, \#_2, h_2\rangle$ such that $|\mathcal{X}|, |\mathcal{E}_1^{G,B}| \stackrel{\text{def}}{=} |E_1|$, and $|\mathcal{E}_2^{G,B}| \stackrel{\text{def}}{=} |E_2|$ are polynomial in $|G| \stackrel{\text{def}}{=} |V|+|F|$ and G, B has a DHC if and only if $\mathcal{L}(\mathcal{E}_1^{G,B}) \subseteq \mathcal{L}(\mathcal{E}_2^{G,B})$. X^+ denotes the transitive closure of relation X.

We define the components of $\mathcal{E}_1^{G,B}$ as follows, where $\#_1^i$ denotes immediate conflicts:

$$E_1 \stackrel{\text{def}}{=} \{\bot\} \cup \bigcup_{i=1}^{n} \{ev_i\} \cup \bigcup_{i=1}^{m} \{ein_i, eD_i, eout_i\}$$

$$<_1 \stackrel{\text{def}}{=} \left(\{(\bot, ev_1)\} \cup \bigcup_{i=2}^{n} \{(ev_{i-1}, ev_i)\} \cup \bigcup_{i=1}^{m} \{(\bot, ein_i), (\bot, eout_i), (ein_i, eD_i)\}\right)^+$$

$$\#_1^i \stackrel{\text{def}}{=} \bigcup_{i=1}^{m} \{(ein_i, eout_i)\}$$

$$h_1(e) \stackrel{\text{def}}{=} \begin{cases} x & \text{for } e \in \{ev_1, \ldots, ev_n\} \\ lb_i & \text{for } e = ein_i \\ y & \text{for } e \in \{eD_1, \ldots, eD_m\} \\ \varepsilon & \text{otherwise} \end{cases}$$

For the definition of $\mathcal{E}_2^{G,B}$, we make use of the event structure $\mathcal{E}^G$ encoding all Hamiltonian cycles in $\vec{G}$ defined in the proof of Theorem 7.4.2 and embed it into $\mathcal{E}_2^{G,B}$ using a new root event $es \in E_2$:

$$E^{HC} \stackrel{\text{def}}{=} \bigcup_{j=1}^{n} \left(\{e_{f,j} \mid f \in \vec{F}\} \cup \{e_{f,f',j} \mid f, f' \in \vec{F} \wedge t(f) = s(f')\}\right)$$

$$<^{HC} \stackrel{\text{def}}{=} \bigcup_{j=1}^{n} \left(\{(es, e_{f,j}) \mid f \in \vec{F}\} \cup \{(e_{f,j}, e_{f,f',j}), (e_{f',\mathrm{su}(j)}, e_{f,f',j}) \mid e_{f,f',j} \in E^{HC}\}\right)$$

$$\#^{HC} \stackrel{\text{def}}{=} \bigcup_{j=1}^{n} \left(\bigcup_{i=1, i\neq j}^{n} \left(\{(e_{f,j}, e_{f,i}) \mid f \in \vec{F}\} \cup \{(e_{f,j}, e_{f',i}) \mid f, f' \in \vec{F} \wedge f \neq f' \wedge t(f) = t(f')\}\right) \right.$$
$$\left. \cup \{(e_{f,j}, e_{f',j}) \mid f, f' \in \vec{F} \wedge f \neq f'\}\right)$$

Finally, we define the components of $\mathcal{E}_2^{G,B}$ as follows, where $\#_2^i$ denotes immediate conflicts:

$$E_2 \stackrel{\text{def}}{=} \{\bot, es\} \cup \bigcup_{i=1}^{n} \{ev_i\} \cup \bigcup_{i=1}^{m} \{ein_i, eout_i, eD_i, efix_{i-1}\} \cup E^{HC}$$

$$<_2 \stackrel{\text{def}}{=} \Big(\{(\bot, es), (\bot, ev_1), (\bot, efix_0), (\bot, eD_1)\} \cup \bigcup_{i=2}^{n} \{(ev_{i-1}, ev_i)\} \cup \bigcup_{i=1}^{m} \{(\bot, ein_i), (\bot, eout_i)\} \cup \bigcup_{i=1}^{m-1} \{(eD_i, efix_i), (eD_i, eD_{i+1})\} \cup <^{HC} \Big)^+$$

$$\#_2^i \stackrel{\text{def}}{=} \{(es, ev_1)\} \cup \bigcup_{i=1}^{m} \{(ein_i, eout_i), (eD_i, efix_{i-1})\} \cup \bigcup_{i=1}^{bh} \{(efix_i, ev_1) \cup \#^{HC}\} \cup \bigcup_{i=1}^{m} \bigcup_{j=1}^{n} \{(ein_i, e_{\tilde{f}_i,j}), (ein_i, e_{\tilde{f}_i,j})\}$$

$$h_2(e) \stackrel{\text{def}}{=} \begin{cases} x & \text{for } e = e_{f,f',j} \in E_2 \text{ for some } f, f' \in F \text{ and } j \in [1,n] \\ x & \text{for } e \in \{ev_1, \ldots, ev_n\} \\ y & \text{for } e \in \{eD_1, \ldots, eD_m\} \\ lb_i & \text{for } e = ein_i \\ \varepsilon & \text{otherwise} \end{cases}$$

$\mathcal{E}_1^{G,B}$ and $\mathcal{E}_2^{G,B}$ are presented in Figure 7.5 as a visual aid for this proof. From the definition of $\mathcal{E}_1^{G,B}$ and $\mathcal{E}_2^{G,B}$, we can immediately see that the reduction is polynomial. Notice that the causality and the conflict relation, as well as the number of edges of the graph is always at most quadratic in the number of events and vertices in G.

Both event structures $\mathcal{E}_1^{G,B}$ and $\mathcal{E}_2^{G,B}$ encode subsets D of B. In particular, every word $w \in \mathcal{L}(\mathcal{E}_1^{G,B}) \cup \mathcal{L}(\mathcal{E}_2^{G,B})$ encodes a subset $D(w) \stackrel{\text{def}}{=} \{b_i \mid lb_i \in w\}$ of B.

Consider an arbitrary word $w \in \mathcal{L}(\mathcal{E}_1^{G,B})$. We first gather some properties of its symbols and then proceed to show under which condition $w \in \mathcal{L}(\mathcal{E}_2^{G,B})$ holds. Firstly, w must contain the symbol x exactly $|V|$ times due to events ev_i. Secondly, every event ein_i of $\mathcal{E}_1$ causes an event eD_i with label y. Therefore, $w \in \mathcal{L}(\mathcal{E}_1^{G,B})$ encodes the cardinality of $D(w)$ by the number of y symbols in w. Finally, since all events ev_i are concurrent to all events ein_j and all events ein_j are pairwise concurrent, symbols lb_i and x can appear in any permutation.

To show under which conditions $w \in \mathcal{L}(\mathcal{E}_2^{G,B})$ holds, we distinguish whether y occurs in w more often or less or equal than bh times.

Firstly, consider the case that w contains the symbol y more often than bh. That is, w encodes $D(w)$ with $|D(w)| > bh$. Since DHC does not require the existence of a Hamiltonian cycle in $G_{D(w)}$, the word should be contained in $\mathcal{L}(\mathcal{E}_2^{G,B})$ for any graph G. Consider the set of events $C_{D(w)} \stackrel{\text{def}}{=} \{\bot\} \cup \{ein_j \mid b_j \in D(w)\} \cup \{eout_j \mid b_j \notin D(w)\} \cup \lceil efix_{|D(w)|} \rceil \cup \{ev_1, \ldots, ev_n\}$. The set is a maximal configuration, since $efix_{|D(w)|}$ is not in conflict with ev_1, due to $|D(w)| >$

bh. Furthermore, $w \in \mathcal{L}(C_{D(w)})$ because the y-labeled events eD_i are pairwise concurrent with the x-labeled events $\{ev_1, \ldots, ev_n\}$ which are in turn pairwise concurrent with the lb_i-labeled $\{ein_j \mid b_j \in D(w)\}$ events. Furthermore, the lb_i-labeled events $\{ein_j \mid b_j \in D(w)\}$ are pairwise concurrent to each other. Therefore, $\mathcal{L}(C_{D(w)})$ includes exactly all permutations of these symbols, in particular $w \in \mathcal{L}(C_{D(w)})$. In summary, for words w which encode $D(w)$, such that $|D(w)| > bh$, we have $w \in \mathcal{L}(\mathcal{E}_2^{G,B})$.

Secondly, consider the converse case that w encodes a set $D(w)$ with $|D(w)| \leq bh$. Any maximal configuration $C_{D(w)}$ such that $w \in \mathcal{L}(C_{D(w)})$ must include events $\{\bot\} \cup \{ein_j \mid b_j \in D(w)\} \cup \{eout_j \mid b_j \notin D(w)\} \cup \lceil efix_{|D(w)|} \rceil$.

In contrast to the first case, the event ev_1 is in conflict with $efix_{|D(w)|}$, because $|D(w)| \leq bh$. However, the event es is not in conflict with $efix_{|D(w)|}$. Hamiltonian cycles are encoded in the sub-event structure following es, which can be seen by the arguments presented in the proof of Theorem 7.4.2. However, due to conflicts of events ein_i with $e_{\vec{f}_i,j}$ and $e_{\vec{f}_i,j}$, only events $e_{\vec{f},j}$ and $e_{\vec{f},j}$ can be included in $C_{D(w)}$ with $f \notin D(w)$, i.e. edges of the graph $G_{D(w)}$. Thus, analogous to the proof of Theorem 7.4.2, $C_{D(w)}$ include $|V|$ x-labeled events if and only if $G_{D(w)}$ has a Hamiltonian cycle.

In summary, for a word $w \in \mathcal{L}(\mathcal{E}_1)$, we have $w \in \mathcal{L}(\mathcal{E}_2)$ if and only if $|D(w)| > bh$ or $|D(w)| \leq bh$ and $G_{D(w)}$ contains a Hamiltonian cycle. Furthermore, for every $D \subseteq B$, there is a word $w \in \mathcal{L}(\mathcal{E}_1)$, such that $D = D(w)$.

Therefore, we get $\mathcal{L}(\mathcal{E}_1^{G,B}) \subseteq \mathcal{L}(\mathcal{E}_2^{G,B})$ if and only if G and B satisfy DHC. □

Example 7.4.1. Consider the graph shown in Figure 7.6a. The graph and the set of edges $\{b_1, b_2\}$ form a dynamic Hamiltonian cycle.

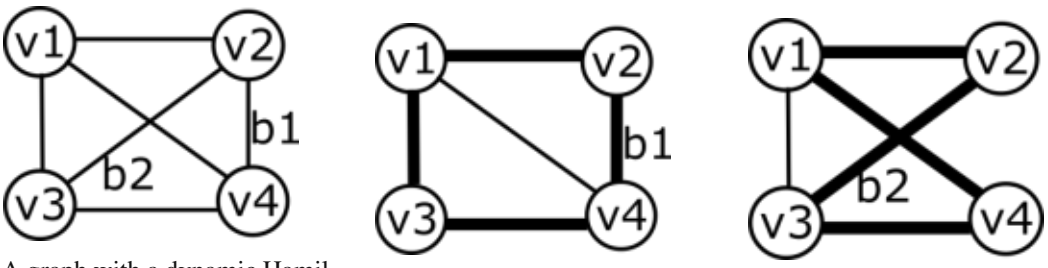
Figures 7.7 and 7.8 show the event structures $\mathcal{E}_1$ and $\mathcal{E}_2$ constructed according to the reduction in the proof of Theorem 7.4.4 on this example graph.

The bold events in Figures 7.7 and 7.8 show the configurations that correspond to the Hamiltonian cycle that can be obtained when b_2 is removed. The configuration where b_1 is removed is similar.

The remaining cases are removing none or both of b_1 and b_2. In the case that both are removed, the number of y labels is 2 and therefore eD_2 has to be part of the configuration, enabling $ev_1, \ldots, ev_4$ to generate 4 times x. In case no b_i is removed, the same configuration as in Figure 7.8 can be used to show existence of a Hamiltonian cycle with small changes to accommodate for the different number of y and the non-existence of lb_2.

7.5 Deciding Language Inclusion

In this section, we introduce a decision algorithm for the FLES language inclusion problem. Furthermore, we provide a language preserving translation of event structures into non-deterministic



(a) A graph with a dynamic Hamiltonian cycle.

(b) A cycle without in_2.

(c) A cycle without in_1.

Figure 7.6: An example demonstrating the dynamic Hamiltonian cycle problem.

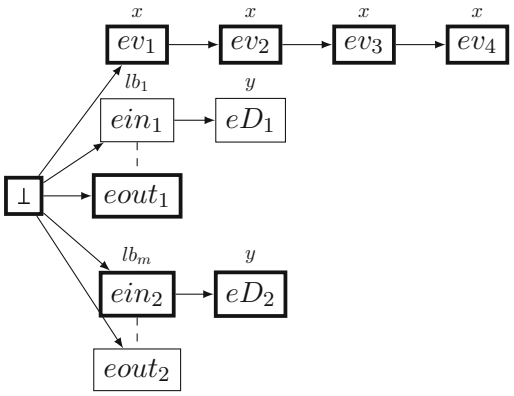
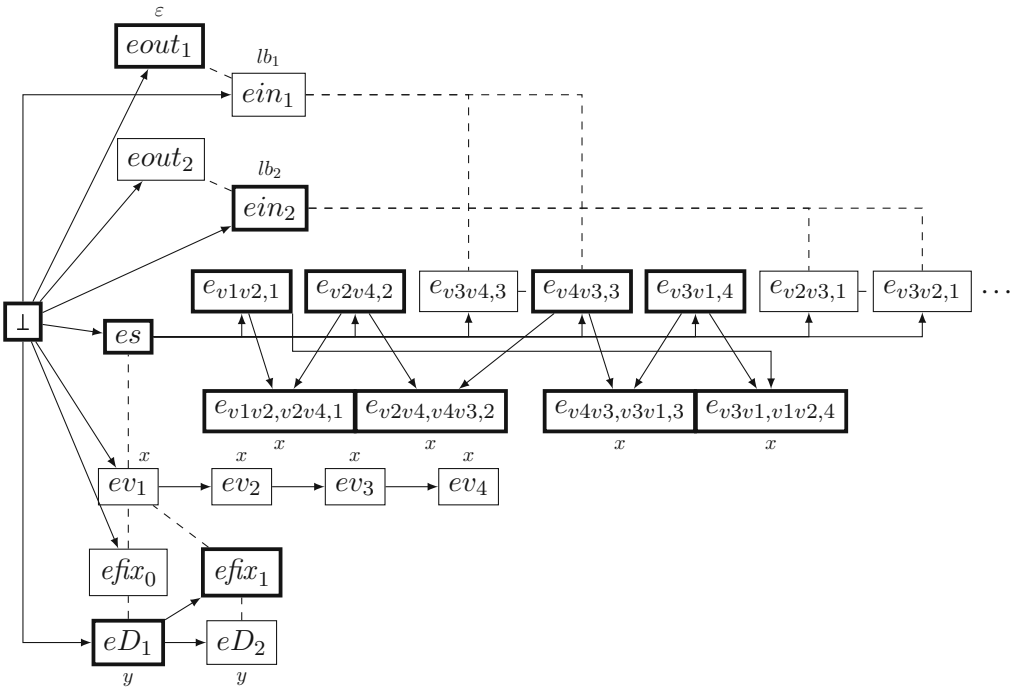


Figure 7.7: An example event structure $\mathcal{E}_1$.

finite automata (NFAs), which allows us to compare our algorithm to NFA language inclusion. In the following presentation of the algorithms, we assume that the causality relations and labeling functions of configurations C_i for $i \in \{1, 2\}$ are implicitly given by its event structure interpretation over $\mathcal{E}_i$. We start by introducing necessary concepts for our decision algorithm.

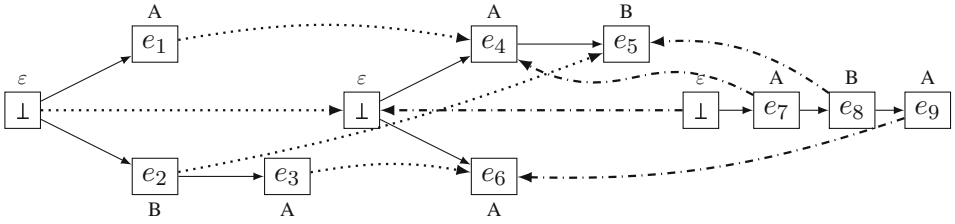
ε-free configurations

For every configuration C, there is a configuration with the same language whose only ε-labeled event is $\perp$. This ε-free configuration can be obtained simply by removing all ε-labeled events besides $\perp$ from its corresponding event structure, in particular from C and $<_{\lfloor C \times C}$. The resulting ε-free configuration has the same language as the initial configuration, because the causality relation is transitive. Furthermore, ε-labeled events do not modify the words and thus removing them does not influence the language of the configuration. Therefore, in order to improve readability, from hereon we assume without loss of generality that configurations are ε-free. We keep the $\perp$ event to improve readability, even though for our purpose this event is not required neither. Note that ε-labeled events are needed during the construction phase of the event structure representing to represent internal transitions.



For space reasons only selected events of the form $e_{f,i}$ and $e_{f,f',i}$ are drawn. For clarity we name the edges $\vec{f}$ and $\overleftarrow{f}$ by the vertices they connect, e.g. $v1v2$ and $v2v1$.

Figure 7.8: An example event structure $\mathcal{E}_2$.



(a) Structure 1 (b) Structure 2 (c) Structure 3

Figure 7.9: Necessary and sufficient embeddings.
$\varphi : \bot \mapsto \bot; e_1 \mapsto e_4; e_2 \mapsto e_5; e_3 \mapsto e_6$ is a necessary embedding (dotted arrows).
$\varphi : \bot \mapsto \bot; e_7 \mapsto e_4; e_8 \mapsto e_5; e_9 \mapsto e_6$ is a sufficient embedding (dash-dotted arrows).

Embeddings

An embedding is a structure-preserving one-to-one mapping between events of two configurations from different event structures. We consider two different types of embeddings that vary in their strictness in terms of structure preservation. Since embeddings are defined between configurations, conflicts do not play a role in these considerations. In order to use these embeddings

for deciding language inclusion between two FLES, we assume that in a step prior to searching for embeddings, the maximal configurations of both $\mathcal{E}_1$ and $\mathcal{E}_2$ are computed. During the construction of event structures from action systems, described in Section 7.2.2, we immediately obtain a maximal configuration once it can not be extended anymore. For given event structures, the maximal configurations can be computed with the unfolding-based partial order reduction algorithm given in [RSSK15].

In the following we consider two configurations C_1 and C_2 of two $\mathcal{X}$-labeled FLES $\mathcal{E}_1 = \langle E_1, <_1, \#_1, h_1\rangle$ respectively $\mathcal{E}_2 = \langle E_2, <_2, \#_2, h_2\rangle$.

Definition 7.5.1 (Necessary Embedding)**.** A mapping $\varphi : C_1 \to C_2$ is a *necessary embedding* if A) φ is bijective, B) $\forall e \in C_1 : h_1(e) = h_2(\varphi(e))$, and C) $\forall e \in C_1 : \neg\big(e(<_1 \cup <_2^{\varphi})^+ e\big)$, where $.^+$ denotes transitive closure and $<_2^{\varphi}$ denotes the relation $<_2$ mapped to the events of C_1. Formally $<_2^{\varphi} \stackrel{\text{def}}{=} \{(\varphi^{-1}(e_1), \varphi^{-1}(e_2)) \mid \exists e_1, e_2 \in C_2.\ e_1 <_2 e_2\}$. For a necessary embedding φ from C_1 to C_2, we write $C_1 \sim_N^{\varphi} C_2$. We write $C_1 \sim_N C_2$ if there exists a necessary embedding φ such that $C_1 \sim_N^{\varphi} C_2$.

A necessary embedding implies that the two configurations have a common word, by requiring they have the same number of events with the same labels and that their partial orders are not contradicting each other. Note that the relation $\sim_N$ is symmetric, since for a necessary embedding $\varphi : C_1 \to C_2$, φ^{-1} is a necessary embedding from C_2 onto C_1

Example 7.5.1. Consider the configurations in Figures 7.9a and 7.9b. There are only two label-preserving bijections between the configurations: $\varphi_1 : \bot \mapsto \bot; e_1 \mapsto e_6; e_2 \mapsto e_5; e_3 \mapsto e_4$ and $\varphi_2 : \bot \mapsto \bot; e_1 \mapsto e_4; e_2 \mapsto e_5; e_3 \mapsto e_6$.

The mapping φ_1 is not a necessary embedding, since $e_2(<_1 \cup <_2^{\varphi})e_2$, which violates C). To see this, consider the chain of events $e_2 <_1 e_3 <_2^{\varphi} e_2$, where $e_3 = \varphi^{-1}(e_4), e_2 = \varphi^{-1}(e_5)$, and $e_4 <_2 e_5$. In contrast, φ_2 is a necessary embedding and a witness to the common word ABA of both configurations.

Lemma 7.5.1. Let C_1 and C_2 be maximal configuration of FLES $\mathcal{E}_1$ respectively $\mathcal{E}_2$.
$C_1 \sim_N C_2$ if and only if $\mathcal{L}(C_1) \cap \mathcal{L}(C_2) \neq \varnothing$.

Proof. $\Rightarrow$: There is a label preserving bijection φ, such that $<_1 \cup <_2^{\varphi}$ is a partial order. Thus, via a depth first search over the events of C_1 with the order $<_1 \cup <_2^{\varphi}$, we can find a sequence of events $\langle e_1, \ldots, e_n\rangle$, such that $e_j(<_1 \cup <_2^{\varphi})e_i$ implies $j < i$. It follows that $e_j <_1 e_i$ implies $j < i$, i.e. $\langle e_1, \ldots, e_n\rangle \in T(C_1)$. Likewise it is the case that $e_j(<_2^{\varphi})e_i$ implies $j < i$ and by the definition of $<^{\varphi}$, we get $\varphi(e_j) <_2 \varphi(e_i)$, i.e. $\varphi(\langle e_1, \ldots, e_n\rangle) \in T(C_2)$. Since φ is label-preserving, $w \stackrel{\text{def}}{=} h(\langle e_1, \ldots, e_n\rangle) = h(\varphi(\langle e_1, \ldots, e_n\rangle))$, which implies $w \in \mathcal{L}(C_1)$ and $w \in \mathcal{L}(C_2)$.

$\Leftarrow$: Let $t_1 = \langle e_1, \ldots, e_n\rangle$ and $t_2 = \langle e'_1, \ldots, e'_n\rangle$ be traces of C_1 and C_2 respectively, such that $h(t_1) = h(t_2) \in \mathcal{L}(C_1) \cap \mathcal{L}(C_2)$. We show that the mapping $\varphi : e_i \mapsto e'_i$ is a necessary embedding. Clearly φ is bijective and label-preserving. We need to show $\forall e \in C_1 : \neg e(<_1 \cup <_2^{\varphi})e$. Assume the contrary, i.e. $\exists e \in C_1 : e(<_1 \cup <_2^{\varphi})e$. In order to close the cycle and since $<_1$ and $<_2$ are order

relations, there must exist an event $e' \in C_1$, such that $e' <_1 e$ and $e <_2^{\varphi} e'$, i.e. $\varphi(e) <_2 \varphi(e')$. This implies, that for all $t \in T(C_1)$ we have that e' appears earlier than e and all traces $t \in T(C_2)$ are such that $\varphi(e)$ appears earlier than $\varphi(e')$. This is a contradiction to $t_1 \in T(C_1), t_2 \in T(C_2)$ and $t_2 = \varphi(t_1)$. □

The following corollary gives rise to a termination criterion of the decision algorithm. If we find a configuration C in $\mathcal{E}_1$, such that there exists no configuration in $\mathcal{E}_2$ that shares a word with C, we can abort the search and report non-inclusion.

Corollary 7.5.1. Let $C, C_1, \ldots, C_n$ be configurations such that $C \neq \emptyset$. If $(\forall i = 1, \ldots, n : C_i \not\sim_N C)$ then $\mathcal{L}(C) \notin \bigcup_{i=1}^{n} \mathcal{L}(C_i)$.

The second type of embedding has a stronger requirement on structure preservation. Intuitively, it requires that the source of such an embedding is at least as strict in terms of causality as the target.

Definition 7.5.2 (Sufficient Embedding)**.** A mapping $\varphi : C_1 \to C_2$ is a *sufficient embedding* if A) φ is bijective, B) $\forall e \in C_1 : h_1(e) = h_2(\varphi(e))$, and C) $\forall e_1, e_2 \in C_1 : \varphi(e_1) <_2 \varphi(e_2) \implies e_1 <_1 e_2$. If there exists a sufficient embedding φ from C_1 to C_2, we write $C_1 \sqsubset_S^{\varphi} C_2$. We write $C_1 \sqsubset_S C_2$ if there exists sufficient embedding φ, such that $C_1 \sqsubset_S^{\varphi} C_2$.

A sufficient embedding is a witness to language inclusion between configurations. The reason to work with two kinds of embeddings is that we can construct necessary embeddings using a backtracking algorithm. It is easy to check whether a necessary embedding is also sufficient, whereas it is not straight forward to construct a sufficient embedding from scratch.

Example 7.5.2. Consider the configurations in Figures 7.9b and 7.9c. The mapping $\varphi_1 : \bot \mapsto \bot; e_7 \mapsto e_4; e_8 \mapsto e_5; e_9 \mapsto e_6$ is a sufficient embedding. The only non-trivial causality to check is $e_4 <_2 e_5$, for which we have $\varphi_1^{-1}(e_4) = e_7 <_3 e_8 = \varphi_1^{-1}(e_5)$. In contrast, $\varphi_2 : \bot \mapsto \bot; e_7 \mapsto e_6; e_8 \mapsto e_5; e_9 \mapsto e_4$ is not a sufficient embedding, since in this case $e_4 <_2 e_5$ and $\varphi_1^{-1}(e_4) = e_9 \not<_3 e_8 = \varphi_1^{-1}(e_5)$. This shows that the language of the event structure in Figure 7.9c is included in language of the event structure in Figure 7.9b.

The following Lemma provides a connection between sufficient embeddings and language inclusion. In case there exists a sufficient embedding, the respective languages are included.

Lemma 7.5.2. Let C_1 and C_2 be maximal configurations of FLES $\mathcal{E}_1$ and $\mathcal{E}_2$ respectively. If $C_1 \sqsubset_S C_2$ then $\mathcal{L}(C_1) \subseteq \mathcal{L}(C_2)$.

Proof. We show the claim by induction on $|C_1| = |C_2| = n$.

In the base case we have $C_1 = \{\bot\}$ and $C_2 = \{\bot\}$. For these configurations the claim is trivially fulfilled.

Let the induction hypothesis be that the claim holds for all configurations of size n.

Assume that C_1 and C_2 are configurations of size $n+1$, such that $C_1 \sqsubset_S^{\varphi} C_2$. We show $T(C_1) \subseteq \varphi(T(C_2))$. Since φ is bijective and label-preserving, $T(C_1) \subseteq \varphi(T(C_2))$ is equivalent to $\mathcal{L}(C_1) \subseteq \mathcal{L}(C_2)$.

Let $t = \langle e_1, \ldots, e_{n+1} \rangle \in T(C_1)$. We need to show that $\varphi(t) \in T(C_2)$.

Since e_{n+1} is the last event of a trace and we consider only traces that include all events of the configuration, clearly e_{n+1} is maximal in C_1. According to Lemmas 7.5.6 and 7.5.7, we have that $C_1 \setminus \{e_{n+1}\}$ and $C_2 \setminus \{\varphi(e_{n+1})\}$ are configurations of size n. Furthermore, φ restricted to $C_1 \setminus \{e_{n+1}\}$ is a witness for $C_1 \setminus \{e_{n+1}\} \sqsubset_S C_2 \setminus \{\varphi(e_{n+1})\}$. From the induction hypothesis we get $\varphi(\langle e_1, \ldots, e_n \rangle) \in T(C_2 \setminus \{\varphi(e_{n+1})\})$. Since $\varphi(e_{n+1})$ is maximal in C_2, from Lemma 7.5.7 follows $\varphi(\langle e_1, \ldots, e_n, e_{n+1} \rangle) \in T(C_2)$. □

The converse statement is not always true. To see this, consider a configuration $C_1 = \{\bot, e_1, e_1'\}$ such that e_1 and e_1' are concurrent and $h(e_1) = h(e_1') = A$. Furthermore, consider a configuration $C_2 = \{\bot, e_2, e_2'\}$ such that e_2 and e_2' are sequential and $h(e_2) = h(e_2') = A$. Clearly, the configurations have the same language $\{AA\}$. However, there is no sufficient embedding from C_1 to C_2.

Our decision algorithm performs an additional refinement step in such a case and concludes language inclusion only after checking the refined configurations. In Lemma 7.5.3, we show that in the case of unique labels, the converse statement also holds.

Splits

Our language inclusion decision algorithm continuously performs configuration refinement steps that we call splits. To be precise, we refine the causality relation of its corresponding event structure.

Definition 7.5.3 (Split)**.** Let C be a configuration of event structure $\langle E, <, \#, h \rangle$ and let $e_1, e_2 \in C$ be two concurrent events. The *split* of C on e_1 before e_2 is the configuration $C_{e_1<e_2} \stackrel{\text{def}}{=} \langle C, (< \cup \{(e_1, e_2)\})^+_{\lceil C \times C}, \varnothing, h_{\lceil C} \rangle$ where $.^+$ denotes transitive closure.

A split on two concurrent events e_1 and e_2 simply adds an additional ordering constraint between the two events. In our algorithm, we always split both ways, creating two new configurations that order concurrent events e_1 and e_2 one way and the other. Note that in order to avoid duplication of events, in practice splits can be implemented via additional, optional causalities on the event structure. The following lemma states that splitting a configuration in both ways produces two new configurations with languages whose union is the original language.

Lemma 7.5.3. Let C be a configuration and $e_1, e_2 \in C$ be concurrent events, then the languages of the split configurations are such that $\mathcal{L}(C) = \mathcal{L}(C_{e_1<e_2}) \cup \mathcal{L}(C_{e_2<e_1})$. If h is injective (labels are unique), then $\mathcal{L}(C_{e_1<e_2}) \cap \mathcal{L}(C_{e_2<e_1}) = \varnothing$.

Proof. $\mathcal{L}(C) = \mathcal{L}(C_{e_1<e_2}) \cup \mathcal{L}(C_{e_2<e_1})$ is a direct consequence of Lemma 7.5.10 and the definition of the language of prime event structures. If h is injective, i.e. labels are unique, then

each word in $\mathcal{L}(C_{e_1<e_2})$ contains the sub sequence $\ldots h(e_1)\ldots h(e_2)\ldots$ whereas each word in $\mathcal{L}(C_{e_2<e_1})$ contains the different sub sequence $\ldots h(e_2)\ldots h(e_1)\ldots$, which shows that no word can be part of both languages. □

The following lemma guarantees progress of our algorithm. It states that if we find a necessary, but not sufficient embedding, there are events that can be used to split C_1. The goal is that after a finite number of splits a sufficient embedding can be established.

Lemma 7.5.4. Let C_1, C_2 be maximal configurations of FLES $\mathcal{E}_1$ respectively $\mathcal{E}_2$. Furthermore, let $C_1 \sim_N^\varphi C_2$ and $C_1 \not\sim_S^\varphi C_2$. Then there are concurrent events $e, e' \in C_1$ such that $\varphi(e) <_2 \varphi(e')$.

Proof. Since φ is a label-preserving bijection, from $C_1 \not\sim_S^\varphi C_2$ it follows that there are events $e, e' \in C$, such that $\varphi(e) <_2 \varphi(e')$ and $e \not<_1 e'$. Towards contradiction assume $e' <_1 e$. Then we have $e'(<_1 \cup <_2^\varphi)e'$, which is a contradiction to φ being a necessary embedding. Since C_1 and C_2 are conflict-free it follows that e, e' are concurrent and a witness to the claim. □

In the following, we prove properties of labeled event structures and embeddings that we use to show the correctness of our approach.

Lemma 7.5.5. Let C be a configuration. $|T(C)| = 1$ if and only if $|C| = 1$ or for all events $e, e' \in C$ either $(e < e')$ or $(e' < e)$.

Proof. The claim is trivial for $|C| = 1$.

$\Rightarrow$: Let $T(C) = \{\langle e_1, \ldots, e_n\rangle\}$. Since $\langle e_2, e_1, \ldots, e_n\rangle \notin T(C)$, we have that $e_1 < e_2$. Likewise, we can show that $e_i < e_{i+1}$ for all $i \in [1, n-1]$. Therefore, $e_i < e_j$ for $i < j$. Since $C = \{e_1, \ldots, e_n\}$ have shown the claim.

$\Leftarrow$: Since $<$ is irreflexive, we immediately get that $|T(C)| > 0$. From the definition of $T(C)$ follows that $e < e'$ then e appears earlier in any trace in $T(C)$ than e'. Since we assume that every pair of events is ordered by $<$, we have that the order in any trace is fully fixed. In other words, there can only be a single trace. □

Corollary 7.5.2. $|T(C)| > 1$ if and only if there are concurrent events $e, e' \in C$.

Lemma 7.5.6. Let C_1 and C_2 be configurations such that $C_1 \sqsubset_S^\varphi C_2$. If e is maximal in C_1 then $\varphi(e)$ is maximal in C_2.

Proof. Assume there exists e', such that $\varphi(e) <_2 \varphi(e')$. Since φ is a sufficient embedding, we have that $e <_1 e'$, which is a contradiction to e being maximal in C_1. Therefore, no such e' exists. Since φ is bijective, $\varphi(e)$ is maximal in C_2. □

Lemma 7.5.7. Let e be maximal in configuration C, then $C \setminus \{e\}$ is a configuration and for every $\langle e_1, \ldots, e_n\rangle \in T(C \setminus \{e\})$ it is the case that $\langle e_1, \ldots, e_n, e\rangle \in T(C)$.

Proof. $C \setminus \{e\}$ is conflict free, because C is conflict free. $C \setminus \{e\}$ is left closed, because e is maximal, therefore there is no event e' such that $e < e'$. Note that traces in $T(.)$ contain all events of the configuration. Therefore, from $\langle e_1, \ldots, e_n \rangle \in T(C \setminus \{e\})$ follows $\{e_1, \ldots, e_n, e\} = C$. Furthermore, for every $e' \in C$, such that $e' < e$, we have that $e' \in \{e_1, \ldots, e_n\}$, which implies $\langle e_1, \ldots, e_n, e \rangle \in T(C)$. □

Lemma 7.5.8. Let C be a configuration. $\langle e_1, \ldots, e_n \rangle \in T(C)$ implies that e_n is maximal in C.

Proof. Assume e_n is not maximal, i.e. there exists $j \in [1, n-1]$ such that $e_n < e_j$, which is a contradiction to $\langle e_1, \ldots, e_n \rangle \in T(C)$.

□

Lemma 7.5.9. Let C_1 and C_2 be maximal configuration of FLES $\mathcal{E}_1$ and $\mathcal{E}_2$ and for every $e_1 \in C_1$ and $e_2 \in C_2$ it is the case that $|\{e \in C_1 \mid h(e_1) = h(e)\}| = 1$ and $|\{e \in C_2 \mid h(e_2) = h(e)\}| = 1$ If $\mathcal{L}(C_1) \subseteq \mathcal{L}(C_2)$ then $C_1 \sqsubset_S C_2$.

Proof. We show the claim by induction on $|C_1| = |C_2| = n$.

In the base case we have $C_1 = \{\bot\}$ and $C_2 = \{\bot\}$. For these configurations the claim is trivially fulfilled.

Let the induction hypothesis be that the claim holds for all configurations of size n.

Let e_{n+1} be a maximal event of C_1. Let $M_{e_{n+1}} \stackrel{\text{def}}{=} \{e \in C_2 \mid e \text{ is maximal and } h(e_{n+1}) = h(e)\}$. We start by showing $\exists e \in M_{e_{n+1}}$, such that $\mathcal{L}(C_1 \setminus \{e_{n+1}\}) \subseteq \mathcal{L}(C_2 \setminus \{e\})$.

Assume the contrary. That is, for every maximal event $e \in M_{e_{n+1}}$ it is the case that $\mathcal{L}(C_1 \setminus \{e_{n+1}\}) \nsubseteq \mathcal{L}(C_2 \setminus \{e\})$. Then $\mathcal{L}(C_1 \setminus \{e_{n+1}\}) \nsubseteq \bigcup_{e \in M_{e_{n+1}}} \mathcal{L}(C_2 \setminus \{e\})$. That is, there is a word $w \in \mathcal{L}(C_1 \setminus \{e_{n+1}\})$ such that $w \notin \bigcup_{e \in M_{e_{n+1}}} \mathcal{L}(C_2 \setminus \{e\})$. Since e_{n+1} is maximal $w \circ h(e_{n+1}) \in \mathcal{L}(C_1)$ ($\circ$ denotes concatenation). Let us define language $\mathcal{L}^\circ \stackrel{\text{def}}{=} \bigcup_{e \in M_{e_{n+1}}} \mathcal{L}(C_2 \setminus \{e\}) \circ h(e_{n+1})$ ($\circ$ is applied element-wise). Clearly, $w \circ h(e_{n+1}) \notin \mathcal{L}^\circ$.

We claim that no word in $\mathcal{L}(C_2) \setminus \mathcal{L}^\circ$ ends with $h(e_{n+1})$. Assume there is such a word w'. This word corresponds to a trace ending with some event e' with label $h(e_{n+1})$. Since e' is the last event in a trace, and we only consider maximal configurations, it needs to be maximal. Therefore, $e' \in M_{e_{n+1}}$, which is a contradiction to $w' \in \mathcal{L}(C_2) \setminus \mathcal{L}^\circ$. Since $w \circ h(e_{n+1})$ ends with $h(e_{n+1})$, we have $w \circ h(e_{n+1}) \notin \mathcal{L}(C_2) \setminus \mathcal{L}^\circ$.

However, clearly $\mathcal{L}(C_2) = \mathcal{L}^\circ \cup (\mathcal{L}(C_2) \setminus \mathcal{L}^\circ)$. In summary, we found $w \circ h(e_{n+1}) \in \mathcal{L}(C_1)$ such that $w \circ h(e_{n+1}) \notin \mathcal{L}(C_2)$, which is a contradiction to $\mathcal{L}(C_1) \subseteq \mathcal{L}(C_2)$.

Therefore, we can apply the induction hypothesis to $C_1 \setminus \{e_{n+1}\}$ and $C_2 \setminus \{e\}$ for the existing $e \in M_{e_{n+1}}$. That is, $C_1 \setminus \{e_{n+1}\} \sqsubset_S^\varphi C_2 \setminus \{e\}$. We extend φ to C_1 by setting $\varphi(e_{n+1}) \stackrel{\text{def}}{=} e$. Clearly, φ is bijective and label-preserving. Furthermore, causality preservation for events other than e_{n+1} carries over from $C_1 \setminus \{e_{n+1}\}$ to C_1. What remains to show is that for all $e \in C$ with $\varphi(e) < \varphi(e_{n+1})$ we have $e < e_{n+1}$. Since e_{n+1} is maximal, clearly it is not the case that

$e_{n+1} < e$. Assume that e and e_{n+1} are concurrent. Then there is a word, in which $h(e_{n+1})$ appears earlier than $h(e)$. Since events are uniquely labeled and $\varphi(e) < \varphi(e_{n+1})$, this word is not in $\mathcal{L}(\mathcal{E}_2)$, which is a contradiction to $\mathcal{L}(\mathcal{E}_1) \subseteq \mathcal{L}(\mathcal{E}_2)$. Therefore, $e < e_{n+1}$ and φ is a sufficient embedding. □

Lemma 7.5.10. Let C be a configuration and $e_1, e_2 \in C$ be concurrent events, then $T(C) = T(\mathcal{E}^C_{e_1<e_2}) \cup T(\mathcal{E}^C_{e_2<e_1})$ and $T(\mathcal{E}^C_{e_1<e_2}) \cap T(\mathcal{E}^C_{e_2<e_1}) = \emptyset$.

Proof. From $t \in T(C_{e_1<e_2})$ directly follows $t \in T(C)$, since there is simply one less constraint to fulfill. Thus, we have $T(C_{e_1<e_2}) \subseteq T(C)$. Symmetrically we can show $T(C_{e_2<e_1}) \subseteq T(C)$, thus showing $T(C_{e_1<e_2}) \cup T(C_{e_2<e_1}) \subseteq T(C)$.

Let $t = \langle e'_1, \ldots, e'_n \rangle \in T(C)$. There are $i \neq j$, such that $e'_i = e_1$ and $e'_j = e_2$. In the case $i < j$, t fulfills all constraints to be a trace of $C_{e_1<e_2}$, thus $t \in T(C_{e_1<e_2})$. Likewise $j < i$ implies $t \in T(C_{e_2<e_1})$. In any case $t \in T(C_{e_1<e_2}) \cup T(C_{e_2<e_1})$, showing $T(C) \subseteq T(C_{e_1<e_2}) \cup T(C_{e_2<e_1})$.

Further, the $T(\mathcal{E}^C_{e_1<e_2})$ and $T(\mathcal{E}^C_{e_2<e_1})$ are disjoint because no trace can fulfill both constraints $e_1 < e_2$ and $e_2 < e_1$. □

7.5.1 Language Inclusion Decision Algorithm

Algorithm 4: The language inclusion decision algorithm.

Input: Finite, labeled Prime Event Structures $\mathcal{E}_1$ and $\mathcal{E}_2$
Result: $\mathcal{L}(\mathcal{E}_1) \subseteq \mathcal{L}(\mathcal{E}_2)$

1 $\{C^1_1, \ldots, C^n_1\}, \{C^1_2, \ldots, C^m_2\} \leftarrow$ all maximal configurations of $\mathcal{E}_1$ respectively $\mathcal{E}_2$
2 **return** $\bigwedge_{C_1 \in \{C^1_1, \ldots, C^n_1\}}$ `Check` $(C_1, \{C^1_2, \ldots, C^m_2\})$
3 **Function** `Check` $(C_1, \{C^{i_1}_2, \ldots, C^{i_l}_2\})$ **:**
  Result: $\mathcal{L}(C_1) \subseteq \bigcup_{j=1}^{l} \mathcal{L}(C^{i_j}_2)$
4 $Candidates \leftarrow \{C^{i_1}_2, \ldots, C^{i_l}_2\}$
5 **foreach** $C_2 \in \{C^{i_1}_2, \ldots, C^{i_l}_2\}$ **do**
6 **if** $\exists \varphi : C_1 \to C_2.\ C_1 \sim^{\varphi}_N C_2$ **then**
7 **return** `SufficientOrSplit` $(C_1, C_2, \varphi, Candidates)$
8 **else**
9 $Candidates \leftarrow Candidates \setminus \{C_2\}$
10 **return** false ▷ `counter-example` C_1
11 **Function** `SufficientOrSplit` $(C_1, C_2, \varphi, Candidates)$ **:**
12 **if** $C_1 \sqsubset^{\varphi}_S C_2$ **then**
13 **return** true
14 Let $e, e' \in C_1$ be concurrent and $\varphi(e) <_2 \varphi(e')$ ▷ `exists (Lemma 7.5.4)`
15 **return** `SufficientOrSplit` $(C_{e<e'}, C_2, \varphi, Candidates) \wedge$
  `Check` $(C_{e'<e}, Candidates)$

We present our decision algorithm in Algorithm 4. Inputs to the algorithm are finite, labeled prime event structures $\mathcal{E}_1$ and $\mathcal{E}_2$.

The first step of the algorithm is to calculate the maximal configurations of the event structure, which can be done with the algorithm described in [RSSK15]. For every maximal configuration C_1 of $\mathcal{E}_1$, the function `Check` attempts to show that $\mathcal{L}(C_1) \subset \mathcal{L}(\mathcal{E}_2)$ by searching for sufficient embeddings from (refined versions of) C_1 to maximal configurations of $\mathcal{E}_2$.

In order to construct candidate sufficient embeddings, in Line 6 the algorithm attempts to construct necessary embeddings, using Algorithm 5. Thereafter, function `SufficientOrSplit` checks whether a necessary embedding φ is also sufficient. This can be done by checking $\forall e \in C_2 : \forall e' \in \mathsf{dsucc}(e) : \varphi^{-1}(e)$ is not concurrent with $\varphi^{-1}(e')$. In case φ is not a sufficient embedding, such a pair of events is guaranteed to exist by Lemma 7.5.4. For efficiency, this check can already be done during construction of the necessary embedding.

In case φ is not sufficient, Lemma 7.5.4 guarantees the existence of a pair of concurrent events that can be split. The resulting split configurations are recursively checked for language inclusion in Line 15. Lemma 7.5.4 guarantees us that φ is a necessary embedding for one of the splits (say $C_{e<e'}$). Therefore, for $C_{e<e'}$ we do not need to construct a new necessary embedding again, but can immediately check whether φ is a sufficient embedding for $C_{e<e'}$.

In case no necessary embedding can be found for some configuration C_1 and its candidates, according to Lemma 7.5.1, we can conclude $\mathcal{L}(C_1) \cap \mathcal{L}(\mathcal{E}) = \emptyset$, i.e. all words in C_1 are counter-examples to language inclusion. Once Line 10 is reached, we know that C_1 does not share any word with any $\{C_2^1, \ldots, C_2^m\}$. Therefore, C_1 is a counter-example to language inclusion.

The algorithm terminates, because the notions of necessary and sufficient embedding collapse in case the configuration contains only a single trace, which is the case when the causality relation is a total order on the events of the configuration (see Lemma 7.5.5).

As the algorithm recursively searches for sufficient embeddings, for efficiency, we can reduce the set of candidate configurations, because in case there is no necessary embedding between two configurations, there is clearly also no necessary embedding between any of their split configurations.

We present the algorithm to construct necessary embeddings in Algorithm 5. Intuitively, the algorithm is a combined depth first search over the causality relation, as well as the space of possible bijective, label-preserving mappings.

The algorithm starts by dismissing configurations that can never have a necessary embedding because the number of events with the same label differs (Line 1). The actual embedding is established with the recursive function `NecessaryEmbedding`. The recursion maintains a frontier of events that are yet to be explored and a partial mapping of already explored events. It ends if the frontier becomes empty (Line 5). The exploration is done on C_1 by adding the successors of the current event e to the frontier (Line 8). Then for every event e'' in C_2 with the same label as e a mapping φ' is created and a cycle check is performed. If this mapping does not introduce a cycle we recurs on it (Line 13). The first valid (not `None`) mapping that is constructed during the recursion is returned. If no such mapping is found, then `None` is returned. The cycle check in Line 12 basically checks if the two causality relations $<_1$ and $<_2$ are compatible for the mapped events, in the sense that the events can be brought in an order

Algorithm 5: The $\sim_N$ decision algorithm.

```
   Input: Configurations C_1, C_2
   Result: φ : C_1 → C_2 if C_1 ~_N^φ C_2, None otherwise
1  if ∃x ∈ 𝒳. |{e ∈ C_1 | h_1(e) = x}| ≠ |{e ∈ C_2 | h_2(e) = x}| then
2      return false
3  return NecessaryEmbedding({⊥_1}, {⊥_1 ↦ ⊥_2})
4  NecessaryEmbedding(frontier, φ):
       Input: frontier stack of events C_1
       Input: φ : C_1 → C_2 (partial mapping)
       Result: φ : E_1 → E_2 if C_1 ~_N^φ C_2, None otherwise
5      if frontier = ∅ then
6          return φ
7      e ← frontier.pop()
8      foreach e' ∈ dsucc_{ℰC_1}(e) do          ▷ direct successors of e in C_1
9          frontier.push(e')
10     foreach e'' ∈ C_2 such that h_1(e) = h_2(e'') ∧ e'' ∉ range(φ) do
11         φ' ← φ ∪ {e ↦ e''}
12         if ∄e_1, e_2 ∈ E_1. e_1 <_1 e_2 ∧ φ'(e_2) <_2 φ'(e_1) then      ▷ check for cycle
13             φ'' ← NecessaryEmbedding(frontier, φ')
14             if φ'' ≠ None then
15                 return φ''
16     return None
```

that respects both causality relations. The procedure can be implemented using any of the well known cycle detection algorithms over the graph with nodes being events of C_1 and edges being causalities $<_1 \cup <_2^{\varphi}$.

The worst-case runtime of the decision algorithm is exponential in $O(2^{2n})$, where $n = |E_1|+|E_2|$, which is not surprising for an algorithm solving a Π_2^p hard problem. There are two dominant factors of the exponential complexity.

First, the number of maximal configurations can be exponential in n and Algorithm 4 potentially has to compare all pairs of maximal configurations. The CCNFS benchmark in Section 7.6 is an example for an event structure with an exponential number of maximal configurations in n.

Second, the number of mappings between configurations that need to be considered as candidates for necessary embeddings can be exponential in n. That is, Algorithm 5 has worst case runtime exponential in n. Note that for a fixed mapping, the algorithm performs a linear search over the configuration and the combined causality relation. Furthermore, note that the number of possible embeddings decreases with the number of calls to `Check` and the size of maximal configurations decreases relative to n with the number of maximal configurations. Therefore, the amortized runtime should be much better than the worst case complexity.

7.5.2 Automaton Based Language Inclusion

We provide a language preserving encoding of event structures into non-deterministic finite automata (NFA). The encoding allows us to compare our algorithm to well researched language inclusion algorithms in our evaluation (Section 7.6).

The encoding has a state for every configuration of the event structure. There is a transition between two states, if the difference between the corresponding configurations is just one event. The transition is labeled with the label of that event. In essence, the encoding is an automaton representation of what is known as the configuration structure of a prime event structure [vGP09].

Definition 7.5.4 (Automaton Encoding)**.** Let $\mathcal{E} = \langle E, <, \#, h\rangle$ be a finite prime event structure with labels $\mathcal{X}$. We define the non-deterministic finite automaton $\mathcal{N}^{\mathcal{E}} = \langle Q^{\mathcal{E}}, \Omega^{\mathcal{E}}, \delta^{\mathcal{E}}, q_0^{\mathcal{E}}, F^{\mathcal{E}}\rangle$ as $Q^{\mathcal{E}} = \{q_C^{\mathcal{E}} \mid C \text{ is a configuration of } \mathcal{E}\}$, $\Omega^{\mathcal{E}} = \mathcal{X}$, $(q_{C_1}^{\mathcal{E}}, \sigma, q_{C_2}^{\mathcal{E}}) \in \delta^{\mathcal{E}}$ iff there is $e \in E$, such that $C_1 \cup \{e\} = C_2$ and $h(e) = \sigma$, $q_0^{\mathcal{E}} \stackrel{\text{def}}{=} q_{\{\bot\}}^{\mathcal{E}}$, and $F^{\mathcal{E}} = \{q_C^{\mathcal{E}} \mid C \text{ is maximal}\}$.

Lemma 7.5.11. Let $\mathcal{E}$ be a labeled, finite prime event structure, then $\mathcal{L}(\mathcal{E}) = \mathcal{L}(\mathcal{N}^{\mathcal{E}})$.

Proof. Given a trace $\vec{e} = \langle e_1, \ldots, e_n\rangle$ we define $C_l^{\vec{e}} \stackrel{\text{def}}{=} \cup_{i=1}^{l}\{e_i\}$.

Let $w \in \mathcal{L}(\mathcal{N}^{\mathcal{E}})$, then by the definition of $\mathcal{N}^{\mathcal{E}}$, there is a trace $\vec{e} = \langle e_1, \ldots, e_n\rangle$ such that $w = \langle h(e_1), \ldots, h(e_n)\rangle$, such that for every $l \leq n : C_l^{\vec{e}}$ is a configuration and $C_n^{\vec{e}}$ is a maximal configuration. Since all $C_l^{\vec{e}}$ are configurations, we have $\langle e_1, \ldots, e_n\rangle \in T(C_n^{\vec{e}})$. Since $C_n^{\vec{e}}$ is maximal, we have $w \in \mathcal{L}(\mathcal{E})$.

Conversely, for every $w \in \mathcal{L}(\mathcal{E})$, there is a trace $\vec{e} = \langle e_1, \ldots, e_n\rangle$ such that $w = \langle h(e_1), \ldots, h(e_n)\rangle$ and $\langle e_1, \ldots, e_n\rangle \in T(\mathcal{E})$. From $\vec{e} \in T(\mathcal{E})$ follows that for every $l \leq n : C_l^{\vec{e}}$ is a configuration and that $C_n^{\vec{e}}$ is maximal. By the definition of $\delta^{\mathcal{E}}$, we have for every $i = 1, \ldots, n-1 : (q_{C_i^{\vec{e}}}^{\mathcal{E}}, h(e_i), q_{C_{i+1}^{\vec{e}}}^{\mathcal{E}}) \in \delta^{\mathcal{E}}$. Since $C_n^{\vec{e}}$ is maximal, $q_{C_n^{\vec{e}}}^{\mathcal{E}}$ is accepting. Therefore, $w \in \mathcal{L}(Q^{\mathcal{E}})$. □

The provided encoding is not optimal in general due to conflicts and the fact that events of prime event structures are caused in a unique way, which is a well known caveat of prime event structures [Win88]. However, for the family of event structures that consists of the $\bot$ event and n concurrent events (c.f. the proof of Theorem 7.5.1), our encoding contains exactly $2^n + 1$ states, which is one state more than the provably optimal NFA accepting the language of the event structure. Furthermore, in our experiments, we apply optimized NFA reduction techniques [MC13] before checking language inclusion on automata.

Theorem 7.5.1. There is a family of event structures $\mathcal{E}_n$ with events E_n, such that $|E_n| = n + 1$, $|\mathcal{L}(\mathcal{E}_n)| = n!$, and $|Q^{\mathcal{E}}| = 2^n + 1$. Every NFA $\mathcal{N}$ with $\mathcal{L}(\mathcal{N}) = \mathcal{L}(\mathcal{E}_n)$ has at least 2^n states.

Proof. The family is given by the $\mathcal{X}$-labeled prime event structures $\mathcal{E}_n \stackrel{\text{def}}{=} \langle E_n, <_n, \varnothing, h_n\rangle$, where $\mathcal{X} \stackrel{\text{def}}{=} \{\varepsilon\} \cup \{1, \ldots, n\}$, $E_n \stackrel{\text{def}}{=} \{\bot\} \cup \{e_1, \ldots, e_n\}$, $<_n \stackrel{\text{def}}{=} \{(\bot, e_i) \mid i = 1, \ldots, n\}$, and $h_n(e_i) \stackrel{\text{def}}{=} i$. $\mathcal{E}_n$ encodes exactly all permutations of $\{1, \ldots, n\}$. As shown in [EKSW05], no NFA with less

than 2^n states can accept the language of permutations of n symbols, showing that every $\mathcal{N}$ with $\mathcal{L}(\mathcal{N}) = \mathcal{L}(\mathcal{E}_n)$ has at least 2^n states.

Furthermore, the number of all permutations of n symbols is $n!$, showing that $|\mathcal{L}(\mathcal{E}_n)| = n!$.

Finally, every non-empty subset of E_n is a configuration. There are $2^n + 1$ such sets, which are all subsets of $\{e_1, \ldots, e_n\}$ together with $\perp$ plus the set $\{\perp\}$. Since states of our encoding correspond one-to-one with configurations, we have $|Q^{\mathcal{E}}| = 2^n + 1$. □

7.6 Experiments

We run two different experiments to evaluate event structure-based test case generation. Firstly, we evaluate model exploration with unfolding-based partial order reduction by comparing it to sequential successor computation. Secondly, we evaluate the language inclusion algorithm by comparing it to automaton based language inclusion. Both experiments were performed on a server that has an Intel(R) Xeon(R) CPU at 3.47GHz, 24 cores, and 189GB RAM. We start the discussion of the experiments with a presentation of the used benchmarks.

7.6.1 Benchmarks

We use the following benchmarks for our experimental evaluation. The benchmark set demonstrates the strong as well as the weak points of unfolding-based partial order reduction as well as event structure-based language inclusion. Among the benchmark models used in the branching search experiments, presented in Section 6.3, only MMS and LBT have significant concurrency and are thus also used within this evaluation. All other benchmark models, scripts to instantiate the models for any parameter value, and the version of MoMuT used in the experiments can be found in [FTW19], which can be run with the virtual machine provided in [DJ19].

- The **Paxos**$_{n,m,k}$ benchmark models the Paxos distributed consensus protocol [L$^+$01] with n proposers, m acceptors, and k learners. The protocol specifies how the different actors can exchange certain messages to achieve consensus on some proposed value. The actions of the actors are largely independent of each other, which introduces lots of concurrency to the model. Furthermore, test cases extracted from our method should be interesting to concertize and run against implementations of the Paxos algorithm.

- The **Semaphore**$_n$ benchmark models n threads that are synchronized by a semaphore. Exactly $n - 1$ threads are allowed to enter and compute in a critical section at the same time. The amount of parallelism of this model is proportional to n. Furthermore, the model exhibits lots of conflicts, as all operations on the semaphore are in conflict with each other.

- The **ParSum**$_n$ benchmark models a parallel summation algorithm. The sequence of integers $0, \ldots, n^2-1$ is split into n equally sized chunks, which are summed up concurrently. Then the partial results are summed up centrally when all parallel threads are finished.

- The **CCNFS**$_n$ benchmark models a system with n events and unique labels, such that the $2i'th$ event is in conflict exactly with the $2i{+}1'th$ event. Every set of independent events induces an event resetting the state. This benchmark is interesting, because its number of maximal configurations $2^{\lfloor \frac{n}{2} \rfloor}$ (each configuration contains either $2i$ or $2i{+}1$ for each $i = 1 \ldots \lfloor \frac{n}{2} \rfloor$) is exponential in the number of events n. Due to the high number of maximal configurations, this benchmark is challenging for our algorithm.

- The **AllPar**$_n$ benchmark models a system with n independent events and unique labels. The benchmark is the ideal case for our algorithm, because its event structure consists of only one maximal configuration with all events in parallel. In contrast, the benchmark is a very bad case for NFA language inclusion, as the smallest NFA to encode all permutations of n symbols is exponential in n (Theorem 7.5.1).

- The **Sharing**$_{n,m}$ benchmark models a system that has n different prefixes that all share the same suffix of length m. The benchmark particularly exhibits the well known shortcoming of prime event structures not being able to encode shared causes. The NFA is able to express the common suffix more succinct in comparison to the event structure.

- MMS and LBT are industrial benchmarks, described in Section 6.3.

7.6.2 Event Structure Construction Experiment

In this experiment, we evaluate the unfolding-based partial order reduction algorithm for action systems, as described in Section 7.2. To this end, we measure the time for construction- and characteristics of- event structures created during test case generation via queries $unfold(0, s)$. Furthermore, we compare the time for construction against the time for sequential successor computation via queries $vispath(0, s)$.

Test case generation was performed with the test generation framework presented in Chapter 6 for 150 steps with a branch length of 15. We limit the time for $vispath(0, s)$ computations by 10 times the time to compute $unfold(0, s)$ or a minimum of 10 seconds. We report the results of the experiment in Table 7.1, where $|\mathcal{E}|$ denotes the size of the constructed event structure in terms of number of events, **PC** denotes a measurement of the degree of concurrency (explained in the following), $unfold$ **time (ms)** denotes the time for constructing $unfold(0, s)$ for some state s as described in Section 7.2 in milliseconds, and $vispath\Delta$ **time** denotes the ratio of time for $vispath(0, s)$ and $vispath(0, s)$ computations (i.e. $\frac{t(vispath(0,s))}{t(unfold(0,s))}$) on computations that did not time out. For every column, we report the average (μ) and maximum (Max) measurement over successor computations for states s reached by the test case generation run. In addition, in the last column, we report the percentage of timed out $vispath(0, s)$ computations.

The concurrency measure **PC** for a single configuration C is defined as $|C|/max_{e \in C} depth(e)$, where $depth(e)$ denotes the longest paths of causalities between e and the initial event $\perp$. Highly concurrent configurations have a broad structure in terms of causality and therefore a high **PC** coefficient, whereas for configurations that order events sequentially (i.e. $|C| = max_{e \in C} depth(e)$) the **PC** coefficient is one. In Table 7.1 **PC** denotes the average coefficient of maximal configu-

Name	$\lvert\mathcal{E}\rvert$		**PC**		*unfold* **time (ms)**		$vispath\Delta$ **time**		
	μ	Max	μ	Max	μ	Max	μ	Max	TO %
$\text{Paxos}_{2,3,1}$	18.6	49	1.8	2.5	5.0	45.0	25.7	81.0	32.7
$\text{Paxos}_{3,6,1}$	71.4	145	4.8	7.7	258.6	18752.0	0.3	1.0	95.3
Semaphore_{3}	3.0	41	1.0	1.5	0.2	4.5	0.0	1.0	0.0
Semaphore_{11}	13.4	16	1.1	1.2	2.3	5.1	0.1	1.0	0.0
ParSum_{3}	19.0	21	2.3	2.3	2.3	3.0	1.4	2.0	0.0
ParSum_{5}	38.0	38	3.8	3.8	21.6	25.0	49.6	61.0	0.0
ParSum_{10}	322.3	550	8.3	8.3	915745.3	1189478.0	-	-	100.0
CCNFS_{3}	15.0	15	1.7	1.7	1.9	2.1	0.2	1.0	0.0
CCNFS_{6}	77.0	77	2.7	2.7	28.7	33.0	0.3	1.0	0.0
CCNFS_{10}	270.0	270	4.0	4.0	344.3	535.0	3.6	5.0	0.0
AllPar_{10}	12.0	12	4.0	4.0	0.7	1.0	12.6	16.0	0.0
AllPar_{50}	52.0	52	17.3	17.3	6.3	10.0	-	-	100.0
AllPar_{500}	502.0	502	167.3	167.3	581.6	632.0	-	-	100.0
$\text{Sharing}_{5,20}$	111.0	111	1.0	1.0	13.2	21.0	0.1	1.0	0.0
$\text{Sharing}_{50,50}$	1014.0	1014	1.0	1.0	565.4	677.0	0.2	1.0	0.0
MMS	44.6	90	1.3	7.0	352.1	738.0	0.2	9.0	4.0
LBT	375.2	631	1.8	40.8	399550.9	467067.0	-	-	100.0

Table 7.1: The experimental results for event structure construction.

rations of $unfold(0, s)$. Thus μ / Max **PC** denote an average over maximal configurations and then an average / maximum over states.

Two key columns of this table are $vispath\Delta$ **time** and **PC**, which allow us to draw conclusions on the strength of unfolding-based partial order reduction in relation to the degree of concurrency. A high value in either column in the $vispath\Delta$ **time** section indicates that unfolding-based partial order reduction pays off on the example, since sequential successor state construction was slower or even timed out. Indeed many rows have a high value in either one of these columns, which indicates that the method works well. Furthermore, for many rows there is a 100% timeout, which indicates that sequential successor state computation simply is not feasible on these models due to an exponential explosion in the number of paths. However, benchmarks **Semaphore**, **CCNFS**, and **Sharing** are partial outliers and show that visible successor state computation can be faster, for which there are three explanations. Firstly, as discussed above, these benchmarks were deliberately chosen as hard cases for unfolding-based partial order reduction. Secondly, they are not highly concurrent, as can be seen on their low **PC** numbers. Thirdly, they are not particularly large, as can be seen on their low $\lvert\mathcal{E}\rvert$ and *unfold* **time (ms)** numbers. Thus, the overhead of analyzing conflicts and structuring the successor state computation using the unfolding-based partial order reduction algorithm given in [RSSK15] outweighs the benefit of leveraging independence on these examples. However, as can be seen on the CCNFS_{10} row, as soon as the model gets larger, this trend reverses and unfolding-based partial order reduction starts paying off again.

LBT and ParSum_{10} are clearly the most computationally demanding benchmarks. Indeed the

Name	Inclusion		Non-Inclusion		Inclusion		Non-Inclusion	
	Time	Num	Time	Num	Time	Num	Time	Num
$\text{Paxos}_{2,3,1}$	-	0	$4{\cdot}10^3$	65	TO	TO	TO	TO
$\text{Paxos}_{3,6,1}$	-	0	$4{\cdot}10^3$	94	TO	TO	TO	TO
Semaphore_3	-	0	23.8	78	-	0	324.0	78
Semaphore_{11}	3.5	2	48.9	84	-	0	564.2	80
ParSum_3	1.1	80	170.5	92	$3{\cdot}10^3$	80	$9{\cdot}10^3$	84
ParSum_5	342.0	84	67.6	94	TO	TO	TO	TO
ParSum_{10}	-	0	12.6	23	TO	TO	TO	TO
CCNFS_3	3.9	88	1.2	88	262.7	88	610.9	88
CCNFS_6	501.8	108	84.0	65	379.7	108	$1{\cdot}10^3$	92
CCNFS_{10}	$303{\cdot}10^3$	98	$17{\cdot}10^3$	59050	TO	TO	TO	TO
AllPar_{10}	1.4	56	8.6	1025	$10{\cdot}10^3$	56	$82{\cdot}10^3$	144
AllPar_{50}	3.2	44	93.3	144	TO	TO	TO	TO
AllPar_{500}	361.9	40	$8{\cdot}10^3$	156	TO	TO	TO	TO
$\text{Sharing}_{5,20}$	7.2	26	4	174	338.1	26	$1{\cdot}10^3$	174
$\text{Sharing}_{50,50}$	$2{\cdot}10^3$	38	527.2	162	255.4	38	$2{\cdot}10^3$	162
MMS	21.5	286	3.3	129	TO	TO	TO	TO
LBT	397.6	241	136.9	64	TO	TO	TO	TO

Table 7.2: The experimental results for language inclusion checks $\mathcal{L}(\mathcal{E}_A) \subseteq \mathcal{L}(\mathcal{E}_B)$ respectively $\mathcal{L}(\mathcal{N}^{\mathcal{E}_A}) \subseteq \mathcal{L}(\mathcal{N}^{\mathcal{E}_B})$.

enormous LBT model was a key motivator for using partial order reduction during state space exploration, since without such a technique it is hopeless to compute successor paths for this model. While unfolding construction on these benchmarks takes significant time, these experiments nevertheless demonstrate that unfolding-based partial order reduction scales to large problem instances. In conclusion, unfolding-based partial order reduction enables exploration of complex and highly concurrent models. However, for small and sequential models, the method introduces computational overhead.

7.6.3 Language Inclusion Experiment

In this experiment, we evaluate the language inclusion checking algorithm presented in Section 7.5 by using it as the basis of the strong killcheck during mutation-driven test generation. To this end, as described in Section 7.3, we check language inclusion between event structures $\mathit{unfold}(0, s)$ and $\mathit{unfold}(j, s)$ for some mutant id j and state s. We compare our language inclusion algorithm to automata based language inclusion by encoding event structures as nondeterministic finite automata, as described in Section 7.5.2, and minimize the resulting automata as well as check language inclusion for them using the tool RABIT [MC13].

We present the results of our experimental evaluation in Table 7.2. For every benchmark, we report measurements of language inclusion checks for the largest event structure corresponding to $\mathit{unfold}(0, s)$ for some action system state s of the respective model that is constructed during test

case generation on the respective model. We report measurements of event structure based language inclusion $\mathcal{L}(\mathcal{E}_A) \subseteq \mathcal{L}(\mathcal{E}_B)$ and automaton based language inclusion $\mathcal{L}(\mathcal{N}^{\mathcal{E}_A}) \subseteq \mathcal{L}(\mathcal{N}^{\mathcal{E}_B})$. We separate the results into the cases where language inclusion holds (**Inclusion**) respectively does not hold (**Non-Inclusion**). Columns Num show the number of inclusion checks performed on the respective event structure and with the respective language inclusion result. Columns Time show the average time for the respective inclusion checks in milliseconds.

The reported time for language inclusion of event structures is the time for calculation of the maximal configurations plus the time for the actual language inclusion check. The reported time for language inclusion of automata is the time for the language inclusion check on a pre-reduced automaton. The construction and minimization of the NFAs is not included.

The results show that our language inclusion algorithm performs well on models with a lot of concurrency, i.e. those with high **PC**, as reported in Table 7.1. Furthermore, the automaton translation clearly fails in cases with lots of concurrency that are easy for our method (c.f. the **AllPar** benchmark). For these examples our algorithm is very useful. This result is not surprising, since our method exploits concurrency, whereas the NFA encoding does not include any notion of concurrency. Nevertheless, the result demonstrates that the benefits of exploiting concurrency with our method outweigh optimizations and fine-tuning of a well established language inclusion algorithm that has no notion of concurrency.

However, as the **Sharing** benchmark shows, the inability of prime event structures to encode shared causes of events is a limitation of the approach. In contrast, for such examples the automaton representation can be significantly more compact than the event structure representation, rendering the automaton-based language inclusion superior.

Finally, our algorithm tends to be faster when there is no inclusion. This is not surprising, since our algorithm can exit early as soon as we find a counter-example. Interestingly, for NFA-based language inclusion it is exactly the opposite, where showing non-inclusion consistently takes more time than showing inclusion.

7.7 Related Work

7.7.1 Partial Order Reduction-Based Testing

Partial order reduction (POR) is a method to efficiently explore concurrent systems, by circumventing exploration of paths that correspond to the same Mazurkiewicz traces as previously explored paths. Traditionally, POR is used in model checking [God96, CGP99, CGMP99]. Dynamic partial order reduction (DPOR) [FG05] combines the basic POR idea with dynamic execution and is thus sometimes referred to as a testing approach [TKL+12]. Typically DPOR is performed over fixed input, thus we view this notion of testing as somewhat of a stretch of the concept.

However, DPOR was combined with dynamic symbolic execution [GKS05], which continuously executes a program and constructs new inputs that reach uncovered branches. This approach was demonstrated on concurrent Java programs [SA06, KSH12, SKH12, KSH13, KSH15, KH18]

and concurrent C programs [SBR+20]. In particular, [SBR+20] leverage the event structure unfolding semantics presented in [RSSK15, NRS+18], which we also used as basis to construct event structure unfoldings from action systems. In contrast, unfoldings are directly used to do testing of concurrent systems in [KSH12, KSH15, KH18].

In a similar fashion, [FHRV13] presents dynamic symbolic execution for multi-threaded C programs, where in contrast to partial order semantics, interference scenarios are automatically identified that subsume behaviorally equivalent interleavings of multi-threaded programs.

A transfer of the idea to combine dynamic symbolic execution with partial order semantics to our domain is interesting future work. However, there are significant challenges, such as establishing dynamic symbolic execution for reactive systems and combining these techniques with mutation killing.

7.7.2 Event Structures

Prime event structures are a widely used formalism to express concurrency of discrete systems [Win88] that can be obtained from transition systems via the method presented in [RSSK15], or its extended version in [NRS+18]. There are multiple other variants of event structures, such as stable event structures [Win88] and flow event structures [Bou90]. Studying language inclusion for these event structure variants is interesting future work.

Event structure containment based on causality and conflict refinement is considered in [Win88, Win16]. However, as we demonstrate in our work, causality preservation is not necessary for language inclusion. In [vGG89, VGG01] equivalence of event structures under action refinement is investigated. This line of research is orthogonal to our approach, as it considers refinement of event structures, while we compare event structures that can be obtained in multiple different ways. Moreover, there is almost never language inclusion between an event structure and the event structure with refined actions by design.

Model checking over particular types of event structures has been studied in [Pen97] for event structures labeled with atomic propositions and in [Mad03] for event structures labeled with trace languages. However, the proposed model checking methods are not based on language inclusion, which is one of the interesting future directions for our research. Instead, formulas are directly interpreted over the event structure. In [EH08] model checking of Petri net unfoldings is discussed in detail. In particular, several SAT encodings of interesting problems, such as reachability and deadlock freedom, over event structures are presented. These encodings could not only provide an alternative proof of the NP-completeness of language membership to the one presented in this paper, but also a potential solution for language inclusion via an encoding of event structure semantics into quantified Boolean formulas.

7.7.3 Trace Theory

Several formalisms to express concurrency of discrete systems have been proposed and their relationships have been worked out in [WN95]. In particular, trace languages and Petri nets

are formalisms closely related to event structures, for which languages and language related problems have been studied.

The theory of trace languages [Maz86, Maz95] studies closure of string languages under independence relations. [ČCH+17] presents an efficient method to show trace language inclusion over languages defined by non-deterministic finite automata.

In [DM97] decidability results of rational trace languages are studied. In particular, it is shown that language inclusion of rational (closed under union, concatenation, and Kleene-star) trace languages is decidable if and only if the common independence relation is transitive. Language membership for context free and regular trace languages was shown to be NP-complete in [BMS89]. In [BEL17], comparison of concurrent programs via trace languages is studied. The suggested trace languages abstract the program executions by considering statement ordering, as well as read and write accesses on a subset of relevant variables and synchronization primitives. Trace language refinement is then reduced to assertion checking. Interestingly, for Boolean programs this refinement check has complexity Δ_2^P for bounded abstraction precision and Σ_2^P for unbounded abstraction precision. In contrast to arbitrary event labels considered in our work, the authors of [BEL17] consider refinement on languages of a more concrete program and dependency model.

Our problem is orthogonal to trace language inclusion in three aspects. Firstly, we do not assume the independence relations of the compared systems to be equal. Secondly, we do not require the independence relation to be defined over labels. That is, we can study systems where two labels occur concurrently in one place, while the labels occur sequentially in another. This can occur because different events can have the same label. Finally, in contrast to automata, which are often used to define trace languages, event structures are acyclic. Therefore, event structures are less expressive than automata. However, the price of the additional expressivity is that trace language inclusion over automata is undecidable in general [BMS82], whereas our problem is decidable.

7.7.4 Petri Nets

Petri nets are a formalism for concurrent systems that is closely related to event structures [NPW81]. The correspondence of trace theory to unfoldings of Petri nets is worked out in [EH08]. A manifold of complexity questions have been studied for Petri nets, see [JLL77, Esp96, EN94] for surveys. In particular, language related problems of labeled Petri nets have been studied, see [Pet19, GG99] for an overview over the types of considered languages and complexity results. Since Petri nets typically describe languages on infinite words and many Petri net related problems are undecidable, complexity results on language related Petri nets problems focus on establishing the boundary between decidability and undecidability. Language inclusion and equivalence were shown to be undecidable for a wide range of types of Petri nets [Gra79, Jan01, EN94]. Language inclusion is decidable for languages of firing of regular Petri nets [VV81] and certain types of deterministic Petri nets [GG99]. In contrast, language membership is decidable for a large class of Petri nets and language types [Pet19, Hac76]. Similarly to trace languages, the additional expressivity of Petri nets over finite prime event structures mani-

fests in increased complexity of solving language inclusion. However, as we demonstrated with our application, finite prime event structures are sufficient for interesting practical problems.

7.7.5 Language and Automaton Theory

Finite asynchronous automata [Fat13] express concurrent systems succinctly in the same spirit as prime event structures. Furthermore, asynchronous automata accept trace languages [Zie87]. However, to the best of our knowledge, there is neither an algorithm, nor a tool to check language inclusion for (loop free) finite asynchronous automata.

Language inclusion of regular languages is a classic problem of computer science [HMU13, MS72, Fri76, SHI85]. Algorithms for the problem are well studied and highly optimized [MC13, ACH+10, ČCH+17]. However, as we demonstrate in Section 7.6, our procedure can outperform these algorithms in the realm of highly concurrent systems. Nevertheless, adapting methods used in classic automaton based language inclusions to event structures is interesting future work.

CHAPTER 8

Conclusions and Future Work

Conclusion

In validation and verification activities, one is typically confronted with trade-offs between rigor and scalability. The extreme on the rigor side are formal proofs of correctness, while the extreme on the scaling side are unit tests or simply observing a deployed system. Whereas the former approach does not scale to many problems, not only due to computational complexity, but also due to the inability to capture every detail of systems formally - relying on the latter approach for verifying one's system can become very costly when errors are found late in the development cycle. Model-based testing in general can be seen as a trade-off between the scaling of code-based testing and the rigor of model checking. Both model checking-based testing as well as mutation testing shift this trade-off towards the rigor side. In this thesis, we are concerned with enabling the combination of these two techniques and regaining their scaling properties.

The combination of model checking-based testing and mutation testing is enabled via the characterization of mutation killing as a hyperproperty and the encodings of these properties in logics for hyperproperties. By abstracting the test case generation engine from the encoding, future improvements in scaling properties of the emerging field of hyperproperty model checking will directly benefit our approach. Furthermore, since the encoding does not require knowledge of the structure of models, but only of the sets of inputs and outputs, it immediately scales better in terms of necessary engineering effort in comparison to previous model checking-based mutation testing approaches, built on top of trap properties or model duplication.

With our branching search-based test case generation algorithm we emphasize scaling even further. Its parameter- and heuristic space allow for a fine-grained positioning on the rigor versus scaling axis. Small models can be explored exhaustively, whereas a more limited search can be applied to large models in order to obtain test cases on models of sizes for which rigid methods would typically not yield any results. In addition, the algorithm mitigates a well known scalability problem of mutation testing, which is the large number of repeated executions for evaluating mutants, via lazy mutation execution. Finally, we enable scaling to highly concurrent systems by

applying partial order semantics to the model exploration and by formulating mutation killing as a language inclusion problem over the graph structure reflecting these semantics. Since neither the theoretical nor the practical side of the language inclusion problem over event structures was developed before, we filled this gap and thereby contribute to the event structure community.

Future Work

Clearly, the quest for even better validation and verification methods is never finished and this work can be extended in multiple directions. One particularly promising direction is the combination of exploration-based mutation testing, yielding a high base mutation coverage, with symbolic methods, yielding rigorous corner case detection. Towards such an approach, hyperproperty-sensitive concolic execution is an interesting research direction that could be both highly useful for test case generation and interesting to the broader verification community.

Highly distributed systems in the emerging internet of things offer a promising application domain for the methods presented in this work. Firstly, a holistic view on systems that are comprised of a myriad of actors is necessary when trying to reason about their correctness or safety. Secondly, such a holistic view is bound to use some form of model-based approach, since an explicit representation of all of the heterogeneous actors is not feasible. Thirdly, the interactions between the actors is likely to be highly concurrent and largely independent of each other. Finally, since more and more aspects of our lives will depend on such systems, it will be increasingly important to assure their correct functioning and the methods presented in Chapter 7 offer an ideal basis to testing such systems.

List of Figures

List of Tables

List of Algorithms

Bibliography

[AAJ+14] Bernhard K. Aichernig, Jakob Auer, Elisabeth Jöbstl, Robert Korosec, Willibald Krenn, Rupert Schlick, and Birgit Vera Schmidt. Model-based mutation testing of an industrial measurement device. TAP, pages 1–19, 2014.

[ABJ+15a] B. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick, and S. Tiran. MoMuT::UML model-based mutation testing for UML. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, ICST, pages 1–8, April 2015.

[ABJ+15b] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, Willibald Krenn, Rupert Schlick, and Stefan Tiran. Killing strategies for model-based mutation testing. *Softw. Test., Verif. Reliab.*, 25(8):716–748, 2015.

[ABJK11] Bernhard K Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. Efficient mutation killers in action. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 120–129. IEEE, 2011.

[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*, ICSE, pages 402–411. ACM, 2005.

[ABM98] Paul E Ammann, Paul E Black, and William Majurski. Using model checking to generate tests from specifications. In *Formal Engineering Methods, 1998. Proceedings. Second International Conference on*, pages 46–54. IEEE, 1998.

[Abr10a] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[Abr10b] Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[ACH+10] Parosh Aziz Abdulla, Yu-Fang Chen, Lukáš Holík, Richard Mayr, and Tomáš Vojnar. When simulation meets antichains. In *International Conference on*

Tools and Algorithms for the Construction and Analysis of Systems, pages 158–174. Springer, 2010.

[AGR15] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. Using mutation to assess fault detection capability of model review. *Softw. Test., Verif. Reliab.*, 25(5-7):629–652, 2015.

[AGR17] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. Nuseen: A tool framework for the nusmv model checker. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, pages 476–483. IEEE Computer Society, 2017.

[AGV15] Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. Generating tests for detecting faults in feature models. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE, 2015.

[AGV16] Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. Automatic detection and removal of conformance faults in feature models. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 102–112. IEEE, 2016.

[AH09] Bernhard K. Aichernig and Jifeng He. Mutation testing in UTP. *Formal Asp. Comput.*, 21(1-2):33–64, 2009.

[AHL14a] Bernhard K. Aichernig, Klaus Hörmaier, and Florian Lorber. Debugging with timed automata mutations. In Andrea Bondavalli and Felicita Di Giandomenico, editors, *Computer Safety, Reliability, and Security - 33rd International Conference, SAFECOMP 2014, Florence, Italy, September 10-12, 2014. Proceedings*, volume 8666 of *Lecture Notes in Computer Science*, pages 49–64. Springer, 2014.

[AHL+14b] Bernhard K. Aichernig, Klaus Hörmaier, Florian Lorber, Dejan Nickovic, Rupert Schlick, Didier Simoneau, and Stefan Tiran. Integration of requirements engineering and test-case generation via OSLC. In *2014 14th International Conference on Quality Software, Allen, TX, USA, October 2-3, 2014*, pages 117–126. IEEE, 2014.

[AHL+15] Bernhard K. Aichernig, Klaus Hörmaier, Florian Lorber, Dejan Nickovic, and Stefan Tiran. Require, test and trace IT. In Manuel Núñez and Matthias Güdemann, editors, *Formal Methods for Industrial Critical Systems - 20th International Workshop, FMICS 2015, Oslo, Norway, June 22-23, 2015 Proceedings*, volume 9128 of *Lecture Notes in Computer Science*, pages 113–127. Springer, 2015.

[AHL+17] Bernhard K. Aichernig, Klaus Hörmaier, Florian Lorber, Dejan Nickovic, and Stefan Tiran. Require, test, and trace IT. *Int. J. Softw. Tools Technol. Transf.*, 19(4):409–426, 2017.

[AJT14] Bernhard K. Aichernig, Elisabeth Jöbstl, and Stefan Tiran. Model-based mutation testing via symbolic refinement checking. 2014.

[AL15] Bernhard K. Aichernig and Florian Lorber. Towards generation of adaptive test cases from partial models of determinized timed automata. In *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*, pages 1–6. IEEE Computer Society, 2015.

[ALN13] Bernhard K. Aichernig, Florian Lorber, and Dejan Nickovic. Time for mutants - model-based mutation testing with timed automata. In Margus Veanes and Luca Viganò, editors, *Tests and Proofs - 7th International Conference, TAP@STAF 2013, Budapest, Hungary, June 16-20, 2013. Proceedings*, volume 7942 of *Lecture Notes in Computer Science*, pages 20–38. Springer, 2013.

[AM10] Salem Fawaz Adra and Phil McMinn. Mutation operators for agent-based models. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010, Workshops Proceedings*, pages 151–156. IEEE Computer Society, 2010.

[AME+15] Vincent Aranega, Jean-Marie Mottu, Anne Etien, Thomas Degueule, Benoit Baudry, and Jean-Luc Dekeyser. Towards an automation of the mutation analysis dedicated to model transformation. *Software Testing, Verification and Reliability*, 25(5-7):653–683, 2015.

[AMM+18] Bernhard K. Aichernig, Wojciech Mostowski, Mohammad Reza Mousavi, Martin Tappler, and Masoumeh Taromirad. Model learning and model-based testing. In Amel Bennaceur, Reiner Hähnle, and Karl Meinke, editors, *Machine Learning for Dynamic Software Analysis: Potentials and Limits - International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers*, volume 11026 of *Lecture Notes in Computer Science*, pages 74–100. Springer, 2018.

[AS11] Baris Akgün and Mike Stilman. Sampling heuristics for optimal motion planning in high dimensions. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2011, San Francisco, CA, USA, September 25-30, 2011*, pages 2640–2645. IEEE, 2011.

[Ayc03] John Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35(2):97–113, 2003.

[BBH+16] Fevzi Belli, Christof J Budnik, Axel Hollmann, Tugkan Tuglular, and W Eric Wong. Model-based mutation testingapproach and case studies. *Science of Computer Programming*, 120:25–48, 2016.

[BBTF11] Fevzi Belli, Mutlu Beyazit, Tomohiko Takagi, and Zengo Furukawa. Mutation testing of "go-back" functions based on pushdown automata. In *Fourth*

IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011, pages 249–258. IEEE Computer Society, 2011.

[BBTF12] Fevzi Belli, Mutlu Beyazit, Tomohiko Takagi, and Zengo Furukawa. Model-based mutation testing using pushdown automata. *IEICE TRANSACTIONS on Information and Systems*, 95(9):2211–2218, 2012.

[BCLM03] Michael S Branicky, Michael M Curtiss, Joshua A Levine, and Stuart B Morgan. Rrts for nonlinear, discrete, and hybrid planning and control. In *Decision and Control, 2003. Proceedings. 42nd IEEE Conference on*, volume 1, pages 657–663. IEEE, 2003.

[BDD+15] Sébastien Bardin, Mickaël Delahaye, Robin David, Nikolai Kosmatov, Mike Papadakis, Yves Le Traon, and Jean-Yves Marion. Sound and quasi-complete detection of infeasible test requirements. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, pages 1–10, 2015.

[BDH+19] Michael J. Butler, Dana Dghaym, Thai Son Hoang, Tope Omitola, Colin F. Snook, Andreas Fellner, Rupert Schlick, Thorsten Tarrach, Tomas Fischer, and Peter Tummeltshammer. Behaviour-driven formal model development of the ETCS hybrid level 3. In Jun Pang and Jing Sun, editors, *24th International Conference on Engineering of Complex Computer Systems, ICECCS 2019, Guangzhou, China, November 10-13, 2019*, pages 97–106. IEEE, 2019.

[BDLS80] Timothy A Budd, Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 220–233, 1980.

[BEL17] Ahmed Bouajjani, Constantin Enea, and Shuvendu K. Lahiri. Abstract semantic diffing of evolving concurrent programs. In Francesco Ranzato, editor, *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*, volume 10422 of *Lecture Notes in Computer Science*, pages 46–65. Springer, 2017.

[ben] Fles language inclusion benchmarks. `https://git-service.ait.ac.at/sct-dse-public/les-language-inclusion`. Uploaded: 2019-04-22.

[BF18] Borzoo Bonakdarpour and Bernd Finkbeiner. The complexity of monitoring hyperproperties. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 162–174. IEEE, 2018.

[BFP08] Nicola Bombieri, Franco Fummi, and Graziano Pravadelli. A mutation model for the systemc TLM 2.0 communication interfaces. In Donatella Sciuto, editor, *Design, Automation and Test in Europe, DATE 2008, Munich, Germany, March 10-14, 2008*, pages 396–401. ACM, 2008.

[BG85] Timothy A. Budd and Ajei S. Gopal. Program testing by specification mutation. *Comput. Lang.*, 10(1):63–73, 1985.

[BHW11] Armin Biere, Keijo Heljanko, and Siert Wieringa. AIGER 1.9 and beyond, 2011. Available at `fmv.jku.at/hwmcc11/beyond1.pdf`.

[Bin00] Robert Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000.

[BJK+05] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-based testing of reactive systems: advanced lectures*, volume 3472. Springer, 2005.

[BKC14] Sébastien Bardin, Nikolai Kosmatov, and François Cheynier. Efficient leveraging of symbolic execution to advanced coverage criteria. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, pages 173–182, 2014.

[BKS83] Ralph-Johan Back and Reino Kurki-Suonio. Decentralization of process nets with centralized control. In *Proceedings of the 2nd Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 1983)*, PODC, pages 131–142. ACM, 1983.

[BKS98] Marcello M. Bonsangue, Joost N. Kok, and Kaisa Sere. An approach to object-orientation in action systems. In Johan Jeuring, editor, *Mathematics of Program Construction, MPC'98, Marstrand, Sweden, June 15-17, 1998, Proceedings*, volume 1422 of *MPC*, pages 68–95. Springer, 1998.

[BLDS78] Timothy A. Budd, Richard J. Lipton, Richard A. DeMillo, and Frederick G. Sayward. The design of a prototype mutation system for program testing. In Sakti P. Ghosh and Leonard Y. Liu, editors, *American Federation of Information Processing Societies: 1978 National Computer Conference, June 5-8, 1978, Anaheim, CA, USA*, volume 47 of *AFIPS Conference Proceedings*, pages 623–629. AFIPS Press, 1978.

[BLDS79] Timothy A Budd, Richard J Lipton, Richard A DeMillo, and Frederick G Sayward. Mutation analysis. Technical report, DTIC Document, 1979.

[BMS82] Alberto Bertoni, Giancarlo Mauri, and Nicoletta Sabadini. Equivalence and membership problems for regular trace languages. In *International Colloquium on Automata, Languages, and Programming*, pages 61–71. Springer, 1982.

[BMS89] Alberto Bertoni, Giancarlo Mauri, and Nicoletta Sabadini. Membership problems for regular and context-free trace languages. *Information and Computation*, 82(2):135–150, 1989.

[Bou90] Gérard Boudol. Flow event structures and flow nets. In *LITP Spring School on Theoretical Computer Science*, pages 62–95. Springer, 1990.

[BOY00] Paul E Black, Vadim Okun, and Yaacov Yesha. Mutation operators for specifications. In *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*, pages 81–88. IEEE, 2000.

[BOY01] Paul E Black, Vadim Okun, and Yaacov Yesha. Mutation of model checker specifications for test generation and evaluation. In *Mutation testing for the new century*, pages 14–20. Springer, 2001.

[BPG07] Sergiy Boroday, Alexandre Petrenko, and Roland Groz. Can a model checker generate tests for non-deterministic systems? *Electronic Notes in Theoretical Computer Science*, 190(2):3–19, 2007.

[BPGQ02] Sergiy Boroday, Alexandre Petrenko, Roland Groz, and Yves-Marie Quemener. Test generation for CEFSM combining specification and fault coverage. In Ina Schieferdecker, Hartmut König, and Adam Wolisz, editors, *Testing of Communicating Systems XIV, Applications to Internet Technologies and Services, Proceedings of the IFIP 14th International Conference on Testing Communicating Systems - TestCom 2002, Berlin, Germany, March 19-22, 2002*, volume 210 of *IFIP Conference Proceedings*, pages 355–372. Kluwer, 2002.

[BSB17] Noel Brett, Umair Siddique, and Borzoo Bonakdarpour. Rewriting-based runtime verification for alternation-free hyperltl. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, volume 10206 of *Lecture Notes in Computer Science*, pages 77–93, 2017.

[BVCU07] Samrat S. Batth, Elisangela Rodrigues Vieira, Ana R. Cavalli, and M. Ümit Uyar. Specification of timed EFSM fault models in SDL. In John Derrick and Jüri Vain, editors, *Formal Techniques for Networked and Distributed Systems - FORTE 2007, 27th IFIP WG 6.1 International Conference, Tallinn, Estonia, June 27-29, 2007, Proceedings*, volume 4574 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2007.

[ČCH+17] Pavol Černý, Edmund M. Clarke, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, Roopsha Samanta, and Thorsten Tarrach. From non-preemptive to preemptive scheduling using synchronization synthesis. *Formal Methods in System Design*, 50(2):97–139, Jun 2017.

[CDH13] John A. Clark, Haitao Dan, and Robert M. Hierons. Semantic mutation testing. *Sci. Comput. Program.*, 78(4):345–363, 2013.

[CDK85] M Chandrasekharan, B Dasarathy, and Zen Kishimoto. Requirements-based testing of real-time systems. modeling for testability. *Computer*, (4):71–80, 1985.

[CFHH19] Norine Coenen, Bernd Finkbeiner, Christopher Hahn, and Jana Hofmann. The hierarchy of hyperlogics. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*, pages 1–13. IEEE, 2019.

[CFK+14] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. *Temporal Logics for Hyperproperties*, pages 265–284. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[CFST19] Norine Coenen, Bernd Finkbeiner, César Sánchez, and Leander Tentrup. Verifying hyperliveness. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 121–139. Springer, 2019.

[CGMP99] Edmund M Clarke, Orna Grumberg, Marius Minea, and Doron Peled. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer*, 2(3):279–287, 1999.

[CGP99] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, December 1999.

[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 TH Tse. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.

[CLOM08] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Artoo: adaptive random testing for object-oriented software. In *Proceedings of the 30th international conference on Software engineering*, pages 71–80. ACM, 2008.

[CS10] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[CYB93] Szu-Tsung Cheng, Gary York, and Robert K Brayton. Vl2mv: A compiler from verilog to blif-mv. *HSIS Distribution*, 1993.

[DDD+15] Tommaso Dreossi, Thao Dang, Alexandre Donzé, James Kapinski, Xiaoqing Jin, and Jyotirmoy V Deshmukh. Efficient guiding strategies for testing of temporal properties of hybrid systems. In *NASA Formal Methods Symposium*, NFM, pages 127–142. Springer, 2015.

[DGM+88] Richard A DeMillo, Dany S Guindi, WM McCracken, A Jefferson Offutt, and Kim N King. An extended overview of the mothra software testing environment. In *Workshop on Software Testing, Verification, and Analysis*, pages 142–143. IEEE Computer Society, 1988.

[Dij75] Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[DJ19] Daniel Dietsch and Marie-Christine Jakobs. Vmcai 2020 virtual machine, November 2019.

[DLS78] R.A. Demillo, R.J. Lipton, and F.G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41, April 1978.

[DM90] Richard Demillo and Aditya Mathur. On the use of software artifacts to evaluate the effectiveness of mutation analysis for detecting errors in production software. 01 1990.

[DM97] Volker Diekert and Yves Métivier. Partial commutation and traces. In *Handbook of formal languages*, pages 457–533. Springer, 1997.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. volume 4963, pages 337–340, 2008.

[DNSVT07] Arilo C Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H Travassos. A survey on model-based testing approaches: a systematic review. pages 31–36. ACM, 2007.

[DO91] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Trans. Software Eng.*, 17(9):900–910, 1991.

[DT96] Muriel Daran and Pascale Thévenod-Fosse. Software error analysis: A real case study involving real faults and mutations. In Steve J. Zeil and Will Tracz, editors, *Proceedings of the 1996 International Symposium on Software Testing and Analysis, ISSTA 1996, San Diego, CA, USA, January 8-10, 1996*, pages 158–171. ACM, 1996.

[ECO+16] Eduard Paul Enoiu, Adnan Causevic, Thomas J. Ostrand, Elaine J. Weyuker, Daniel Sundmark, and Paul Pettersson. Automated test generation using model checking: an industrial evaluation. *STTT*, 18(3):335–353, 2016.

[EFDYB12] KA El-Fakih, Rita Dorofeeva, NV Yevtushenko, and GV Bochmann. Fsm-based testing from user defined faults adapted to incremental and mutation testing. *Programming and Computer Software*, 38(4):201–209, 2012.

[EH00] Javier Esparza and Keijo Heljanko. A new unfolding approach to ltl model checking. In *International Colloquium on Automata, Languages, and Programming*, pages 475–486. Springer, 2000.

[EH08] Javier Esparza and Keijo Heljanko. *Unfoldings: a partial-order approach to model checking*. Springer Science & Business Media, 2008.

[EKSW05] Keith Ellul, Bryan Krawetz, Jeffrey Shallit, and Ming-wei Wang. Regular expressions: New results and open problems. *Journal of Automata, Languages and Combinatorics*, 10(4):407–437, 2005.

[EN94] Javier Esparza and Mogens Nielsen. Decidability issues for Petri nets - a survey. *Bulletin of the EATCS*, 52:244–262, 1994.

[ESC+16] Eduard Paul Enoiu, Daniel Sundmark, Adnan Causevic, Robert Feldt, and Paul Pettersson. Mutation-based test generation for PLC embedded software using model checking. In Franz Wotawa, Mihai Nica, and Natalia Kushik, editors, *Testing Software and Systems - 28th IFIP WG 6.1 International Conference, ICTSS 2016, Graz, Austria, October 17-19, 2016, Proceedings*, volume 9976 of *Lecture Notes in Computer Science*, pages 155–171, 2016.

[Esp96] Javier Esparza. Decidability and complexity of Petri net problems - an introduction. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*, volume 1491 of *Lecture Notes in Computer Science*, pages 374–428. Springer, 1996.

[FA11] Gordon Fraser and Andrea Arcuri. Evolutionary generation of whole test suites. In Manuel Núñez, Robert M. Hierons, and Mercedes G. Merayo, editors, *Proceedings of the 11th International Conference on Quality Software, QSIC 2011, Madrid, Spain, July 13-14, 2011.*, QSIC, pages 31–40. IEEE Computer Society, 2011.

[Fat13] Nazim Fates. A guided tour of asynchronous cellular automata. In *International Workshop on Cellular Automata and Discrete Complex Systems*, pages 15–30. Springer, 2013.

[FBW19] Andreas Fellner, Mitra Tabaei Befrouei, and Georg Weissenbacher. Mutation testing with hyperproperties. In Peter Csaba Ölveczky and Gwen Salaün, editors, *Software Engineering and Formal Methods - 17th International Conference, SEFM 2019, Oslo, Norway, September 18-20, 2019, Proceedings*, volume 11724 of *Lecture Notes in Computer Science*, pages 203–221. Springer, 2019.

[FBW20] Andreas Fellner, Mitra Tabaei Befrouei, and Georg Weissenbacher. Mutation testing with hyperproperties. *Software and Systems Modeling*, Special Issue, 2020. Accepted. Publication pending.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *ACM Sigplan Notices*, volume 40, pages 110–121. ACM, 2005.

[FG09] Gordon Fraser and Angelo Gargantini. An evaluation of model checkers for specification based test case generation. In *Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009*, pages 41–50. IEEE Computer Society, 2009.

[FH16] Bernd Finkbeiner and Christopher Hahn. Deciding hyperproperties. In Josée Desharnais and Radha Jagadeesan, editors, *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, volume 59 of *LIPIcs*, pages 13:1–13:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[FHRV13] Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. Con2colic testing. pages 37–47, 2013.

[FHS17] Bernd Finkbeiner, Christopher Hahn, and Marvin Stenger. Eahyper: Satisfiability, implication, and equivalence checking of hyperproperties. In *CAV (2)*, volume 10427 of *Lecture Notes in Computer Science*, pages 564–570. Springer, 2017.

[FHST19] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. Monitoring hyperproperties. *Formal Methods in System Design*, 54(3):336–363, 2019.

[FHT19] Bernd Finkbeiner, Lennart Haas, and Hazem Torfah. Canonical representations of k-safety hyperproperties. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 17–1714. IEEE, 2019.

[FKS06] Dave Ferguson, Nidhi Kalra, and Anthony Stentz. Replanning with rrts. In *Proceedings of the 2006 IEEE International Conference on Robotics and Automation, ICRA 2006, May 15-19, 2006, Orlando, Florida, USA*, pages 1243–1248. IEEE, 2006.

[FKS+17] Andreas Fellner, Willibald Krenn, Rupert Schlick, Thorsten Tarrach, and Georg Weissenbacher. Model-based, mutation-driven test case generation via heuristic-guided branching search. In Jean-Pierre Talpin, Patricia Derler, and Klaus Schneider, editors, *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2017, Vienna, Austria, September 29 - October 02, 2017*, pages 56–66. ACM, 2017.

[FKS+19] Andreas Fellner, Willibald Krenn, Rupert Schlick, Thorsten Tarrach, and Georg Weissenbacher. Model-based, mutation-driven test-case generation via heuristic-guided branching search. *ACM Trans. Embed. Comput. Syst.*, 18(1):4:1–4:28, January 2019.

[FMM+95] Sandra Camargo Pinto Ferraz Fabbri, José Carlos Maldonado, Paulo Cesar Masiero, Márcio Eduardo Delamaro, and E Wong. Mutation testing applied to validate specifications based on petri nets. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 329–337. Springer, 1995.

[FMMD99] Sandra Camargo Pinto Ferraz Fabbri, José Carlos Maldonado, Paulo Cesar Masiero, and Márcio Eduardo Delamaro. Proteum/fsm: A tool to support finite state machine validation based on mutation testing. In *19th International Conference of the Chilean Computer Science Society (SCCC '99), 11-13 November 1999, Talca, Chile*, pages 96–104. IEEE Computer Society, 1999.

[FMSM99] Sandra Camargo Pinto Ferraz Fabbri, José Carlos Maldonado, Tatiana Sugeta, and Paulo Cesar Masiero. Mutation testing applied to validate specifications based on statecharts. In *Proceedings 10th International Symposium on Software Reliability Engineering (Cat. No. PR00443)*, pages 210–219. IEEE, 1999.

[Fri76] Emily P Friedman. The inclusion problem for simple languages. *Theoretical Computer Science*, 1(4):297–316, 1976.

[FRS15] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for model checking HyperLTL and HyperCTL*. pages 30–48, 2015.

[FTW19] Andreas Fellner, Thorsten Tarrach, and Georg Weissenbacher. Language Inclusion for Finite Prime Event Structures Artifact, October 2019.

[FTW20] Andreas Fellner, Thorsten Tarrach, and Georg Weissenbacher. Language inclusion for finite prime event structures. In Dirk Beyer and Damien Zufferey, editors, *Verification, Model Checking, and Abstract Interpretation - 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16-21, 2020, Proceedings*, volume 11990 of *Lecture Notes in Computer Science*, pages 314–336. Springer, 2020.

[FW08] Gordon Fraser and Franz Wotawa. Using model-checkers to generate and analyze property relevant test-cases. *Softw. Qual. J.*, 16(2):161–183, 2008.

[FWA09a] Gordon Fraser, Franz Wotawa, and Paul Ammann. Issues in using model checkers for test case generation. *Journal of Systems and Software*, 82(9):1403 – 1418, 2009.

[FWA09b] Gordon Fraser, Franz Wotawa, and Paul E Ammann. Testing with model checkers: a survey. *Software Testing, Verification and Reliability*, 19(3):215–261, 2009.

[FWH96] Phyllis G Frankl, Stewart N Weiss, and Cang Hu. All-uses versus mutation testing: An experimental comparison of e ectiveness. 1996.

[FZ17] Bernd Finkbeiner and Martin Zimmermann. The first-order logic of hyperproperties. In *STACS*, volume 66 of *LIPIcs*, pages 30:1–30:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[GG99] Stephane Gaubert and Alessandro Giua. Petri net languages and infinite subsets of m. *J. Comput. Syst. Sci.*, 59(3):373–391, 1999.

[GH99] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. In *ACM SIGSOFT Software Engineering Notes*, volume 24, pages 146–162. Springer-Verlag, 1999.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[GMH81] John Gannon, Paul McMullin, and Richard Hamlet. Data abstraction, implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(3):211–223, 1981.

[GMM18] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: a survey. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, page 1219. ACM, 2018.

[GN97] Matthew J. Gallagher and V Lakshmi Narasimhan. Adtest: A test data generation suite for ada software systems. *IEEE Transactions on Software Engineering*, 23(8):473–484, 1997.

[God96] Patrice Godefroid. Partial-order methods for the verification of concurrent systems, 1996.

[Gon70] Guney Gonenc. A method for the design of fault detection experiments. *IEEE Trans. Computers*, 19(6):551–558, 1970.

[Gou83] J. S. Gourlay. A mathematical framework for the investigation of testing. *IEEE Transactions on Software Engineering*, SE-9(6):686–709, 1983.

[Gra79] Jan Grabowski. The unsolvability of some Petri net language problems. *Inf. Process. Lett.*, 9(2):60–63, 1979.

[Hac76] Michel Hack. *Decidability questions for Petri Nets*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1976.

[Ham77] Richard G. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. Software Eng.*, 3(4):279–290, 1977.

[Hel00] Keijo Heljanko. Model checking with finite complete prefixes is pspace-complete. In *International Conference on Concurrency Theory*, pages 108–122. Springer, 2000.

[Hen64] F. C. Hennie. Fault detecting experiments for sequential circuits. In *5th Annual Symposium on Switching Circuit Theory and Logical Design, Princeton, New Jersey, USA, November 11-13, 1964*, pages 95–110. IEEE Computer Society, 1964.

[HH14] Shai Haim and Marijn Heule. Towards ultra rapid restarts. *CoRR*, abs/1402.4413, 2014.

[HLSU02] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A temporal logic based theory of test coverage and generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 327–341. Springer, 2002.

[HM07] Robert M Hierons and Mercedes G Merayo. Mutation testing from probabilistic finite state machines. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 141–150. IEEE, 2007.

[HM09] Robert M. Hierons and Mercedes G. Merayo. Mutation testing from probabilistic and stochastic finite state machines. *J. Syst. Softw.*, 82(11):1804–1818, 2009.

[HMU13] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson, 3 edition, 2013.

[How82] William E. Howden. Weak mutation testing and completeness of test sets. *IEEE Trans. Software Eng.*, 8(4):371–379, 1982.

[HPLT14] Christopher Henard, Mike Papadakis, and Yves Le Traon. Mutalog: A tool for mutating logic formulas. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 399–404. IEEE, 2014.

[HPP+13a] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Assessing software product line testing via model-based mutation: An application to similarity testing. In *2013 IEEE Sixth international conference on software testing, verification and validation workshops*, pages 188–197. IEEE, 2013.

[HPP+13b] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Towards automated testing and fixing of re-engineered feature models. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1245–1248. IEEE, 2013.

[HPP+13c] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Assessing software product line testing via model-based mutation: An application to similarity testing. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013 Workshops Proceedings, Luxembourg, Luxembourg, March 18-22, 2013*, pages 188–197. IEEE Computer Society, 2013.

[HPT14] Christopher Henard, Mike Papadakis, and Yves Le Traon. Mutation-based generation of software product line test configurations. In Claire Le Goues and Shin Yoo, editors, *Search-Based Software Engineering - 6th International Symposium, SSBSE 2014, Fortaleza, Brazil, August 26-29, 2014. Proceedings*, volume 8636 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2014.

[HSB20] Tzu-Han Hsu, Cesar Sanchez, and Borzoo Bonakdarpour. Bounded model checking for hyperproperties. *arXiv preprint arXiv:2009.08907*, 2020.

[Jan01] Petr Jancar. Nonprimitive recursive complexity and undecidability for Petri net equivalences. *Theor. Comput. Sci.*, 256(1-2):23–30, 2001.

[JCS08] Leonard Jaillet, Juan Cortés, and Thierry Siméon. Transition-based RRT for path planning in continuous cost spaces. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems, September 22-26, 2008, Acropolis Convention Center, Nice, France*, pages 2145–2150. IEEE, 2008.

[JH11] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 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*, FSE, pages 654–665. ACM, 2014.

[JKRS76] W. H. Jessop, J. Richard Kane, S. Roy, and J. M. Scanlon. ATLAS - an automated software testing system. In Raymond T. Yeh and C. V. Ramamoorthy, editors, *Proceedings of the 2nd International Conference on Software Engineering, San Francisco, California, USA, October 13-15, 1976*, pages 629–635. IEEE Computer Society, 1976.

[JLL77] Neil D. Jones, Lawrence H. Landweber, and Y. Edmund Lien. Complexity of some problems in Petri nets. *Theor. Comput. Sci.*, 4(3):277–299, 1977.

[Jöb14] E Jöbstl. *Model-based mutation testing with constraint and SMT solvers*. PhD thesis, Ph. D. thesis, Graz University of Technology, Institute for Software Technology, 2014.

[Kap06] Kalpesh Kapoor. Formal analysis of coupling hypothesis for logical faults. *ISSE*, 2(2):80–87, 2006.

[Kar72] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[KH18] Kari Kähkönen and Keijo Heljanko. Testing programs with contextual unfoldings. *ACM Trans. Embedded Comput. Syst.*, 17(1):23:1–23:25, 2018.

[KIN15] Jaber Karimpour, Ayaz Isazadeh, and Ali A. Noroozi. Verifying observational determinism. In Hannes Federrath and Dieter Gollmann, editors, *ICT Systems Security and Privacy Protection - 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany, May 26-28, 2015, Proceedings*, volume 455 of *IFIP Advances in Information and Communication Technology*, pages 82–93. Springer, 2015.

[KL95] Ker-I Ko and Chih-Long Lin. On the complexity of min-max optimization problems and their approximation. In *Minimax and Applications*, pages 219–239. Springer, 1995.

[KL00] James J Kuffner and Steven M LaValle. RRT-connect: An efficient approach to single-query path planning. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 2 of *ICRA*, pages 995–1001. IEEE, 2000.

[Kor90] Bogdan Korel. Automated software test data generation. *IEEE Transactions on software engineering*, 16(8):870–879, 1990.

[KPLV+03] Gábor Kovács, Zoltán Pap, Dung Le Viet, Antal Wu-Hen-Chang, and Gyula Csopaki. Applying mutation analysis to sdl specifications. In *International SDL Forum*, pages 269–284. Springer, 2003.

[KPM10] Marinos Kintis, Mike Papadakis, and Nicos Malevris. Evaluating mutation testing alternatives: A collateral experiment. In *2010 Asia Pacific Software Engineering Conference*, pages 300–309. IEEE, 2010.

[KSA09] Willibald Krenn, Rupert Schlick, and Bernhard K. Aichernig. Mapping UML to labeled transition systems for test-case generation: A translation via object-oriented action systems. In *Proceedings of the 8th International Conference on Formal Methods for Components and Objects*, FMCO, pages 186–207, Berlin, Heidelberg, 2009. Springer-Verlag.

[KSH12] Kari Kähkönen, Olli Saarikivi, and Keijo Heljanko. Using unfoldings in automated testing of multithreaded programs. In Michael Goedicke, Tim Menzies, and Motoshi Saeki, editors, *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, pages 150–159. ACM, 2012.

[KSH13] Kari Kähkönen, Olli Saarikivi, and Keijo Heljanko. Lct: A parallel distributed testing tool for multithreaded java programs. *Electronic Notes in Theoretical Computer Science*, 296:253–259, 2013.

[KSH15] Kari Kähkönen, Olli Saarikivi, and Keijo Heljanko. Unfolding based automated testing of multithreaded programs. *Autom. Softw. Eng.*, 22(4):475–515, 2015.

[KV13] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 1–35. Springer, 2013.

[L+01] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA, Mar 2004.

[LAO+15] Birgitta Lindström, Sten F. Andler, Jeff Offutt, Paul Pettersson, and Daniel Sundmark. Mutating aspect-oriented models to test cross-cutting concerns. In *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*, pages 1–10. IEEE Computer Society, 2015.

[LaV98] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. 1998.

[Lip75] Richard J Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[LLGS17] Wenbin Li, Franck Le Gall, and Naum Spaseski. A survey on model-based testing tools for test case generation. In *International Conference on Tools and Methods for Program Analysis*, pages 77–89. Springer, 2017.

[LLNN17] Kim G. Larsen, Florian Lorber, Brian Nielsen, and Ulrik Nyman. Mutation-based test-case generation with ecdar. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2017, Tokyo, Japan, March 13-17, 2017*, pages 319–328. IEEE Computer Society, 2017.

[LM05a] Jian Bing Li and James Miller. Testing the semantics of W3C XML schema. In *29th Annual International Computer Software and Applications Conference, COMPSAC 2005, Edinburgh, Scotland, UK, July 25-28, 2005. Volume 1*, pages 443–448. IEEE Computer Society, 2005.

[LM05b] Jian Bing Li and James Miller. Testing the semantics of w3c xml schema. In *29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, volume 1, pages 443–448. IEEE, 2005.

[LM05c] Ling Liu and Huaikou Miao. Mutation operators for object-z specification. In *10th International Conference on Engineering of Complex Computer Systems (ICECCS 2005), 16-20 June 2005, Shanghai, China*, pages 498–506. IEEE Computer Society, 2005.

[LO01] Suet Chun Lee and Jeff Offutt. Generating test cases for xml-based web component interactions using mutation analysis. In *12th International Symposium on Software Reliability Engineering (ISSRE 2001), 27-30 November 2001, Hong Kong, China*, pages 200–209. IEEE Computer Society, 2001.

[LR09] Akash Lal and Thomas Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, 2009.

[LY96] David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[MAB+14] Gunter Mussbacher, Daniel Amyot, Ruth Breu, Jean-Michel Bruel, Betty H. C. Cheng, Philippe Collet, Benoît Combemale, Robert B. France, Rogardt Heldal, James H. Hill, Jörg Kienzle, Matthias Schöttle, Friedrich Steimann, Dave R. Stikkolorum, and Jon Whittle. The relevance of model-driven engineering thirty years from now. In Jürgen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfrán, editors, *Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings*, volume 8767 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2014.

[Mad03] P. Madhusudan. Model-checking trace event structures. In *18th IEEE Symposium on Logic in Computer Science (LICS 2003), 22-25 June 2003, Ottawa, Canada, Proceedings*, pages 371–380. IEEE Computer Society, 2003.

[Mar91] Brian Marick. The weak mutation hypothesis. In *Proceedings of the symposium on Testing, analysis, and verification*, pages 190–199, 1991.

[Maz86] Antoni W. Mazurkiewicz. Trace theory. In *Petri Nets: Central Models and Their Properties, Advances in Petri Nets*, volume 255, pages 279–324, 1986.

[Maz95] Antoni Mazurkiewicz. Introduction to trace theory. *The Book of Traces*, pages 3–41, 1995.

[MAZ+15] Lei Ma, Cyrille Artho, Cheng Zhang, Hiroyuki Sato, Johannes Gmeiner, and Rudolf Ramler. Grt: Program-analysis-guided random testing (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 212–223. IEEE, 2015.

[MC13] Richard Mayr and Lorenzo Clemente. Advanced automata minimization. In *ACM SIGPLAN Notices*, volume 48, pages 63–74. ACM, 2013.

[McL92] John McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1(1):37–58, 1992.

[McM92a] Kenneth L McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *International Conference on Computer Aided Verification*, pages 164–177. Springer, 1992.

[McM92b] McMillan, Kenneth L. The SMV system. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[McM04] Phil McMinn. Search-based software test data generation: A survey. *Software Testing Verification and Reliability*, 14(2):105–156, 2004.

[Mea55] George H Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955.

[MF11] Jan Malburg and Gordon Fraser. Combining search-based and constraint-based testing. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, ASE, pages 436–439. IEEE Computer Society, 2011.

[MFV15] Rui Angelo Matnei Filho and Silvia Regina Vergilio. A mutation and multi-objective test data generation approach for feature testing of software product lines. In *2015 29th Brazilian Symposium on Software Engineering*, pages 21–30. IEEE, 2015.

[MFV16] Rui A Matnei Filho and Silvia R Vergilio. A multi-objective test data generation approach for mutation testing of feature models. *Journal of Software Engineering Research and Development*, 4(1):4, 2016.

[MOSH09] Mohamed Mussa, Samir Ouchani, Waseem Al Sammane, and Abdelwahab Hamou-Lhadj. A survey of model-driven testing techniques. In Byoungju Choi, editor, *Proceedings of the Ninth International Conference on Quality Software, QSIC 2009, Jeju, Korea, August 24-25, 2009*, pages 167–172. IEEE Computer Society, 2009.

[MR01] Tafline Murnane and Karl Reed. On the effectiveness of mutation analysis as a black box testing technique. In *13th Australian Software Engineering Conference (ASWEC 2001), 26-28 August 2001, Canberra, Australia*, pages 12–20. IEEE Computer Society, 2001.

[MS72] Albert R Meyer and Larry J Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *SWAT (FOCS)*, pages 125–129, 1972.

[MW47] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[MW94] Aditya P Mathur and W Eric Wong. An empirical comparison of data flow and mutation-based test adequacy criteria. *Software Testing, Verification and Reliability*, 4(1):9–31, 1994.

[Nel89] Greg Nelson. A generalization of Dijkstra's calculus. *ACM Trans. Program. Lang. Syst.*, 11(4):517–561, 1989.

[NPW81] Mogens Nielsen, Gordon Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part i. *Theoretical Computer Science*, 13(1):85–108, 1981.

[NRS+18] Huyen T. T. Nguyen, César Rodríguez, Marcelo Sousa, Camille Coti, and Laure Petrucci. Quasi-optimal partial order reduction. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 354–371. Springer, 2018.

[OBY03] Vadim Okun, Paul E Black, and Yaacov Yesha. Testing with model checker: Insuring fault visibility. In *Proceedings of 2002 WSEAS international conference on system science, applied mathematics & computer science, and power engineering systems*, pages 1351–1356, 2003.

[Off89] A. Jefferson Offutt. The coupling effect: Fact or fiction. In Richard A. Kemmerer, editor, *Proceedings of the ACM SIGSOFT '89 Third Symposium on Testing, Analysis, and Verification, TAV 1989, Key West, Florida, USA, December 13-15, 1989*, pages 131–140. ACM, 1989.

[Off92] A. Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, 1(1):5–20, 1992.

[Off11] Jeff Offutt. A mutation carol: Past, present and future. *Information and Software Technology*, 53(10):1098–1107, 2011.

[OL91] A Jefferson Offutt and Stephen D Lee. How strong is weak mutation? In *Proceedings of the symposium on Testing, analysis, and verification*, pages 200–213, 1991.

[OL94] A Jefferson Offutt and Stephen D Lee. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, 1994.

[OPTZ96] A Jefferson Offutt, Jie Pan, Kanupriya Tewary, and Tong Zhang. An experimental evaluation of data flow and mutation testing. *Software: Practice and Experience*, 26(2):165–176, 1996.

[OU01] A Jefferson Offutt and Roland H Untch. Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century*, pages 34–44. Springer, 2001.

[PDMM94] S. C. Pinto Ferraz Fabbri, M. E. Delamaro, J. C. Maldonado, and P. C. Masiero. Mutation analysis testing for finite state machines. In *Proc. IEEE Int. Symp. Software Reliability Engineering*, pages 220–229, 1994.

[PE07] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA Companion 2007, pages 815–816. ACM, 2007.

[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.

[Pen97] Wojciech Penczek. Model-checking for a subclass of event structures. In Ed Brinksma, editor, *Tools and Algorithms for Construction and Analysis of Systems, Third International Workshop, TACAS '97, Enschede, The Netherlands, April 2-4, 1997, Proceedings*, volume 1217 of *Lecture Notes in Computer Science*, pages 145–164. Springer, 1997.

[Pet19] J.L. Peterson. *Petri Net Theory and the Modeling of Systems*. Independently Published, 2019.

[PH03] T Pyhala and Keijo Heljanko. Specification coverage aided test selection. In *Third International Conference on Application of Concurrency to System Design, 2003. Proceedings.*, pages 187–195. IEEE, 2003.

[PKZ+19] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: an analysis and survey. In *Advances in Computers*, volume 112, pages 275–378. Elsevier, 2019.

[PLEB07] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th international conference on Software Engineering*, ICSE, pages 75–84. IEEE Computer Society, 2007.

[PRWN16] Ingo Pill, Ivan Rubil, Franz Wotawa, and Mihai Nica. SIMULTATE: A toolset for fault injection and mutation testing of simulink models. In *Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, April 11-15, 2016*, pages 168–173. IEEE Computer Society, 2016.

[RBF11] Heinz Riener, Roderick Bloem, and Görschwin Fey. Test case generation from mutants using model checking techniques. In *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings*, pages 388–397. IEEE Computer Society, 2011.

[RC09] Chanchal K Roy and James R Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pages 157–166. IEEE, 2009.

[RFT16] Klaus Reichl, Tomas Fischer, and Peter Tummeltshammer. Using formal methods for verification and validation in railway. In Bernhard K. Aichernig and Carlo A. Furia, editors, *Tests and Proofs - 10th International Conference, TAP 2016, Held as Part of STAF 2016, Vienna, Austria, July 5-7, 2016, Proceedings*, volume 9762 of *Lecture Notes in Computer Science*, pages 3–13. Springer, 2016.

[RH01] Sanjai Rayadurgam and Mats Per Erik Heimdahl. Coverage based test-case generation using model checkers. In *Engineering of Computer Based Systems (ECBS)*, pages 83–91, 2001.

[RSSK15] César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. Unfolding-based Partial Order Reduction. In *26th International Conference on Concurrency Theory (CONCUR 2015)*, pages 456–469, 2015.

[SA06] Koushik Sen and Gul Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In Eyal Bin, Avi Ziv, and Shmuel Ur, editors, *Hardware and Software, Verification and Testing, Second International Haifa Verification Conference, HVC 2006, Haifa, Israel, October 23-26, 2006. Revised Selected Papers*, volume 4383 of *Lecture Notes in Computer Science*, pages 166–182. Springer, 2006.

[SAC14] Matthew Stephan, Manar H. Alalfi, and James R. Cordy. Towards a taxonomy for simulink model mutations. In *Seventh IEEE International Conference on*

Software Testing, Verification and Validation, ICST 2014 Workshops Proceedings, March 31 - April 4, 2014, Cleveland, Ohio, USA, pages 206–215. IEEE Computer Society, 2014.

[SASC13] Matthew Stephan, Manar H Alafi, Andrew Stevenson, and James R Cordy. Using mutation analysis for a model-clone detector comparison framework. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1261–1264. IEEE, 2013.

[SBR+20] Daniel Schemmel, Julian Büning, César Rodríguez, David Laprell, and Klaus Wehrle. Symbolic partial-order execution for testing multi-threaded programs. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 376–400. Springer, 2020.

[SCSP03] Thitima Srivatanakul, John A Clark, Susan Stepney, and Fiona Polack. Challenging formal specifications by mutation: a csp security example. In *Tenth Asia-Pacific Software Engineering Conference, 2003.*, pages 340–350. IEEE, 2003.

[SdSFLdSM00] Simone do Rocio Senger de Souza, Sandra Camargo Pinto Ferraz Fabbri, Wanderley Lopes de Souza, and José Carlos Maldonado. Mutation testing applied to estelle specifications. In *Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 8-Volume 8*, page 8011, 2000.

[SFBD08] Andre Suelflow, Goerschwin Fey, Roderick Bloem, and Rolf Drechsler. Using Unsatisfiable Cores to Debug Multiple Design Errors. In *Proceedings of the 18th ACM Great Lakes Symposium on VLSI*, GLSVLSI '08, pages 77–82, New York, NY, USA, 2008. ACM.

[SHI85] Richard Edwin Stearns and Harry B Hunt III. On the equivalence and containment problems for unambiguous regular expressions, regular grammars and finite automata. *SIAM Journal on Computing*, 14(3):598–611, 1985.

[SKH12] Olli Saarikivi, Kari Kähkönen, and Keijo Heljanko. Improving dynamic partial order reductions for concolic testing. In Jens Brandt and Keijo Heljanko, editors, *12th International Conference on Application of Concurrency to System Design, ACSD 2012, Hamburg, Germany, June 27-29, 2012*, pages 132–141. IEEE Computer Society, 2012.

[SL88] Deepinder Sidhu and T-K Leung. Fault coverage of protocol test methods. In *IEEE INFOCOM'88, Seventh Annual Joint Conference of the IEEE Computer and Communcations Societies. Networks: Evolution or Revolution?*, pages 80–81. IEEE Computer Society, 1988.

[SL10] Muhammad Shafique and Yvan Labiche. A systematic review of model based testing tool support. *Carleton University, Canada, Tech. Rep. Technical Report SCE-10-04*, pages 01–21, 2010.

[SL15] Muhammad Shafique and Yvan Labiche. A systematic review of state-based test tools. *Software Tools for Technology Transfer*, 17(1):59–76, 2015.

[SMC+17] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based gui testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 245–256, 2017.

[SMW04] Tatiana Sugeta, José Carlos Maldonado, and W Eric Wong. Mutation testing applied to validate sdl specifications. In *IFIP International Conference on Testing of Communicating Systems*, pages 193–208. Springer, 2004.

[SWZK17] Allison Sullivan, Kaiyuan Wang, Razieh Nokhbeh Zaeem, and Sarfraz Khurshid. Automated test generation and mutation testing for alloy. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, pages 264–275. IEEE Computer Society, 2017.

[SYR08] Manoranjan Satpathy, Anand Yeolekar, and S Ramesh. Randomized directed testing (redirect) for simulink/stateflow models. In *Proceedings of the 8th ACM international conference on Embedded software*, pages 217–226. ACM, 2008.

[TCMM98] Nigel Tracey, John Clark, Keith Mander, and John McDermid. An automated framework for structural test-data generation. In *Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on*, ASE, pages 285–288. IEEE, 1998.

[TKL+12] Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. Transdpor: A novel dynamic partial-order reduction technique for testing actor programs. In Holger Giese and Grigore Rosu, editors, *Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings*, volume 7273 of *Lecture Notes in Computer Science*, pages 219–234. Springer, 2012.

[TM08] Donald Thomas and Philip Moorby. *The Verilog® hardware description language*. Springer Science & Business Media, 2008.

[Tra10] Mark Trakhtenbrot. Implementation-oriented mutation testing of statechart models. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 120–125. IEEE, 2010.

[Tra17] Mark B. Trakhtenbrot. Mutation patterns for temporal requirements of reactive systems. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2017, Tokyo, Japan, March 13-17, 2017*, pages 116–121. IEEE Computer Society, 2017.

[Tre92] Gerrit Jan Tretmans. *A formal approach to conformance testing*. PhD thesis, University of Twente, Enschede, Netherlands, 1992.

[Tre96] Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.

[UPL12] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.

[vdMZ07] Ron van der Meyden and Chenyi Zhang. Algorithmic verification of noninterference properties. *Electr. Notes Theor. Comput. Sci.*, 168:61–75, 2007.

[vGG89] Rob van Glabbeek and Ursula Goltz. Refinement of actions in causality based models. In *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*, pages 267–300. Springer, 1989.

[VGG01] Rob Van Glabbeek and Ursula Goltz. Refinement of actions and equivalence notions for concurrent systems. *Acta Informatica*, 37(4-5):229–327, 2001.

[vGP09] Rob J van Glabbeek and Gordon D Plotkin. Configuration structures, event structures and Petri nets. *Theoretical Computer Science*, 410(41):4111–4159, 2009.

[VMGF13] Mattia Vivanti, Andre Mis, Alessandra Gorla, and Gordon Fraser. Search-based data-flow test generation. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, ISSRE, pages 370–379. IEEE, 2013.

[VPK04] Willem Visser, Corina S Psreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. *ACM SIGSOFT Software Engineering Notes*, 29(4):97–107, 2004.

[VV81] Rüdiger Valk and Guy Vidal-Naquet. Petri nets and regular languages. *J. Comput. Syst. Sci.*, 23(3):299–325, 1981.

[Wah95] K. S. How Tai Wah. Fault coupling in finite bijective functions. *Softw. Test. Verification Reliab.*, 5(1):3–47, 1995.

[Wah00] K. S. How Tai Wah. A theoretical study of fault coupling. *Softw. Test. Verification Reliab.*, 10(1):3–45, 2000.

[Wah03] K. S. How Tai Wah. An analysis of the coupling effect I: single test data. *Sci. Comput. Program.*, 48(2-3):119–161, 2003.

[WBP02] Joachim Wegener, Kerstin Buhr, and Hartmut Pohlheim. Automatic test data generation for structural testing of embedded software systems by evolutionary testing. In *GECCO*, volume 2 of *GECCO*, pages 1233–1240, 2002.

[Win88] Glynn Winskel. An introduction to event structures. In *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*, pages 364–397. Springer, 1988.

[Win16] Glynn Winskel. Event structures, stable families and concurrent games, 2016.

[WN95] Glynn Winskel and Mogens Nielsen. Handbook of logic in computer science (vol. 4). chapter Models for Concurrency, pages 1–148. Oxford University Press, Inc., New York, NY, USA, 1995.

[Woo93] M. R. Woodward. Errors in algebraic specifications and an experimental mutation testing tool. *Softw. Eng. J.*, 8(4):211–224, 1993.

[Wra76] Celia Wrathall. Complete sets and the polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):23–33, 1976.

[WSK18] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. Mualloy: a mutation testing framework for alloy. In *International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 29–32, 2018.

[XeAXW12] Dianxiang Xu, Omar el Ariss, Weifeng Xu, and Linzhang Wang. Testing aspect-oriented programs with finite state machines. *Softw. Test. Verification Reliab.*, 22(4):267–293, 2012.

[XOL05] Wuzhi Xu, Jeff Offutt, and Juan Luo. Testing web services by XML perturbation. In *16th International Symposium on Software Reliability Engineering (ISSRE 2005), 8-11 November 2005, Chicago, IL, USA*, pages 257–266. IEEE Computer Society, 2005.

[YCJ98] Hoijin Yoon, Byoungju Choi, and Jin-Ok Jeon. Mutation-based inter-class testing. In *Proceedings 1998 Asia Pacific Software Engineering Conference (Cat. No. 98EX240)*, pages 174–181. IEEE, 1998.

[ZC05] Yuan Zhan and John A Clark. Search-based mutation testing for simulink models. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, GECCO, pages 1061–1068. ACM, 2005.

[ZDK07] Songtao Zhang, Thomas Dean, and Scott Knight. Lightweight state based mutation testing for security. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 223–232. IEEE, 2007.

[Zie87] Wieslaw Zielonka. Notes on finite asynchronous automata. *RAIRO-Theoretical Informatics and Applications*, 21(2):99–135, 1987.

[ZSL+14] Tingliang Zhou, Haiying Sun, Jing Liu, Xiaohong Chen, and Dehui Du. Improving testing coverage for safety-critical system by mutated specification. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 1, pages 43–46. IEEE, 2014.

[ZWT12] Zhao Zhang, Qiao-Yan Wen, and Wen Tang. An efficient mutation-based fuzz testing approach for detecting flaws of network protocol. In *2012 International Conference on Computer Science and Service System*, pages 814–817. IEEE, 2012.