Universität Bremen
Fachbereich 3: Mathematik und Informatik
Arbeitsgruppe Betriebssysteme und Verteilte Systeme
Leiter: Prof. Dr. Jan Peleska

Model-Based Scenario Testing and Model Checking with Applications in the Railway Domain

Dissertation
zur Erlangung des Doktorgrades
Doktor der Ingenieurwissenschaften
— Dr.-Ing. —

Helge Löding
Bremen im September 2014

1. Gutachter: Prof. Dr. Jan Peleska
2. Gutachter: Prof. Dr.-Ing. Görschwin Fey

Datum des Promotionskolloquiums: 5. Februar 2015

Abstract

This thesis introduces Timed Moore Automata, a specification formalism, which extends the classical Moore Automata by adding the concept of abstract timers without concrete delay time values, which can be started and reset, and which can change their state from running to elapsed. The formalism is used in real-world railway domain applications, and algorithms for the automated test data generation and explicit model checking of Timed Moore Automata models are presented.

In addition, this thesis deals with test data generation for larger scale test models using standardized modeling formalisms such as UML. An existing framework for the automated test data generation is presented, and its underlying work-flow is extended and modified in order to allow user interaction and guidance within the generation process. As opposed to specifying generation constraints for entire test scenarios, the modified work flow then allows for an iterative approach to elaborating and formalizing test generation goals.

Acknowledgment

I would like to thank my supervisor Jan Peleska for his guidance as well as for his patience. His unique brand of directing overall content, of giving creative leeway and of petitioning me for palpable results has ensured, that this thesis would eventually come to be. As it stands, this thesis is the latest milestone in my professional career, which Jan Peleska has influenced as teacher, mentor and employer in its entirety.

Within this same vein, I owe thanks to the GESy graduate college for embedded systems, the Siemens railway automation graduate school and its respective members for constructive and insightful discussions during their combined workshops.

I would also like to thank the Siemens AG for their financial support. Their doctoral candidate scholarship has started me and this thesis on the right foot.

At Verified Systems Intl. GmbH, my work assignments and my colleagues in particular have helped me to gain valuable experience in the testing of real-world applications, software engineering best-practices and software quality in general.

Huge gratitude is owed to my family and friends, who have supported me throughout.

Contents

Chapter 1

Introduction

This thesis elaborates and discusses different aspects and real-world challenges of model-based software testing on different scales and test integration levels. A domain-specific specification formalism used in the railway domain is introduced and defined. Algorithms for test data generation and explicit model checking are presented.

Additionally, an existing framework for fully automated model-based test data generation is introduced and analyzed. Thereupon the framework is extended to accommodate a more interactive approach to scenario testing.

1.1 Overview

This chapter introduces the scope and context of the thesis. Specifically, section 1.2 explains the motivation for this work. In section 1.3, the main contribution of this thesis is discussed. Section 1.4 gives an overview over related research being done by others.

Chapter 2 introduces a real-world domain-specific formalism used for modeling safety-critical control systems in the railway domain. Algorithms used for test data generation and model checking of specifications using this formalism are presented.

Chapter 3 initially introduces a framework used for the automated generation of test data on the basis of test models specified using standardized specification formalisms. The framework is then extended to allow interactive user-guidance of the generation process.

Finally, chapter 4 summarizes results and gives an outlook over possible future research topics and tool expansions.

1.2 Motivation

Safety-critical systems are systems, which – upon failure –may cause human injury or death, loss of or catastrophic damage to equipment, or severe harm to the environment. As such, they are usually subject to rigid norms and regulations regarding their development and deployment.

Norms such as the RTCA DO-178C ([oR11]) for the aviation domain and the CENELEC EN-50128 ([Cen11]) for the railway domain prescribe software development processes, which are specifically tailored to ensure high software quality. While software quality attributes like functionality, availability, reliability, maintainability and efficiency certainly benefit from a well-defined development process, safety will usually be the main characteristic under consideration during the certification phase of a system for real-world deployment.

Within the domain of regulated safety-critical systems, all utilized software development processes, may they be of the waterfall-, V-, W- or iteration-based categories, will mandate some verification and validation activities for their development artifacts. Within this context, validation ensures, that each refinement step of a given development artifact into a more concrete form is a valid specialization. For example, given a system architecture document, all sub-system architecture documents must adhere to the overall architecture set forth in the source system architecture document. As such, validation is meant to ensure, that we are building the *right system.*

Conversely, verification is the process of showing, that we are building the *system right.* This entails providing sufficient proof to show, that the system under development is a correct implementation conforming to all specifications identified in all validated system specification documents.

Typically, each refinement of a system design into a more concrete specification will give rise to validation activities to ensure conformance of the specialization to the source abstraction. Additionally, each layer of abstraction will usually give rise to verification activities to ensure conformance between specification and implementation. System-, sub-system and module implementations will have to be evaluated against their respective system-, sub-system and module-specifications.

Verification methodology can roughly be split into static analysis and dynamic testing techniques, which evaluate system source code and system instances being executed respectively. While static approaches generally attempt to construct mathematical proofs of relevant properties on the basis of source code (or the respective development process artifacts of the abstraction layer under consideration), dynamic testing (which will simply be called testing from here on) will systematically stimulate a running instance of a system under test and evaluate the observed system responses against expected (specified) behavior.

It is not surprising, that the precision of all development, verification and validation activities heavily depend on the level of formalization of referenced or evaluated documents. However, a high level of formalization may be detrimental to maintainability, especially when using specification formalisms, which themselves lack readability. As a resulting trade-off, graphical specification formalisms with an underlying formal semantics have become the de-facto standard for specification purposes, and a range of computer-aided software engineering (CASE) tools, which support various graphical specification formalisms for modeling architecture and behavior on all system abstraction levels, exist.

The usefulness of formal models in the context of system verification is manifold. Within static analysis, models may be used as inputs for theorem provers, model checkers or other static analysis tools. For testing purposes, test models can typically serve two separate functions: They can be used to systematically generate stimulations to be applied to the system under test, and they can be used to assess the correctness of the observable system under test reactions to those stimulations.

Note, however, that care needs to be taken when selecting reference system models. Whenever, for example, the same model is used as a basis for generating source code and generating tests for that source code, the redundancy between system under test and tests for that system is lost. In such a scenario, not the system under test, but only the source code and test generation process is being evaluated.

Another trade-off to be considered is between standardized modeling formalisms and domain- or even application specific (graphical) languages. On the one hand, standardized formalisms such as the OMG UML ([OMG11a], [OMG11b]) provide common specification grounds for developers as well as verification and validation specialists, and a wide array of tools are available.

On the other hand, since standardized formalisms need to be broad and feature-rich enough for a multitude of applications and domains, their formal semantics are - if even available - usually very complex and difficult to deal with for model checkers and test data generators. In contrast, domain-specific languages allow practitioners to tailor specification formalisms specifically to their needs. As a result, the corresponding formal semantics will contain only the complexity needed by its application.

Note, that efforts such as UML2 Profiles are made to bridge this gap by providing facilities to restrict, customize and constrain features of generic specification formalisms. However, just as with domain-specific languages, practitioners still need to annotate profiles with formal semantics suitable for their application domain. As such, profiles are in large parts simply a more standardized way to create domain-specific languages.

This thesis elaborates two separate and very different model-based verification approaches, one dealing with a domain-specific specification formalism used in the railway domain, the other dealing with extensions made to a model-based testing framework used in conjunction with standardized specification formalisms.

Within the first part of the thesis, railway level crossing system component specifications using a domain-specific graphical notation and the corresponding system component implementations are given. Since all components are implemented separately, the graphical system component specifications can serve as test models.

It is assumed, that conformance of the specifications to functional requirements have already been validated in earlier phases. As such, the verification task at hand is to ensure conformance of the implementations to their respective specifications. The system components are considered to be white boxes, and structural coverage criteria are used to determine test end conditions.

Additionally, the domain-specific specification formalism used here lends itself to performing explicit model checking. The corresponding algorithms are elaborated and used to ensure some basic liveness properties.

In contrast, the second part of this thesis is concerned with test data generation for test models using standardized specification formalisms. Here, the scope lies on a higher test integration level. Entire (sub-)systems are considered in a black-box testing context. The focus lies on how larger test models may be utilized to generate test data for scenarios reflecting functional requirements.

1.3 Main Contribution

Within the first part of this thesis, a new graphical specification formalism used for modeling embedded controllers in the railway domain is introduced and elaborated. While this specification formalism called Timed Moore Automata has already been heavily used to specify expected behavior of real-world applications, verification of systems built from Timed Moore Automata specifications has previously been based on manually deriving test cases for the resulting source code. The thesis presents approaches and algorithms for performing model-checking and test data generation directly from test models specified using Timed Moore Automata.

Moreover, the presented specification formalism may be of general scientific interest. Timed Moore Automata introduce the concept of abstract timers. Abstract timers are not associated with concrete time durations, after which they must elapse. Rather, they provide the abstract notions of timer statuses *timer running* and *timer elapsed* as well as timer actions *timer start* and *timer stop* and the general semantic rules, under which timer actions and timer statuses may influence each other. As such, Timed Moore Automata may be useful as a further abstraction level between entirely timeless specification formalisms such as classical Moore Automata ([Moo56]) and timed formalisms based on concrete timer durations, such as the region graphs derived from Timed Automata ([AD94]).

In order to facilitate model-based scenario testing, the second part of this thesis builds upon an existing framework used for model-based test generation from larger scale test models. The framework can accommodate multiple standardized modeling formalisms, and for any test model, the test data generation process follows a generic and completely automated work-flow. The thesis discusses some weaknesses inherent in this paradigm and introduces a new interactive generation work-flow, which allows the user to visualize and influence each step of the generation process.

As such, the interactive test generation paradigm realized in this thesis may serve as an exemplary prototype, which allows a user to inject application expertise into the model-based test generation process while still harvesting the power of fully automated test generation approaches. As a result, automated model-based test generation approaches can become valuable tools for developing test cases as well as test data intended for scenario testing.

1.4 Related Work

Within this section, research related to the topics of this thesis being done by other researchers and groups is presented.

The following subsections list a number publications, which give overviews of different scope and detailing over the field of model-based testing. The most common tools and frameworks – including the framework used within this thesis – are enumerated and briefly summarized.

Furthermore, publications introducing the fundamental concepts of model checking and the model checking tools considered to be state of the art are given. Papers on various approaches regarding the interaction and interdependence between model checking and model-based testing are layed out additionally.

In a later subsection, the research landscape concerning constraint solving algorithms and corresponding solver implementations is sketched. Notably, this includes the constraint solver used at the heart of the model-based test generation framework used in the thesis.

Finally, notable papers from other sub-domains of model-based testing not immediately related to this thesis and the used model-based testing approach are briefly enumerated.

1.4.1 Model-Based Testing

As a position paper on model-based testing, [Utt08] gives a comprehensive summary over the field of model-based testing.

[Tre11] expands on this and gives a more formal overview for model-based testing in general, and for testing labeled transition systems in particular.

Another introduction into model-based testing with emphasis on the processes and characteristics of different approaches to model-based testing can be found in [UPL12].

In [Bel10], the author introduces JTorX, a testing tool for model-based test generation and execution. The tool compares a given labeled transition system specification to an implementation using the "ioco" implementation relation. The corresponding testing theory is set forth in [Tre08].

The MOTES tool [EKRV06] generates test date for extended finite state machines using the model checker UPPAAL Cora ([BLR05]) and various structural coverage criteria.

[DBI12] proposes another extension of finite state machine called "Stream X-machine" (SXM), where state machines can be annotated with data structures and functions to operate on that data. [DBI12] presents a testing theory and a corresponding testing tool JSXM for testing Stream X-machines.

The MaTeLo tool [DZ03] derives test cases in the TTCN-3 notation ([Din04]) from Markov chain usage models (MCUM) [Pro05] to perform model-based statistical testing. While MaTeLo automatically derives its MCUMs from different formal model descriptions, the JUMBL tool [Pro03] can be used to directly model MCUMs and generate test cases from it.

UPPAAL Cover/TRON [HLM+08] utilizes test models formulated as timed automata ([AD94]) and coverage criteria formulated in an observer language to generate test cases in the form of timed input sequences.

Using the UML2 Testing Profile ([BJ07]), the TTmodeler tool [PSK08] generates test cases in the TTCN-3 notation. Generation goals are modeled directly into the test model.

The Conformiq Qtronic tool [Hui07] accepts UML test models and yields TTCN-3 test cases as well. Additionally, Qtronic introduces the proprietary Qtronic Modeling Language (QML).

The Smartesting CertifyIt tool [BGLP08] is yet another test generator based upon UML. It allows usage of OCL constraints ([Obj10]) within the test model to guide the generation process.

The RT-Tester Model-Based Testing Extension [Pel13] utilizes an (internal) intermediate modeling language to incorporate multiple test model specification formalisms and their respective front-ends. Using the semantics of

Harel's state charts ([HN96]), test data can be generated for multiple test notations using corresponding back-ends. As a major prerequisite of this thesis, it is described in more detail in section 3.1.

Case studies for testing embedded systems using the above framework are given in [EP11] and [PHL+11] for the avionics and automotive domain respectively. The usage of the framework as part of model-driven software verification of synchronous components is published in [MGP+12].

Handling of aliasing problems within automated test generation based on control flow graphs is introduced in [LP08], and the further combination of these code-based test generation algorithms with static analysis methods is presented in [PLK07].

A model-based equivalence class testing strategy for test models using SysML semantics ([Hau06]) is presented in [HP13].

Transformations based on case grammar theory ([Coo89]) of controlled natural language requirements initially into an intermediate test model, and eventually into executable test cases is shown in [CBL+14].

A multitude of considerations and aspects of combining model-based testing with scenario testing, and of interactively injecting domain expertise into automated test generation processes can be found in [RK11], [AQ13], [DKT08], [DCT12], [CDJ11], [MLL09], [LK01], [CDKM11], [BW05], [GKP00] and [MFT12]. The evaluation section 3.5 of chapter 3 on interactive model-based testing summarizes these publications in more detail.

1.4.2 Model Checking

Foundations of model checking are comprehensively presented in [CGP99]. It introduces Kripke structures as basic data structures, temporal logics to express properties to be checked as well as the algorithms used.

Notable model checking tools are SPIN ([Hol03]), SMV and NuSMV ([CCGR00]), UPPAAL ([BDL+06]) and the Java Pathfinder ([HP00]).

An adaption of Tarjan's algorithm [Tar71] for the computation of strongly connected components - an algorithm at the heart of model checking - is used in [JM99] to perform test data generation.

Automatic test case generation for state charts using the CTL temporal logic to formalize coverage criteria and the SMV model checker to construct reachability (counter-)witnesses is described in [HLSC01].

[CSE96] describes the use of counter-witness extracted from model checking to generate test cases for requirements specified in the LTL temporal logic.

As opposed to finding witnesses for reachability properties and transforming these into test cases, [ABM98] performs model mutations and uses model checking to construct witnesses, which distinguish the mutation from the original model.

[AB00] assess utilizing model checking for test generation using the MC/DC ([CM94]) coverage criterion. Additionally, they argue for the use of model checking for test set recognition.

In [AADO00], the authors elaborate, how model checking might be used to achieve specification-mutation coverage, full predicate coverage and transition-pair coverage.

In [FW08], a notion of property relevance of test cases is introduced in order to evaluate test suites against their ability to detect requirements violations.

In [Eng05], the authors evaluate trade-offs between model checking and static analysis when finding errors, particularly in file system code.

1.4.3 Constraint Solving

In constraint solving, the values of a set of variables are restricted using a set of constraints (equality, inequality, affiliation with a certain value domain, etc.). [RBW06] gives an overview over the field of constraint programming.

Integer programming and solving approaches such as the well-known simplex algorithm to solve problem instances initially known from operations research are presented in [Rav07].

Boolean satisfiability (SAT) solvers "have become the key enabling technology in automated verification" ([BHvMW09]). Biere et al. present a collection of papers on the theoretical and practical impact of SAT solving specifically on (bounded) model checking and program verification. In [DEFT09], this is expanded on to show, how SAT solvers can be utilized for automatic test pattern generation for hardware circuit verification.

Combining multiple solvers capable of handling different theories (e.g. integer programming, arithmetic, bit-vectors, etc.) with a SAT solver used to coordinate the (sub-)theory-solvers yields Satisfiability-Modulo-Theory (SMT) solvers, which are capable of solving sets of mixed constraints. [BSST09] takes a look at how to construct such SMT solvers.

The Satisfiability Modulo Theory Library (SMT-LIB) [BRST08] maintains a list of state-of the art SMT solvers and their references. These currently include: Alt-Ergo, Boolector, CVC4, MathSAT 5, MiniSmt, Mistral, SMT-Interpol, SONOLAR, UCLID, veriT, Yices 2, Z3.

The SONOLAR solver ([PVL11]) in particular is used as an integral part of the model-based testing framework [Pel13] expanded on within this thesis.

In addition to [Pel13], [AS05], [CIvdPS05] and [GMS98] all utilize constraint solving for model bases test generation, albeit with different intentions.

1.4.4 Other Test Generation Techniques

Search-based test generation utilizes meta-heuristic search approaches such as genetic algorithms, simulated annealing or other probabilistic algorithms to solve the test generation problem. A selection of such approaches can be found in [ATF09], [HHL$^+$07], [LHM08], [MS04] or [McM04].

In random testing, large numbers of test cases are created with little effort and little regard for the quality of single test cases. Rather, the sheer amount of test cases and their statistical distribution is assumed to detect a majority of faults. Among other considerations, [Pre06], ,[ODC06], [CLOM06] and [CLOM07] discuss the efficiency of random testing with regard to strength of resulting test suites and their cost-effectiveness.

Evolutionary testing – possibly as an extension of random testing – refines given test input data using mutations and fitness functions. Several approaches to evolutionary test generation can be found in [KG04], [HM07], [WS07], [Weg03], [WL05] or [WB04].

Chapter 2

Timed Moore Automata

This chapter introduces Timed Moore Automata, a real-world formalism used for modeling safety-critical control systems in the railway domain. In order to provide software verification and validation tool support for control systems modeled as Timed Moore Automata, algorithms for the automated generation of test data and the validation of required application properties were developed. While some results are already known from [LP10], this thesis greatly expands on and illuminates the material presented there.

Section 2.1 reviews the classical Moore Automata as introduced in [Moo56] with a special focus on applying the formalism to the specification of control systems. Section 2.2 extends classical Moore Automata to introduce Timed Moore Automata, a formalism, which introduces the notion of timers to Moore Automata.

In section 2.3 we describe a model checking approach used in checking instances of Timed Moore Automata specifically for live-locks. Section 2.4 introduces specialized algorithms used to generate test data for Timed Moore Automata. Section 2.5 presents achieved results for a real-world railway application.

2.1 Classical Moore Automata

This section re-introduces the classical Moore Automata as invented in [Moo56]. While they are usually considered to be a computation model for recognizing regular expression languages, they can be viewed as a simple modeling formalism for specifying control systems.

Moore automata are simple finite-state machines (deterministic or not). Their defining characteristic is, that all outputs are determined only by the state they are in.

Within this thesis, we will ignore the notion of final (or accepting) states as introduced by Moore. Final states are significant when recognizing a word as belonging to a language. However, as this thesis has a bias towards controller specifications, we will ignore this since controllers are usually (conceptually) meant to run infinitely long and never terminate.

This section will first introduce the abstract and concrete syntax of classical Moore Automata. The subsection on static semantics will then introduce some additional constraints on well-defined automata. The following subsections will then introduce the operational semantics of classical Moore Automata and consider the notion of determinism for them.

2.1.1 Abstract Syntax

In order to formally model the syntax of a given Moore Automaton, we need to define a mathematical structure, which can then contain all information from within the concrete (graphical) representation of an automaton.

Here, we define a tuple, which will contain locations, variable symbols, location transitions, guard conditions and entry actions. An instance of such a tuple will serve as a basis for defining the behavior of automata later.

Definition 1. *The abstract syntax of any given classical Moore Automata consists of the 6-tuple:*

$$(LOC, loc_0, VAR_{in}, VAR_{out}, L, R)$$

The elements of this tuple are given as:

(1) LOC indicates the set of locations within the automaton.

(2) $loc_0 \in LOC$ denotes the initial location, which will be assumed upon start-up of the automaton.

(3) VAR_{in} *defines the input alphabet of the automaton. As such, it is formalized as a set of input variable symbols.*

(4) VAR_{out} *denotes the output alphabet of the automaton. It is again defined as a set of output variable symbols.*

(5) $L : LOC \longrightarrow VAR_{out}$ *is a total labeling function, which assigns one output variable symbol to each location.*

(6) $R : LOC \times VAR_{in} \times LOC$ *is a location transition relation, which relates predecessor locations and input variable symbols to successor locations.*

Note, that in contrast to the description of classical Moore Automata from [Moo56], we do not reference or define final/accepting locations. This is again due to the fact, that this study is focused mainly on controller implementations, which are – in concept – executed indefinitely.

2.1.2 Concrete Syntax

In order to illuminate the abstract syntax of classical Moore Automata, consider the graph representation of a Moore Automata from figure 2.1.

The set of locations LOC can be found simply by collecting the names of locations drawn in the graph. It is hence defined as the set:

$$LOC = \{S_1, S_2, S_3\}$$

The initial location loc_0 is designated by the arrow without successor state. In the given automaton, this means:

$$loc_0 = S_1$$

The sets of input- and output variable symbols VAR_{in} and VAR_{out} can be found by collecting location entry actions and transition labels respectively:

$$VAR_{in} = \{a, b, c\}$$
$$VAR_{out} = \{X, Y, Z\}$$

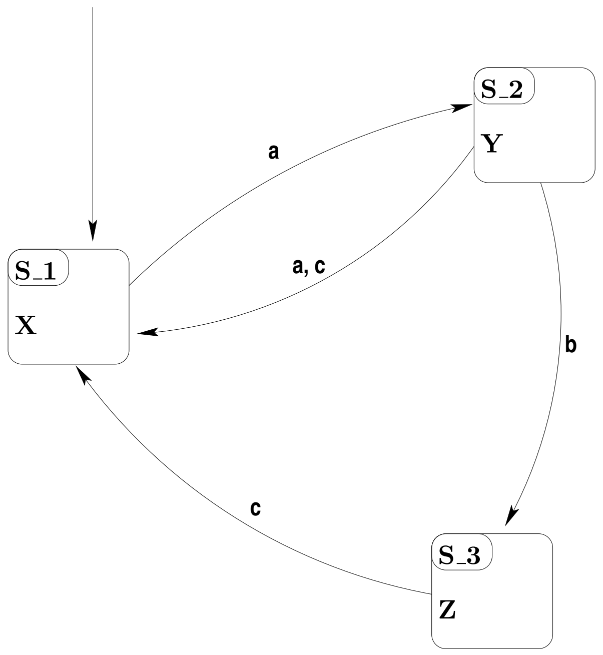


Figure 2.1: Moore Automaton Example

The labeling of locations with output variable symbols L can be constructed by collecting all location entry actions individually. For the given automaton, the result is:

$$L = \{S_1 \mapsto X, S_2 \mapsto Y, S_3 \mapsto Z\}$$

The location transition relation R can be inferred from each arrow within the graph. Each arrow specifies a predecessor and a successor location as well as an input variable symbol associated with the respective arrow's location transition.

Note, that multiple input variable symbols associated with a single arrow are merely used as a shorthand for multiple arrows with identical predecessor and successor locations. Within the given automaton, the arrow labeled a, c thus stands for two separate location transitions.

The location transition relation from the given example looks as follows:

$$\begin{aligned} R = \{ \quad & (S_1, a, S_2), \\ & (S_2, a, S_1), (S_2, b, S_3), (S_2, c, S_1), \\ & (S_3, c, S_1) \qquad\qquad\qquad\qquad \} \end{aligned}$$

2.1.3 Static Semantics

In order for a given Moore Automaton $(LOC, loc_0, VAR_{in}, VAR_{out}, L, R)$ to be well defined, some constraints must hold for the 6-tuple:

Firstly, only a finite number of locations are allowed. Secondly, all symbolic sets must be pairwise disjoint, i.e. no symbol may appear in more than one of the sets LOC, VAR_{in} and VAR_{out}.

Definition 2. *A classical Moore Automaton*

$$(LOC, loc_0, VAR_{in}, VAR_{out}, L, R)$$

is well defined, iff the following holds:

$$\begin{array}{ll} \mid LOC \mid < \ \infty & \wedge \\ LOC \cap VAR_{in} = \emptyset & \wedge \\ LOC \cap VAR_{out} = \emptyset & \wedge \\ VAR_{in} \cap VAR_{out} = \emptyset & \end{array}$$

2.1.4 Operational Semantics

In order to define the behavior of an implementation of any given classical Moore Automaton, we consider the state-space S of an execution of the automaton. This state space is the conceptual set of execution states, which an automaton implementation must differentiate in order to provide a specific behavior.

Furthermore, for each predecessor execution state $s \in S$ an automaton implementation must consider, which states $s' \in S$ are possible successor execution states. An automaton implementation is then characterized by providing a definite state transition relation T, which relates all predecessor and successor states for any given automaton.

The intended behavior of Moore Automata is then specified as a triple:

$$(S, s_0, T)$$

Within this triple, S is the state space, s_0 is the initial execution state and T is the state transition relation.

Classical Moore Automata derive their importance in computer science precisely from the fact, that they are a class of automata, for which only the set of locations needs to be considered as state space. This is more commonly described as the property, that outputs of Moore Automata only depend on the location the automaton is in. The state space S is therefore defined as:

$$S = LOC$$

For each execution state $s \in S$, the visible output of the automaton is immediately defined as $L(s)$ using the location labeling function L from definition 1.

The initial execution state s_0 is trivial and can be taken directly from the syntax of a given automaton:

$$s_0 = l_0$$

The state transition relation T for classical Moore Automata must relate pairs of states and variable input symbols to successor states. As such, it is of the form:

$$T : S \times VAR_{in} \times S$$

The state transition relation T needs to contain all triples $(s, v_{in}, s') \in S \times VAR_{in} \times S$, which are already part of location transition labeling relation R from definition 1. Additionally, if for a given input a location has no matching emanating transition in R, the corresponding execution state must transition to itself.

We can now define the operational semantics for a classical Moore Automaton:

Definition 3. *The operational semantics of a classical Moore Automaton* $(LOC, loc_0, VAR_{in}, VAR_{out}, L, R)$ *is defined as triple:*

$$(S, s_0, T)$$

The state space S *is defined as:*

$$S = LOC$$

The initial execution state s_0 *is defined as:*

$$s_0 = l_0$$

The state transition relation T *is defined as:*

$$T : S \times VAR_{in} \times S$$

$$\begin{aligned} T = \{ \quad & (s, v_{in}, s') \in S \times VAR_{in} \times S \mid \\ & \quad ((s, v_{in}, s') \in R) \lor \\ & \quad\quad ((\nexists(\overline{s}, \overline{v_{in}}, \overline{s'}) \in R : (\overline{s} = s) \land (\overline{v_{in}} = v_{in})) \land \\ & \quad\quad (s = s')) \qquad\qquad\qquad \} \end{aligned}$$

For any system state $s \in S$*, the output can be calculated simply as* $L(s)$*.*

This definition is well formed in the sense, that each execution state will always have at least one successor state. In other words, the transition relation T is total.

Proof. Let $(LOC, loc_0, VAR_{in}, VAR_{out}, L, R)$ be any well defined classical Moore Automaton.

Suppose, there exists an execution state $\overline{s} \in S$, such that $\overline{s}$ has no successor state. Formally, suppose:

$$\exists \overline{s} \in S : \nexists (s, v_{in}, s') \in T : \overline{s} = s$$

Consider $\overline{s}$ and any input variable symbol $\overline{v} \in V_{in}$.

Case 1 – Existing location transition label:

Suppose:

$$\exists (s, v_{in}, s') \in R : (\overline{s} = s) \wedge (\overline{v} = v_{in})$$

Then:

$$(\overline{s}, \overline{v}, s') \in R$$

And by definition of T:

$$(\overline{s}, \overline{v}, s') \in T$$

This is a counter-example, since apparently:

$$\exists (s, v_{in}, s') \in T : s = \overline{s}$$

Case 2 – Non-existing location transition label:

Suppose:

$$\nexists (s, v_{in}, s') \in R : (\overline{s} = s) \wedge (\overline{v} = v_{in})$$

We can augment this:

$$\begin{aligned}&(\nexists (s, v_{in}, s') \in R : (\overline{s} = s) \wedge (\overline{v} = v_{in})) \wedge \\ &\quad (\overline{s} = \overline{s})\end{aligned}$$

Then, by definition of T:

$$(\overline{s}, \overline{v}, \overline{s}) \in T$$

This is a counter-example, since apparently:

$$\exists (s, v_{in}, s') \in T : s = \overline{s}$$

$\square$

2.1.5 Determinism

In 2.1.4 it was proven, that the operational semantics for classical Moore Automata is well formed, i.e. does not cause any automaton execution to fail due to a dead-lock. This was done by showing, that each execution state of any automaton always has *at least* one successor state.

Additionally a classical Moore Automaton is considered to be deterministic, if each execution state has *exactly* one successor state. This can be achieved by restricting the location transition labeling relation R from definition 1.

Definition 4. *A classical Moore Automaton*

$$(LOC, loc_0, VAR_{in}, VAR_{out}, L, R)$$

is deterministic, if the following constraint holds for location transition labeling function R:

$$\nexists (s_1, v_1, s_1'), (s_2, v_2, s_2') \in R : (s_1 = s_2) \wedge (v_1 = v_2) \wedge (s_1' \neq s_2')$$

This restriction means, that R is now in fact a partial function:

$$R : LOC \times VAR_{in} \not\longrightarrow LOC$$

Accordingly, the state transition relation T then becomes a total function:

$$T : S \times VAR_{in} \longrightarrow S$$

Proof. In order to show, that state transition relation T for deterministic Moore Automata is a total state transition function, it needs to be shown, that:

$$\forall \overline{s} \in S, \overline{v} \in V_{in} : \mid \{(s, v_{in}, s') \in T \mid (\overline{s} = s) \wedge (\overline{v} = v_{in})\} \mid = 1$$

Case 1 – Existing location transition label:

Suppose:

$$\exists (s, v_{in}, s') \in R : (\overline{s} = s) \wedge (\overline{v} = v_{in})$$

Since R is now a partial function, this means:

$$\exists s' \in S : R(\overline{s}, \overline{v}) = s'$$

The successor state s' must be unique, again because R is a partial function.

Case 2 – Non-existing location transition label:

Suppose:

$$\nexists (s, v_{in}, s') \in R : (\overline{s} = s) \wedge (\overline{v} = v_{in})$$

Since R is now a partial function, this means:

$$\nexists s' \in S : R(\overline{s}, \overline{v}) = s'$$

As seen in the proof from 2.1.4, the state transition $(\overline{s}, \overline{v}, \overline{s})$ is an element of T. Additionally, the state transition relation T from definition 3 ensures this to be the only element of the form $(\overline{s}, \overline{v}, s')$, since T explicitly enforces $(\overline{s} = s')$.

□

2.2 Abstract Timing for Moore Automata

This section introduces Timed Moore Automata. After an informal introduction, we discuss the extensions to classical Moore Automata needed to introduce the abstract notion of timers.

As such, this section is organized largely in analogy to the previous discussion of classical Moore Automata. Again, we consider abstract and concrete semantics, static and operational semantics and determinism.

2.2.1 Informal Introduction to Timed Moore Automata

Timed Moore Automata are an extension of the classical Moore Automata in the sense, that they preserve the fundamental Moore Automata property: Outputs of an automaton are still only dependent on the current location of the automaton. Apart from this, several modifications and enhancements have been introduces in the Timed Moore Automata formalism in order to get closer to a practical embedded controller specification formalism.

While real-world controllers may internally defer computations to some synchronous processing mechanisms, it is uncommon for external controller input interfaces to queue inputs as events to be processed since this might impede reaction time requirements.

Consequently, outputs are commonly published collectively at the external output interface of a controller for all who might be concerned. Again, producing outputs as a sequence of events is uncommon for external controller output interfaces since this might place too many restrictions on receivers as well as senders.

Classical Moore Automata processed their inputs and outputs in a synchronous fashion as queues of events. In contrast, Timed Moore Automata view inputs and outputs as vectors of signals with Boolean values to be processed in an asynchronous fashion. As such, they execute in a run-to-completion mode before accepting new inputs. As compared to the classical Moore Automata, which would accept a new input after each discrete transition, Timed Moore Automata perform discrete Transitions for as long as they are enabled and only accept new inputs when no more transitions are enabled.

The Timed Moore Automaton formalism is designed for modeling controllers as collections of automata running in parallel. As such, they could well have been designed to employ a synchronous execution semantics. However, the application domain necessitates, that collections of interacting automata can run in different processes or even on different computation nodes. As such, an asynchronous approach is more suited.

Safety-critical controllers are regularly required to behave deterministically with respect to any reproducible sequence of inputs. This can pose a modeling hazard when formulating guard conditions for a multitude of transitions leaving a specific location. In such a situation, it is not always trivial to see, whether all transition guards are pairwise preclusive. As a concession to model developers, Timed Moore Automata introduce the notion of unique transition priorities.

As the name might suggest, Timed Moore Automata introduce an abstract concept of timers. These timers are abstract in the sense, that no specific timer elapse time spans are ever specified. Instead, Timed Moore Automata only employ notions of timer actions and timer statuses. Timers may be *(re)started* or *stopped*, and they may be in a *running* or *elapsed* state. While timer actions *start* and *stop* are unremarkable when compared to other modeling formalisms, the notion of a timer *elapse* as a non-deterministic input free of any concept of time span makes this formalism special.

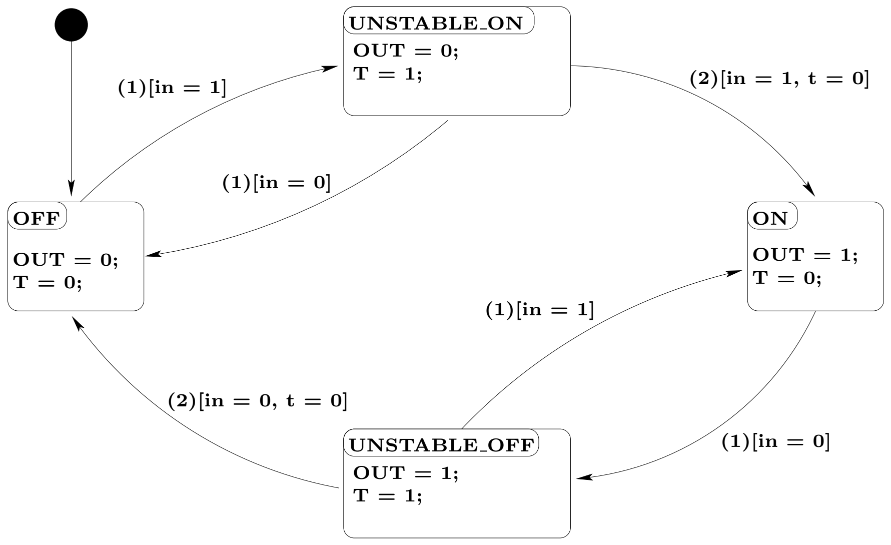


Figure 2.2: Timed Moore automaton for input debouncing

Timer actions are specified as entry action assignments to timer action variables. Starting a timer is done by assigning a value of 1 to its variable, an assignment of 0 stops the timer. Conversely, timer statuses can be evaluated using timer status variables. Here, a value of 1 denotes a running timer, while a value of 0 denotes an elapsed timer.

Consider the Timed Moore Automaton from figure 2.2, which specifies a simple input signal debouncing automaton. In locations ON and OFF, the input in is stably true or false, respectively. The output OUT reflects this in each location.

Whenever the input changes value, the automaton transitions to location $UNSTABLE\_OFF$ or $UNSTABLE\_ON$ respectively. Here, the timer T is started. Should the input revert back to its original value, the automaton transitions back to its previous location ON or OFF. However, should timer T elapse while the input remains different from the output, the automaton transitions to new stable output location OFF or ON respectively.

Note, that no timer elapse time span is specified for timer T and timer elapse variable t. Rather, t is a special (non-deterministic) input to the automaton.

2.2.2 Abstract Syntax Extensions

As with classical Moore Automata, we need to define a mathematical structure to contain all relevant syntactical information for a given Timed Moore Automaton. This definition will largely resemble the earlier definition for the abstract syntax of classical Moore Automata from section 2.1.1 in the sense, that again locations, variable symbols, location transitions, guard conditions and entry actions are considered.

However, the definition of the abstract syntax for Timed Moore Automata will be extended to incorporate timer symbols and their appearance within guard conditions and entry actions.

Definition 5. *The abstract syntax of a Timed Moore Automaton consists of the 10-tuple:*

$$(LOC, loc_0, VAR_{in}, VAR_{out}, VAR_{ta}, VAR_{ts}, \beta, L_{out}, L_{ta}, R)$$

The elements of this tuple are now given as:

(1) LOC indicates the set of locations within the automaton.

(2) $loc_0 \in LOC$ denotes the initial location, which will be assumed upon start-up of the automaton.

(3) VAR_{in} defines the input alphabet of the automaton. As such, it is formalized as a set of input variable symbols.

(4) VAR_{out} denotes the output alphabet of the automaton. It is again defined as a set of output variable symbols.

(5) VAR_{ta} defines the set of timer activation symbols.

(6) VAR_{ts} denotes the corresponding set of timer status symbols.

(7) $\beta : VAR_{ta} \longleftrightarrow VAR_{ts}$ is a bijection mapping timer activation variables and timer status variables to each other.

(8) $L_{out} : LOC \longrightarrow (VAR_{out} \longrightarrow \mathbb{B})$ is a labeling function, which for each location assigns Boolean valuations to each output variable.

(9) $L_{ta} : LOC \longrightarrow (VAR_{ta} \not\longrightarrow \mathbb{B})$ *is a labeling function, which for each location may assign Boolean valuations to some timer activation variables.*

(10) $R : LOC \longrightarrow (\mathbb{N} \not\longrightarrow ((VAR_{in} \cup VAR_{ts}) \not\longrightarrow \mathbb{B}) \times LOC)$ *is the location transition relation, which relates predecessor and successor locations. More precisely, each predecessor location is mapped to a partial function, which maps transition priorities to pairs of corresponding guard conditions and successor locations. The guard conditions themselves are partial functions, which map input- and timer status variables to their required transition guard values.*

2.2.3 Concrete Syntax Extensions

The concrete syntax of Timed Moore Automata shows some key differences when compared to the concrete syntax of classical Moore Automata. Again consider the Timed Moore Automaton from figure 2.2.

As compared to classical Moore Automata, Timed Moore Automata locations do not merely list output symbols (and timer activation symbols), but assign them Boolean values. Consequently, transitions are guarded not only by input symbols (and timer status symbols), but are guarded by specific Boolean valuations.

This is due to the fact, that inputs and outputs are now processed as vectors of Boolean values rather than as input and output events. Another manifestation of this is the artificial initial location ●, which each Timed Automaton must furnish in order to provide a defined output state prior to automaton execution.

It is worth noting, that since Timed Moore Automata are extensions of the classical Moore Automata, each location must correspond to an entire output vector. It is therefore assumed, that all output variables not explicitly listed within a location shall have their values set to $false$. Note, that this is explicitly **not** true for timer activation variables, as timers may retain their status across multiple location transitions.

Timed Moore Automata provide transition priorities for all location transitions. For transitions emanating from a specific location, these are unique (usually consecutive) natural numbers intended to enforce deterministic modeling. Here, lower numbers mean higher transition priorities.

Finally, a subtle naming convention distinguishes input, output, timer activation and timer status variables. Input variables and timer status variables are always given in lower case. Output variables and timer activation variables are always given in upper case. Furthermore, timer activation variables will always begin with the letter 'T'. Their associated timer status variables will share the respective variable name, but be in lower case.

For the given automaton, the resulting abstract syntax instance looks as follows:

The set of locations is given as:

$$LOC = \{loc_0, ON, OFF, UNSTABLE\_ON, UNSTABLE\_OFF\}$$

Not surprisingly, the initial location of the automaton is the artificial initial location loc_0.

The variable symbol sets are:

$$\begin{array}{l} VAR_{in} = \{in\} \\ VAR_{out} = \{OUT\} \\ VAR_{ta} = \{T\} \\ VAR_{ts} = \{t\} \end{array}$$

The mapping between timer variables is then:

$$\beta = \{T \mapsto t\}$$

The location labeling functions look as follows:

$$\begin{array}{ll} L_{out} = \{ & loc_0 \mapsto \{OUT \mapsto 0\}, \\ & ON \mapsto \{OUT \mapsto 1\}, \\ & OFF \mapsto \{OUT \mapsto 0\}, \\ & UNSTABLE\_ON \mapsto \{OUT \mapsto 0\}, \\ & UNSTABLE\_OFF \mapsto \{OUT \mapsto 1\} \quad \} \end{array}$$

$$\begin{array}{ll} L_{ta} = \{ & loc_0 \mapsto \{T \mapsto 0\}, \\ & ON \mapsto \{T \mapsto 0\}, \\ & OFF \mapsto \{T \mapsto 0\}, \\ & UNSTABLE\_ON \mapsto \{T \mapsto 1\}, \\ & UNSTABLE\_OFF \mapsto \{T \mapsto 1\} \quad \} \end{array}$$

Finally, the location transition relation is given as:

$$\begin{array}{lll} R = \{ & loc_0 \mapsto \{ & 1 \mapsto (\emptyset, OFF)\} \\ & ON \mapsto \{ & 1 \mapsto (\{in \mapsto 0\}, UNSTABLE\_OFF)\} \\ & OFF \mapsto \{ & 1 \mapsto (\{in \mapsto 1\}, UNSTABLE\_ON)\} \\ & UNSTABLE\_ON \mapsto \{ & 1 \mapsto (\{in \mapsto 0\}, OFF), \\ & & 2 \mapsto (\{in \mapsto 1, t \mapsto 0\}, ON)\} \\ & UNSTABLE\_OFF \mapsto \{ & 1 \mapsto (\{in \mapsto 1\}, ON), \\ & & 2 \mapsto (\{in \mapsto 0, t \mapsto 0\}, OFF)\} \quad \} \end{array}$$

2.2.4 Static Semantics Extensions

As with classical Moore Automata, some constraints must hold in order for a given Timed Moore Automaton to be well defined.

In accordance to classical Moore Automata, the number of locations must be finite. Again, all symbolic variable sets must be pairwise disjoint.

As we now have an artificial initial location, some additional constraints must hold for it. The initial location must have a single emanating transition, which must be unguarded. No transitions may have the initial location as their target.

Finally, for each location all emanating transitions must be attributed with unique priorities.

Definition 6. *A Timed Moore Automaton*

$$(LOC, loc_0, VAR_{in}, VAR_{out}, VAR_{ta}, VAR_{ts}, \beta, L_{out}, L_{ta}, R)$$

is well defined, iff the following holds:

(1) The number of locations is finite:

$$\mid LOC \mid < \infty$$

(2) All symbolic sets are pairwise disjoint:

$$\begin{array}{ll}
LOC \cap VAR_{in} = \emptyset & \wedge \\
LOC \cap VAR_{out} = \emptyset & \wedge \\
LOC \cap VAR_{ta} = \emptyset & \wedge \\
LOC \cap VAR_{ts} = \emptyset & \wedge \\
VAR_{in} \cap VAR_{out} = \emptyset & \wedge \\
VAR_{in} \cap VAR_{ta} = \emptyset & \wedge \\
VAR_{in} \cap VAR_{ts} = \emptyset & \wedge \\
VAR_{out} \cap VAR_{ta} = \emptyset & \wedge \\
VAR_{out} \cap VAR_{ts} = \emptyset & \wedge \\
VAR_{ta} \cap VAR_{ts} = \emptyset &
\end{array}$$

(3) Only a single unguarded transition emanates from the initial location. It leads to another location.

$$\exists loc \in LOC : loc \neq loc_0 \wedge R(loc_0) = \{1 \mapsto (\emptyset, loc)\}$$

(4) No transition has the initial location as target

$$\begin{array}{l}
\nexists loc \in LOC, p \in \mathbb{N}, \gamma \in (VAR_{in} \cup VAR_{ts}) \not\longrightarrow \mathbb{B} : \\
\quad (loc \mapsto (p \mapsto (\gamma, loc_0))) \in R
\end{array}$$

Side note: For a given location the uniqueness of all emanating transition priorities is implicitly given, since prioritized transition labels are modeled as a (partial) function with priorities $\mathbb{N}$ as its domain.

2.2.5 Operational Semantics Extensions

As before, we need to consider the state space S of Timed Moore Automata in order to define their behavior. Using a suitable definition for the state space, we can then specify, which predecessor execution states transition into which successor execution states.

As with classical Moore Automata, the current location of an automaton is sufficient to determine its output values, and we do not need to encapsulate output valuations within the state space. The set of locations LOC is therefore again part of the system state.

Additionally, since Timed Moore Automata perform their calculations in a run-to-completion mode, we need to keep track of input valuations. We employ valuation functions $\Sigma_{in} : VAR_{in} \longrightarrow \mathbb{B}$ for this purpose.

Timer actions do not need to be kept within the state space, since they can be immediately inferred from the current location of a Timed Moore Automaton. However, timer statuses need to be considered, and we use Boolean valuation functions $\Sigma_{ts} : VAR_{ts} \longrightarrow \mathbb{B}$ for this.

In order to determine whether an automaton is performing discrete transitions, or whether it is in a stable state, in which it can accept new inputs, an artificial variable $\mathit{stable} \in \mathbb{B}$ is introduced into the state space.

As opposed to the classical Moore Automaton semantics, we need to consider multiple initial execution states $S_0 \subset S$. This is due to the fact, that all possible input valuations are valid prior to automaton execution. In contrast to this, no timers are running initially. As the automaton may have arbitrary input valuations, it is initially considered to be unstable ($\neg \mathit{stable}$).

The state transition relation $T : S \times S$ for Timed Moore Automata must consider three separate cases. Firstly, an automaton may perform a delay transition. Whenever an automaton is stable its location remains unchanged. New inputs may be assigned arbitrarily. Timers may elapse, but elapsed timers must remain so. All resulting execution states then become unstable in order to trigger a subsequent run to completion.

Secondly, an unstable state may perform a discrete transition. An automaton may change its current location if input- and timer status valuations enable a location transition. If multiple transitions are enabled, the automaton will perform the transition with the best priority. As a result, a successor execution state will be associated with the location transition's target location. While input valuations remain unchanged, the results of the target location's timer actions are reflected in successor state's timer status valuations. The successor execution state remains unstable, since additional location transitions may be possible.

Thirdly, an unstable automaton may become stable. This final discrete transition of a run to completion can only happen, if the automaton cannot perform any location transitions. The resulting execution state is identical to the predecessor state, except that it is stable.

Definition 7. *The operational semantics of a Timed Moore Automaton*

$$(LOC, loc_0, VAR_{in}, VAR_{out}, VAR_{ta}, VAR_{ts}, \beta, L_{out}, L_{ta}, R)$$

is defined as the triple:

$$(S, S_0, T)$$

The state space S is defined as:

$$S = LOC \times (VAR_{in} \longrightarrow \mathbb{B}) \times (VAR_{ts} \longrightarrow \mathbb{B}) \times \mathbb{B}$$

The set of initial states is given as:

$$\begin{aligned} S_0 = \{ \quad & (loc, \sigma_{in}, \sigma_{ts}, stable) \in S \mid \\ & \quad loc = loc_0 \wedge \\ & \quad \forall v \in VAR_{ts} : \neg \sigma_{ts}(v) \wedge \\ & \quad \neg stable \qquad \qquad \qquad \} \end{aligned}$$

For notation purposes we say, that a system state $s \in S$ models a location $loc' \in LOC$, if loc' is the current location of s. Let $s = (loc, \sigma_{in}, \sigma_{ts}, stable)$. We define:

$$\forall loc' \in LOC : s \models loc' \Leftrightarrow loc' = loc$$

In analogy for variable valuations σ_{in} and σ_{ts}:

$$\begin{aligned} & \forall v \in VAR_{in} : s \models v \Leftrightarrow \sigma_{in}(v) \\ & \forall v \in VAR_{ts} : s \models v \Leftrightarrow \sigma_{ts}(v) \\ & \forall stable' \in \mathbb{B} : s \models stable' \Leftrightarrow stable' = stable \end{aligned}$$

An execution state s models a location transition guard γ under the following conditions:

$$\begin{aligned} & \forall \gamma \in (VAR_{in} \cup VAR_{ts}) \not\longrightarrow \mathbb{B} : \\ & \quad (s \models \gamma \Leftrightarrow \\ & \qquad (\forall v \in VAR_{in} : v \notin dom(\gamma) \vee s \models \gamma(v)) \wedge \\ & \qquad (\forall v \in VAR_{ts} : v \notin dom(\gamma) \vee s \models \gamma(v))) \end{aligned}$$

The state transition relation T can now be defined as:

$$T : S \times S$$

$$
\begin{aligned}
T = \{(s,s') \in S \times S \mid \;& DISCTRANS_1(s,s') \wedge \\
& DISCTRANS_2(s,s') \wedge \\
& DELAYTRANS(s,s')\}
\end{aligned}
$$

The predicates $DISCTRANS_1$, $DISCTRANS_2$ and $DELAYTRANS$ are defined below.

Note, that Π_n is meant to denote the n-th element of a tuple.

(1) $DISCTRANS_1$: An unstable execution state with enabled location transitions performs a discrete transition to a new unstable successor state. For multiple enabled location transitions, the transition with best priority is taken. The new execution state contains the transition target location as well as the results of all timer entry actions. Inputs remain unchanged.

$$
\begin{array}{l}
DISCTRANS_1((loc, \sigma_{in}, \sigma_{ts}, stable), \\
\qquad\qquad\qquad (loc', \sigma'_{in}, \sigma'_{ts}, stable')) \\
\qquad\qquad =_{def} \\
(\exists p \in \mathbb{N} : (p \in dom(R(loc)) \wedge \\
\quad (loc, \sigma_{in}, \sigma_{ts}, stable) \models \Pi_1(R(loc)(p)) \wedge \\
\quad (\nexists p' \in \mathbb{N} : (p' \in dom(R(loc)) \wedge \\
\qquad (loc, \sigma_{in}, \sigma_{ts}, stable) \models \Pi_1(R(loc)(p')) \wedge \\
\qquad p' < p))) \wedge \\
\neg stable) \\
\qquad\qquad \Longrightarrow \\
(loc' = \Pi_2(R(loc)(p)) \wedge \\
\sigma'_{in} = \sigma_{in} \wedge \\
(\forall v \in VAR_{ta} : \\
\quad (v \notin dom(L_{ta}(\Pi_2(R(loc)(p)))) \wedge \sigma'_{ts}(\beta(v)) = \sigma_{ts}(\beta(v))) \vee \\
\quad (v \in dom(L_{ta}(\Pi_2(R(loc)(p)))) \wedge \sigma'_{ts}(\beta(v)) = L_{ta}(\Pi_2(R(loc)(p)))(v))) \wedge \\
\neg stable')
\end{array}
$$

(2) $DISCTRANS_2$: An unstable execution state with no enabled location transitions performs a discrete transition to a new stable successor state. The new execution state is identical to the predecessor state, except that it is

stable.

$$\begin{array}{l} DISCTRANS_2((loc, \sigma_{in}, \sigma_{ts}, stable), \\ \qquad\qquad (loc', \sigma'_{in}, \sigma'_{ts}, stable')) \\ \qquad =_{def} \\ (\nexists p \in \mathbb{N} : (p \in dom(R(loc))\wedge \\ \quad (loc, \sigma_{in}, \sigma_{ts}, stable) \models \Pi_1(R(loc)(p)))\wedge \\ \neg stable) \\ \qquad \Longrightarrow \\ (loc' = loc\wedge \\ \sigma'_{in} = \sigma_{in}\wedge \\ \sigma'_{ts} = \sigma_{ts}\wedge \\ stable') \end{array}$$

(3) DELAYTRANS: A stable state performs a delay transition into an unstable state with unchanged location, free inputs and possible timer elapses:

$$\begin{array}{l} DELAYTRANS((loc, \sigma_{in}, \sigma_{ts}, stable), \\ \qquad\qquad (loc', \sigma'_{in}, \sigma'_{ts}, stable')) \\ \qquad =_{def} \\ stable \\ \qquad \Longrightarrow \\ (loc' = loc\wedge \\ \forall v \in VAR_{ts} : \neg\sigma_{ts}(v) \Rightarrow \neg\sigma'_{ts}(v)\wedge \\ \neg stable') \end{array}$$

For any system state $(loc, \sigma_{in}, \sigma_{ts}, stable)$, the outputs of an automaton can be calculated simply as $L_{out}(loc)$.

It is important to observe, that the above transition relation $T : S \times S$ again constitutes a well formed semantics definition. There are (1) no spurious execution state transitions within the state transition relation, for which no constraints on the successor state exist. Additionally, (2) the definition ensures, that each execution state has at least one successor state.

To rephrase condition (1) in other words, every execution state $s \in S$ fulfills at least one of the premises of predicates $DISCTRANS_1$, $DISCTRANS_2$ and $DELAYTRANS$, and its successor states are therefore liable to be constrained by at least one predicate conclusion. Moreover, each state s fulfills the premise of precisely one of the predicates.

Proof. Let $s = (loc, \sigma_{in}, \sigma_{ts}, stable)$.

Case 1 – Stable state:

Suppose *stable*. Then neither premise of predicates $DISCTRANS_1$ and $DISCTRANS_2$ can hold, since they mandate $\neg stable$. However, the premise of predicate $DELAYTRANS$ is precisely *stable*. $DELAYTRANS$ is the one and only predicate, whose premise is fulfilled.

Case 2 – Unstable state:

Suppose $\neg stable$. The premise of predicate $DELAYTRANS$ can then never be fulfilled, since it is precisely *stable*.

Case 2.1 – Unstable state / enabled transitions

Suppose furthermore, that at least one location transitions emanating from *loc* is enabled. Then there exists at least one priority $p \in \mathbb{N}$, which is assigned to an enabled transition. Moreover, there must be a smallest priority p, which is assigned to an enabled transition. The premise of predicate $DISCTRANS_1$ holds:

$$\begin{aligned}
&\exists p \in \mathbb{N} : (p \in dom(R(loc)) \wedge \\
&\quad (loc, \sigma_{in}, \sigma_{ts}, stable) \models \Pi_1(R(loc)(p)) \wedge \\
&\quad (\nexists p' \in \mathbb{N} : (p' \in dom(R(loc)) \wedge \\
&\qquad (loc, \sigma_{in}, \sigma_{ts}, stable) \models \Pi_1(R(loc)(p')) \wedge \\
&\qquad p' < p))) \wedge \\
&\neg stable
\end{aligned}$$

The premise of predicate $DISCTRANS_2$ does not hold, since it demands the non-existence of any priority p with enabled transition. In this case, $DISCTRANS_1$ is the one and only predicate, whose premise is fulfilled.

Case 2.2 – Unstable state / no enabled transitions

Suppose now, that no location transitions emanating from *loc* are enabled. Then there exists no priority $p \in \mathbb{N}$, which is assigned to an enabled transition. The premise of predicate $DISCTRANS_2$ holds:

$$\begin{aligned}
&\nexists p \in \mathbb{N} : (p \in dom(R(loc)) \wedge \\
&\quad (loc, \sigma_{in}, \sigma_{ts}, stable) \models \Pi_1(R(loc)(p))) \wedge \\
&\neg stable
\end{aligned}$$

The premise of predicate $DISCTRANS_1$ does not hold, since – among other things – it demands the existence of such a priority p. In this case, $DISCTRANS_2$ is the one and only predicate, whose premise is fulfilled.

$\square$

Condition (2) from above necessitated, that transition relation $T : S \times S$ be total. each execution state $s \in S$ must have a successor state $s' \in S$. Taking condition (1) into account, this is now easy to see.

Proof. Consider the definition of transition relation $T : S \times S$:

$$\begin{aligned} T = \{(s, s') \in S \times S \mid \ & DISCTRANS_1(s, s') \wedge \\ & DISCTRANS_2(s, s') \wedge \\ & DELAYTRANS(s, s')\} \end{aligned}$$

Since each predicate is an implication with a premise formulated over s, and since any state s always fulfills precisely one premise, it remains to be shown, that for each predecessor state s there exists a successor state s', which fulfills the respective implication conclusion.

Case 1 – Stable state:

Stable states s fulfill the premise of predicate $DELAYTRANS$. A successor state s', which fulfills the predicate conclusion can always be constructed. Consider the relevant constraints for $s' = (loc', \sigma'_{in}, \sigma'_{ts}, stable')$:

$$\begin{aligned} & loc' = loc \wedge \\ & \forall v \in VAR_{ts} : \neg\sigma_{ts}(v) \Rightarrow \neg\sigma'_{ts}(v) \wedge \\ & \neg stable' \end{aligned}$$

As compared to the predecessor state, the successor location remains unchanged. No constraints are placed on the successor input valuation function, so such a function can exist. The only constraints placed on the successor timer status valuation function are, that no elapsed timers may start by themselves, and even predecessor σ_{ts} might be used as successor σ'_{ts}. Finally, the successor state must be unstable.

Case 2.1 – Unstable state / enabled transitions

Unstable states s with enabled transitions fulfill the premise of predicate $DISCTRANS_1$. A successor state s', which fulfills the predicate conclusion can always be constructed. Consider the relevant constraints for $s' = (loc', \sigma'_{in}, \sigma'_{ts}, stable')$:

$$
\begin{array}{l}
loc' = \Pi_2(R(loc)(p)) \wedge \\
\sigma'_{in} = \sigma_{in} \wedge \\
(\forall v \in VAR_{ta} : \\
\quad (v \notin dom(L_{ta}(\Pi_2(R(loc)(p)))) \wedge \sigma'_{ts}(\beta(v)) = \sigma_{ts}(\beta(v))) \vee \\
\quad (v \in dom(L_{ta}(\Pi_2(R(loc)(p)))) \wedge \sigma'_{ts}(\beta(v)) = L_{ta}(\Pi_2(R(loc)(p)))(v))) \wedge \\
\neg stable'
\end{array}
$$

As there must be an enabled transition with best priority p, p must be within the domain of $R(loc)$. As such, loc' is simply the second element of tuple $R(loc)(p)$. As compared to the predecessor state, the successor input valuations remain unchanged, and the only constraints placed on the timer status valuations are the effects of target location entry timer actions. As such, the successor timer status valuation function σ'_{ts} can be constructed directly from σ_{ts} and location labeling function $L_{ta}(loc')$. The successor state remains unstable.

Case 2.2 – Unstable state / no enabled transitions

Unstable states s with no enabled transitions fulfill the premise of predicate $DISCTRANS_2$. A successor state s', which fulfills the predicate conclusion can always be constructed. Consider the relevant constraints for $s' = (loc', \sigma'_{in}, \sigma'_{ts}, stable')$:

$$
\begin{array}{l}
loc' = loc \wedge \\
\sigma'_{in} = \sigma_{in} \wedge \\
\sigma'_{ts} = \sigma_{ts} \wedge \\
stable'
\end{array}
$$

The successor state s' is identical to the predecessor state s with the exception, that s' is stable.

□

2.2.6 Determinism

All Timed Moore Automata show deterministic behavior with respect to any given sequence of inputs. While stable execution states may have multiple valid successor states due to new input and timer elapse valuations, any unstable predecessor state s can only have a single successor state s'.

Proof. In the case of an unstable predecessor state s without enabled transitions, the conclusion of predicate $DISCTRANS_1$ immediately specifies the one and only successor state s', which is identical to s except for its stability.

In the case of an unstable predecessor state s with enabled transitions, the conclusion of predicate $DISCTRANS_2$ immediately specifies the one and only successor state s', which is identical to s except for its timer status valuation function σ'_{ts}. However, since σ'_{ts} is directly constructed from σ_{ts}, and since L_{ta} is a function, σ'_{ts} is also unique.

□

2.3 Model Checking for Timed Moore Automata

This section deals with explicit model checking for Timed Moore Automata. Particularly, we consider the means necessary to detect the (non-)existence of live-lock situations within Timed Moore Automata.

This section firstly describes the algorithms necessary in order to explicitly construct Kripke structures over Timed Moore Automata. It then recites the needed prerequisites regarding Computation Tree Logic as well as our take on the well-established Computation Tree Logic model checking algorithms presented in [JGP99]. Finally, some consideration is given to optimizing model checking for live-lock situations in Timed Moore Automata.

2.3.1 Construction of Kripke structures

Model checking for Timed Moore Automata involves the explicit enumeration of all system states an automaton execution may adopt. Each system state must contain all information regarding current automaton location, input-, output-, timer activation- and timer status valuation as well as the automaton's current stability.

As such, the state space for explicit model checking is rather larger than the state space used in section 2.2.5 when defining the operational semantics. In an effort to somewhat combat the effects of the resulting state space explosion, we employ *don't-care* input- and timer status valuations for situations, where the evolution of states into successor states is independent of concrete valuations. These mechanisms are inspired by the notion of *delayed non-determinism* from [NS08].

The explicit state space S_{TMA} considered during model checking of Timed Moore Automata is defined as follows.

Definition 8. *Given a Timed Moore Automaton*

$$(LOC, loc_0, VAR_{in}, VAR_{out}, VAR_{ta}, VAR_{ts}, \beta, L_{out}, L_{ta}, R)$$

the explicit state space S_{TMA} is defined as:

$$\begin{aligned} S_{TMA} = \;& LOC \times \\ & (VAR_{in} \longrightarrow L(\mathbb{B})) \times \\ & (VAR_{out} \longrightarrow \mathbb{B}) \times \\ & (VAR_{ts} \longrightarrow L(\mathbb{B})) \times \\ & (VAR_{ta} \longrightarrow \mathbb{B}) \times \\ & \mathbb{B} \end{aligned}$$

It consists of 6-tuples $(loc, \sigma_{in}, \sigma_{out}, \sigma_{ts}, \sigma_{ta}, stable)$ with the following elements:

(1) $loc \in LOC$ indicates the automaton's current location

(2) $\sigma_{in} \in (VAR_{in} \longrightarrow L(\mathbb{B}))$ is an input variable valuation function

(3) $\sigma_{out} \in (VAR_{out} \longrightarrow \mathbb{B})$ is an output variable valuation function

(4) $\sigma_{ts} \in (VAR_{ts} \longrightarrow L(\mathbb{B}))$ is a timer status variable valuation function

(5) $\sigma_{ta} \in (VAR_{ta} \longrightarrow \mathbb{B})$ is a timer action variable valuation function

(6) stable $\in \mathbb{B}$ indicates, whether the automaton is currently stable and accepts new inputs

The set $L(\mathbb{B})$ used above is a lattice to reflect don't-care-conditions for input- and timer status variable valuations. The element $\top$ denotes a don't-care-valuation, i.e. a possible valuation of either $true$ or $false$. The element $\bot$ is required for $L(\mathbb{B})$ to be a lattice. Finally, $\sqsubseteq$ is the partial order relation for $L(\mathbb{B})$.

We define $(L(\mathbb{B}), \sqsubseteq)$ as:

$$\begin{aligned}
&L(\mathbb{B}) = \{\top, true, false, \bot\}\\
&\sqsubseteq: L(\mathbb{B}) \times L(\mathbb{B})\\
&\quad \bot \sqsubseteq true \wedge\\
&\quad \bot \sqsubseteq false \wedge\\
&\quad \bot \sqsubseteq \top \wedge\\
&\quad true \sqsubseteq \top \wedge\\
&\quad false \sqsubseteq \top
\end{aligned}$$

The atomic propositions used for model checking of Timed Moore Automata involve an automaton's current location, input-, output-, timer activation- and timer status valuation as well as the automaton's stability. Variable symbols are already mapped to Boolean values, so they can immediately be considered to be atomic propositions. Locations can either be currently active or not, so again their symbols may immediately be used as atomic propositions. In order to capture the stability of a given system state, we introduce artificial atomic proposition *idle*.

Definition 9. *Given a Timed Moore Automaton*

$$(LOC, loc_0, VAR_{in}, VAR_{out}, VAR_{ta}, VAR_{ts}, \beta, L_{out}, L_{ta}, R)$$

the set of atomic propositions AP is defined as:

$$AP = LOC \cup VAR_{in} \cup VAR_{out} \cup VAR_{ta} \cup VAR_{ts} \cup \{idle\}$$

Using the above definitions, we may now introduce a labeling function L_{ap}. Its purpose is to enumerate all atomic propositions, which hold in a given system state s. Its definition is given below.

Definition 10. *The atomic proposition labeling function L_{ap} is defined as:*

$$\begin{aligned}
&L_{ap} : S_{TMA} \longrightarrow \mathbb{P}(AP)\\
&L_{ap}((loc, \sigma_{in}, \sigma_{out}, \sigma_{ts}, \sigma_{ta}, stable)) =\\
&\quad \{loc\}\cup\\
&\quad \{v \in VAR_{in} \mid true \sqsubseteq \sigma_{in}(v)\}\cup\\
&\quad \{v \in VAR_{out} \mid true = \sigma_{out}(v)\}\cup\\
&\quad \{v \in VAR_{ts} \mid true \sqsubseteq \sigma_{ts}(v)\}\cup\\
&\quad \{v \in VAR_{ta} \mid true = \sigma_{in}(v)\}\cup\\
&\quad \{idle \mid true = stable\}
\end{aligned}$$

Using the above definitions of explicit state space, atomic propositions and atomic proposition labeling function, we can finally define the Kripke structures employed to perform model checking for Timed Moore Automata.

Definition 11. *Given a Timed Moore Automaton*

$$(LOC, loc_0, VAR_{in}, VAR_{out}, VAR_{ta}, VAR_{ts}, \beta, L_{out}, L_{ta}, R)$$

the Kripke structure $K(M)$ is defined as a 4-tuple:

$$K = (S_K, s_{K_0}, T_K, L_{ap})$$

Its elements are:

(1) The set of all reachable system states $S_K \subseteq S_{TMA}$

(2) The initial system state $s_{K_0} \in S_K$

(3) The state transition relation $T_K : S_K \times S_K$

(4) The atomic labeling function $L_{ap} : S_K \longrightarrow \mathbb{P}(AP)$

The explicit construction of such Kripke structures is the focus of the remainder of this section. It involves (1) the creation of the initial system state, (2) the iterative expansion of all known system states to their respective successor states and (3) the collection of all predecessor-successor relationships of system states. The procedure *createKripkeStructure*() from figure 2.3 formalizes the process.

```
procedure $createKripkeStructure()$
  $s_{K_0} := createInitialState()$
  $S_K := \{s_{K_0}\}$
  $T_K := \emptyset$
  $S_{new} := S_K$
  while $S_{new} \neq \emptyset$
    let $s_K \in S_{new}$
    $S_{new} := S_{new} \backslash \{s_K\}$
    $S_{succ} := createSuccStates(s_K)$
    $S_{new} := S_{new} \cup (S_{succ} \backslash (S_K \cap S_{succ}))$
    $S_K := S_K \cup S_{succ}$
    forall $s'_K \in S_{succ}$ do
      $T_K := T_K \cup \{(s_K, s'_K)\}$
    end forall
  end while
end procedure
```

Figure 2.3: Procedure for creating Kripke structure

Initially, s_{K_0} is created using the function $createInitialState()$. It constitutes the initial member of S_K. Additionally, it is initially the only element of the set $S_{new} \subseteq S_K$ of states, which still need to be evolved to their successor states. The transition relation T_K is initially empty.

The construction of a Kripke structure is performed within a loop, which terminates as soon as there are no more system states to be evolved. Within the loop, a single system state to be evolved is selected, and its successor states $S_{succ} \subseteq S_K$ are calculated using function $createSuccStates()$. Using these newly calculated system states, S_K, T_K and S_{new} are updated.

The function $createInitialState()$ from figure 2.4 initialized the valuation functions σ_{in}, σ_{out}, σ_{ts} and σ_{ta} of the initial state arbitrarily. As designated by the operational semantics from section 2.2.5, all input valuations are initially undefined and are therefore set to $\top$. All other valuations are set to false. The initial location is given by loc_0. The system state is considered unstable since it must still perform its initial run to completion.

The function $createSuccStates()$ from figure 2.5 merely delegates the calculation of successor states in accordance to the stability of the predecessor state. The function $createSuccStatesStable()$ calculates the effects of delay transitions, the function $createSuccStatesRunning()$ calculates the effects

```
function $createInitialState() : S_{TMA}$
  let $\sigma_{in} \in (VAR_{in} \longrightarrow L(\mathbb{B}))$
  let $\sigma_{out} \in (VAR_{out} \longrightarrow \mathbb{B})$
  let $\sigma_{ts} \in (VAR_{ts} \longrightarrow L(\mathbb{B}))$
  let $\sigma_{ta} \in (VAR_{ta} \longrightarrow \mathbb{B})$
  forall $v \in VAR_{in}$ do
    $\sigma_{in} := \sigma_{in} \oplus \{v \mapsto \top\}$
  end forall
  forall $v \in VAR_{out}$ do
    $\sigma_{out} := \sigma_{out} \oplus \{v \mapsto false\}$
  end forall
  forall $v \in VAR_{ta}$ do
    $\sigma_{ta} := \sigma_{ta} \oplus \{v \mapsto false\}$
    $\sigma_{ts} := \sigma_{ts} \oplus \{\beta(v) \mapsto false\}$
  end forall
  $s_{K_0} := (loc_0, \sigma_{in}, \sigma_{out}, \sigma_{ts}, \sigma_{ta}, false)$
  $createInitialState := s_{K_0}$
end function
```

Figure 2.4: Function for creating initial Kripke state

of discrete transitions.

The function *createSuccStatesStable*() from figure 2.6 captures the transition from a stable system state to its successor unstable state with indeterminate inputs. As compared to a predecessor state s_K, the lone successor state s'_K contains an input valuation function σ'_{in}, which maps all input variables to indeterminate valuation $\top$. The new timer status valuation function σ'_{ts} allows for running timers to either be still running or be elapsed. The successor state is set to be unstable, since another run to completion is required whenever modifying inputs or timer statuses.

Given an unstable system state s_K, the function *createSuccStatesRunning*() from figure 2.7 calculates unstable successor states. Since the current location of s_K may have emanating transitions with guard conditions, which reference variable symbols with indeterminate valuations $\top$, the function $unfoldDontCareInputs()$ is employed to resolve the delayed non-determinism of state s_K into a set of states S_{unf}. The states in S_{unf} will then provide concrete valuations for all variable symbols appearing in emanating transition guards.

```
**function** $\mathit{createSuccStates}$(**in** $s_K \in S_{TMA}$) : $\mathbb{P}(S_{TMA})$
  **let** $s_K = (\mathit{loc}, \sigma_{in}, \sigma_{out}, \sigma_{ts}, \sigma_{ta}, \mathit{stable})$
  **if** $\mathit{stable}$
    $\mathit{createSuccStates} := \mathit{createSuccStatesStable}(s_K)$
  **else**
    $\mathit{createSuccStates} := \mathit{createSuccStatesRunning}(s_K)$
  **end if**
**end function**
```

Figure 2.5: Function for creating successor Kripke states

```
**function** $\mathit{createSuccStatesStable}$(**in** $s_K \in S_{TMA}$) : $\mathbb{P}(S_{TMA})$
  **let** $s_K = (\mathit{loc}, \sigma_{in}, \sigma_{out}, \sigma_{ts}, \sigma_{ta}, \mathit{stable})$
  $\sigma'_{in} := \sigma_{in}$
  **forall** $v \in VAR_{in}$ **do** $\sigma'_{in} := \sigma'_{in} \oplus \{v \mapsto \top\}$
  $\sigma'_{ts} := \sigma_{ts}$
  **forall** $v \in VAR_{ts}$ **do**
    **if** $\sigma'_{ts}(v) = \mathit{true}$ **then** $\sigma'_{ts} := \sigma'_{ts} \oplus \{v \mapsto \top\}$
  **end forall**
  $s'_K := (\mathit{loc}, \sigma'_{in}, \sigma_{out}, \sigma'_{ts}, \sigma_{ta}, \neg\mathit{stable})$
  $\mathit{createSuccStatesStable} := \{s'_K\}$
**end function**
```

Figure 2.6: Function for creating successors for stable Kripke states

```
**function** $createSuccStatesRunning(\textbf{in}\ s_K \in S_{TMA}) : \mathbb{P}(S_{TMA})$
  **let** $s_K = (loc, \sigma_{in}, \sigma_{out}, \sigma_{ts}, \sigma_{ta}, \neg stable)$
  $S_{unf} := unfoldDontCareInputs(s_K)$
  $S_{succ} := \emptyset$
  **while** $S_{unf} \neq \emptyset$
    **let** $s'_K \in S_{unf}$
    $S_{unf} := S_{unf} \setminus \{s'_K\}$
    $s''_K := performTransition(s'_K)$
    $S_{succ} := S_{succ} \cup \{s''_K\}$
  **end while**
  $createSuccStatesRunning := S_{succ}$
**end function**
```

Figure 2.7: Function for creating successors for running Kripke states

After resolving indeterminate input valuations, the set S_{unf} is then iteratively processed to calculate the successor state for each element. This is done by function $performTransition()$.

The function $unfoldDontCareInputs()$ from figure 2.8 is used to resolve delayed non-determinism for all variables appearing in any guard condition of an emanating transition. The function therefore collects all input- and timer status variables with valuation $\top$ and appearing in emanating transition guards in the sets VAR_{care_in} and VAR_{care_ts} respectively. The resulting set of system states S_{unf} is then calculated by producing all combinations of $true$ and $false$ valuations for all variables in VAR_{care_in} and VAR_{care_ts}.

The function $performTransitions()$ from figure 2.9 evaluates all transitions leaving the current location loc in order of their priorities. Each transition's guard condition is evaluated using valuation functions σ_{in} and σ_{ts}. Note, that while a valuation of $\top$ fulfills a guard constraint of either $true$ or $false$, such valuations should never appear due to prior applications of function $unfoldDontCareInputs()$.

If an enabled transition could be found, the successor state contains the target location of the transition as new current location. The entry actions of that location are applied to valuation functions σ_{out}, σ_{ta} and σ_{ts}. The successor state remains unstable since successive discrete transitions may be possible.

If no enabled transition could be found, the successor state is identical to

```
**function** $unfoldDontCareInputs(\textbf{in}\ s_K \in S_{TMA}) : \mathbb{P}(S_{TMA})$
  **let** $s_K = (loc, \sigma_{in}, \sigma_{out}, \sigma_{ts}, \sigma_{ta}, \neg stable)$
  $S_{unf} := \emptyset$
  $VAR_{care\_in} := \emptyset$
  $VAR_{care\_ts} := \emptyset$
  **forall** $p \in dom(R(loc))$ **do**
    **forall** $v \in VAR_{in}$ **do**
      **if** $(v \in dom(\Pi_1(R(loc)(p)))) \wedge (\sigma_{in}(v) = \top)$
        $VAR_{care\_in} := VAR_{care\_in} \cup v$
      **end if**
    **end forall**
    **forall** $v \in VAR_{ts}$ **do**
      **if** $(v \in dom(\Pi_1(R(loc)(p)))) \wedge (\sigma_{ts}(v) = \top)$
        $VAR_{care\_ts} := VAR_{care\_ts} \cup v$
      **end if**
    **end forall**
  **end forall**
  $\Sigma_{comb} := \mathbb{B}^{(VAR_{care\_in} \cup VAR_{care\_ts})}$
  **forall** $\sigma_{comb} \in \Sigma_{comb}$ **do**
    $\sigma'_{in} := \sigma_{in}$
    $\sigma'_{ts} := \sigma_{ts}$
    **forall** $v\ inVAR_{in}$ **do**
      $\sigma'_{in} := \sigma'_{in} \oplus \{v \mapsto \sigma_{comb}(v)\}$
    **end forall**
    **forall** $v\ inVAR_{ts}$ **do**
      $\sigma'_{ts} := \sigma'_{ts} \oplus \{v \mapsto \sigma_{comb}(v)\}$
    **end forall**
    $s'_K := (loc, \sigma'_{in}, \sigma_{out}, \sigma'_{ts}, \sigma_{ta}, \neg stable)$
    $S_{unf} := S_{unf} \cup \{s'_K\}$
  **end forall**
  $unfoldDontCareInputs := S_{unf}$
**end function**
```

Figure 2.8: Function for unfolding states with Don't-Care inputs

the predecessor state, except that it becomes stable to support a successive delay transition.

2.3.2 Computation Tree Logic

The model checking algorithms for Timed Moore Automata presented here enable checking against properties formulated in Computation Tree Logic CTL. As the name suggests, CTL was originally conceived to specify properties for generic (and possibly infinite) trees of computation states. While the mentioned computation states correspond perfectly to the system states $s_K \in S_K$ discussed so far, this chapter deals with Kripke structures (i.e. directed graphs) of system states rather than with trees. However, this is mediated by the facts, that (1) the concept of paths as sequences of states is identical when dealing with trees and directed graphs, and that (2) algorithms exist (and are presented later in this chapter) to evaluate CTL on directed graphs.

CTL differentiates state formulas, which argue over the properties of a single system state, from path formulas, which argue over properties of entire paths through the tree.

For a given path as a sequence of states, path formulas may contain unary operators **X**, **F** and **G** to specify, that a certain property must hold in the **next** state, that it must **finally** hold in some state along the path, or that it must **globally** hold in all states within the path. Furthermore, a path formula may contain binary operators **U** and **R** to specify, that one property must hold **until** another property holds, or that the occurrence of one property **releases** another property from having to hold.

For a given state, state formulas may contain conjunctions, disjunctions or negations of atomic propositions, which hold in that state. Additionally, they may argue over all possible paths emanating from the state. For this purpose, unary operators **E** and **A** may be employed to specify, that there **exists** an emanating path fulfilling a specific path constraint, or that **all** emanating paths fulfill a specific path constraint.

The formal syntax of all valid CTL formulas is defined below.

Definition 12. *Consider the set of atomic propositions AP. The syntax of state- and path formulas is then defined:*

```
function $performTransition$(in $s_K \in S_{TMA}$) : $S_{TMA}$
  let $s_K = (loc, \sigma_{in}, \sigma_{out}, \sigma_{ts}, \sigma_{ta}, \neg stable)$
  forall $p \in dom(R(loc))$ from $min(dom(R(loc)))$ to $max(dom(R(loc)))$ do
    $\gamma := \Pi_1(R(loc)(p))$
    $loc' := \Pi_2(R(loc)(p))$
    $guard := true$
    forall $v \in VAR_{in}$ do
      if $v \in dom(\gamma)$
        $guard := guard \wedge (\gamma(v) \sqsubseteq \sigma_{in}(v))$
      end if
    end forall
    forall $v \in VAR_{ts}$ do
      if $v \in dom(\gamma)$
        $guard := guard \wedge (\gamma(v) \sqsubseteq \sigma_{ts}(v))$
      end if
    end forall
    if $guard$ then break
  end forall
  if $guard$
    $\sigma'_{out} := \sigma_{out}$
    $\sigma'_{ts} := \sigma_{ts}$
    $\sigma'_{ta} := \sigma_{ta}$
    forall $v \in VAR_{out}$ do
      $\sigma'_{out} := \sigma'_{out} \oplus \{v \mapsto L_{out}(loc')(v)\}$
    end forall
    forall $v \in VAR_{ta}$ do
      if $v \in dom(L_{ta}(loc'))$
        $\sigma'_{ta} := \sigma'_{ta} \oplus \{v \mapsto L_{ta}(loc')(v)\}$
        $\sigma'_{ts} := \sigma'_{ts} \oplus \{\beta(v) \mapsto L_{ta}(loc')(v)\}$
      end if
    end forall
    $s'_K := (loc', \sigma_{in}, \sigma'_{out}, \sigma'_{ts}, \sigma'_{ta}, \neg stable)$
  else
    $s'_K := (loc, \sigma_{in}, \sigma_{out}, \sigma_{ts}, \sigma_{ta}, stable)$
  end if
  $performTransition := s'_K$
end function
```

Figure 2.9: Function for transforming discrete states transitions

- *If $p \in AP$, then p is a state formula.*
- *If f and g are state formulas, then $\neg f$, $f \vee g$ and $f \wedge g$ are state formulas.*
- *If f and g are state formulas, then $\mathbf{X}\ f$, $\mathbf{F}\ f$, $\mathbf{G}\ f$, $f\ \mathbf{U}\ g$ and $f\ \mathbf{R}\ g$ are path formulas.*
- *If f is a path formula, then $\mathbf{E}\ f$ and $\mathbf{A}\ f$ are state formulas.*

The criteria, under which a given CTL formula is considered to hold are defined below.

Definition 13. *The 'models' relation $\models$: $(K \times S_K) \times CTL$ defines, which CTL state formulas hold within specific states s of a given Kripke structure. In analogy, the relation $\models$: $(K \times S_K^*) \times CTL$ defines, which CTL path formulas hold within specific paths π of a given Kripke structures. They are defined according to the structure of the formula in question:*

- $M, s \models p \quad \Leftrightarrow \quad p \in L(s)$
- $M, s \models \neg f \quad \Leftrightarrow \quad M, s \not\models f$
- $M, s \models f_1 \vee f_2 \quad \Leftrightarrow \quad M, s \models f_1$ *or* $M, s \models f_2$
- $M, s \models f_1 \wedge f_2 \quad \Leftrightarrow \quad M, s \models f_1$ *and* $M, s \models f_2$
- $M, s \models \mathbf{E}\ g \quad \Leftrightarrow \quad$ *there exists a path π from s such that $M, \pi \models g$*
- $M, s \models \mathbf{A}\ g \quad \Leftrightarrow \quad M, \pi \models g$ *for every path π from s*
- $M, \pi \models f \quad \Leftrightarrow \quad M, s \models f$ *where s is first state of π*
- $M, \pi \models \mathbf{X}\ f \quad \Leftrightarrow \quad M, \pi^1 \models f$
- $M, \pi \models \mathbf{F}\ f \quad \Leftrightarrow \quad \exists k \geq 0 : M, \pi^k \models f$
- $M, \pi \models \mathbf{G}\ f \quad \Leftrightarrow \quad \forall k \geq 0 : M, \pi^k \models f$
- $M, \pi \models f_1\ \mathbf{U}\ f_2 \quad \Leftrightarrow \quad \exists k \geq 0 : M, \pi^k \models f_2 \wedge \forall 0 \leq j < k : M, \pi^j \models f_1$
- $M, \pi \models f_1\ \mathbf{R}\ f_2 \quad \Leftrightarrow \quad \forall j \geq 0 : ((\forall i < j : M, \pi^i \not\models f_1) \Rightarrow M, \pi^j \models f_2)$

2.3.3 Model Checking CTL Properties

In order to perform model checking of CTL formulas against Timed Moore Automata Kripke structures, we use adaptations of the algorithms presented in [JGP99]. To begin with, CTL formulas are transformed into a standardized form, which contains only a limited number of CTL operator combinations. After that, the algorithms used in model checking of Timed Moore Automata for each operator combination are given.

When carefully analyzing the syntax of CTL as given in section 2.3.2 it becomes clear, that path quantifiers arguing over a state **A** and **E** are always paired with path operators **X**, **F**, **G**, **U** or **R**. Since – except when using the predicate logic operators $\neg$, $\vee$ or $\wedge$ – state formulas are always defined with respect to subordinate path formulas and vice versa, each state formula CTL operator must always be followed by a path formula operator. It is therefore sufficient to consider only state formulas and the following combinations of applicable operators:

- $\neg$
- $\vee$
- $\wedge$
- **AX**
- **EX**
- **AF**
- **EF**
- **AG**
- **EG**
- **AU**
- **EU**
- **AR**
- **ER**

This list can further be restricted to only contain state formula operators $\neg$, $\vee$, **EX**, **EG** and **EU** using the following tautologies:

- $f \wedge g \quad = \quad \neg(\neg f \vee \neg g)$
- $\mathbf{AX}\ f \quad = \quad \neg\mathbf{EX}(\neg f)$
- $\mathbf{EF}\ f \quad = \quad \mathbf{E}(true\ \mathbf{U}\ f)$
- $\mathbf{AG}\ f \quad = \quad \neg\mathbf{EF}(\neg f)$
- $\mathbf{AF}\ f \quad = \quad \neg\mathbf{EG}(\neg f)$
- $\mathbf{A}(f\ \mathbf{U}\ g) \quad = \quad \neg\mathbf{E}(\neg g\ \mathbf{U}\ (\neg f \wedge \neg g)) \wedge \neg\mathbf{EG}(\neg g)$
- $\mathbf{A}(f\ \mathbf{R}\ g) \quad = \quad \neg\mathbf{E}(\neg f\ \mathbf{U}\ \neg g)$
- $\mathbf{E}(f\ \mathbf{R}\ g) \quad = \quad \neg\mathbf{A}(\neg f\ \mathbf{U}\ \neg g)$

It remains to be shown, how model checking can be performed for state formulas containing these state formula operators $\neg$, $\vee$, **EX**, **EG** and **EU**. This is accomplished by enhancing our labeling function L_{ap} from section 2.3.1. Rather than just labeling Kripke system states with atomic propositions, we now want to label states with CTL properties. Note, that we do not want to label each system state with *all* fulfilled CTL properties, but are rather interested in a labeling, which is guided by a single CTL property to be checked.

The labeling function $L_{CTL} : S_K \longrightarrow \mathbb{P}(CTL)$ is therefore introduced to label system states with sets of CTL formulas. It is iteratively defined using algorithms given later. In its initial form, it corresponds to L_{AP} in the sense, that all states are labeled with the primitive atomic proposition CTL formulas as already established by L_{AP}.

Definition 14. *Let $K = (S_K, s_{K_0}, T_K, L_{ap})$ be a Timed Moore Automaton Kripke structure and $f \in CTL$ be a formula to be checked. We define the* **initial** *CTL labeling function L_{CTL} as:*

$$\begin{array}{l} L_{CTL} : S_K \longrightarrow \mathbb{P}(CTL) \\ \forall s_K \in S_K : L_{CTL}(s_K) = L_{AP}(s_K) \cap \overline{AP}(f) \end{array}$$

Here, $\overline{AP}(f) \subseteq AP$ is the set of atomic propositions, which appear within formula f.

```
procedure labelCTL(in $f \in CTL$)
  switch $f$
    case $\neg f_1$ :
      labelCTL($f_1$)
      labelNot($f_1$)
      break
    case $f_1 \vee f_2$ :
      labelCTL($f_1$)
      labelCTL($f_2$)
      labelOr($f_1, f_2$)
      break
    case EX $f_1$ :
      labelCTL($f_1$)
      labelEX($f_1$)
      break
    case E($f_1$ U $f_2$) :
      labelCTL($f_1$)
      labelCTL($f_2$)
      labelEU($f_1, f_2$)
      break
    case EG $f_1$ :
      labelCTL($f_1$)
      labelEG($f_1$)
      break
  end switch
end procedure
```

Figure 2.10: Procedure for labeling generic CTL state formula f

For a given formula f, the labeling L_{CTL} is now extended recursively in accordance to the outermost operator combination encountered in formula f. Procedure *labelCTL*() from figure 2.10 discriminates between all combinations, then (1) recursively performs labeling for all subordinate operand formulas and (2) delegates labeling to suitable specialized procedures.

Procedure *labelNot*() from figure 2.11 performs the labeling for formulas of the form $\neg f$. It simply labels all states with $\neg f$ if they are not already labeled with f.

Procedure *labelOr*() from figure 2.12 performs the labeling for formulas of

```
procedure labelNot(in $f \in CTL$)
  $P := S_K$
  while $P \neq \emptyset$
    let $s \in P$
    $P := P \setminus \{s\}$
    if $f \notin L_{CTL}(s)$
      $L_{CTL}(s) := L_{CTL}(s) \cup \{\neg f\}$
    end if
  end while
end procedure
```

Figure 2.11: Procedure for labeling $\neg f$

```
procedure labelOr(in $f_1 \in CTL, f_2 \in CTL$)
  $P := S_K$
  while $P \neq \emptyset$
    let $s \in P$
    $P := P \setminus \{s\}$
    if $(f_1 \in L_{CTL}(s)) \vee (f_2 \in L_{CTL}(s))$
      $L_{CTL}(s) := L_{CTL}(s) \cup \{f_1 \vee f_2\}$
    end if
  end while
end procedure
```

Figure 2.12: Procedure for labeling $f_1 \vee f_2$

the form $f_1 \vee f_2$. It labels all states, which are labeled with f_1 or f_2, with formula $f_1 \vee f_2$.

Procedure *labelEX*() from figure 2.13 performs the labeling for formulas of the form **EX** f. It loops over all states $s' \in S_K$, which are already labeled with formula f. For each s', it then collects all predecessor states s according to T_K and labels each predecessor state with formula **EX** f.

Procedure *labelEU*() from figure 2.14 performs the labeling for formulas of the form $\mathbf{E}(f_1 \mathbf{\ U\ } f_2)$. Firstly, it labels all states, which are already labeled with f_2 with $\mathbf{E}(f_1 \mathbf{\ U\ } f_2)$ also. It then loops over all states s', which are already labeled with $\mathbf{E}(f_1 \mathbf{\ U\ } f_2)$. Each predecessor state s of s' is then labeled with $\mathbf{E}(f_1 \mathbf{\ U\ } f_2)$, if it is also already labeled with f_1. The process continues until no more suitable predecessor states s can be found.

```
procedure labelEX(in $f \in CTL$)
  $P := \{s \in S_K \mid f \in L_{CTL}(s)\}$
  while $P \neq \emptyset$
    let $s' \in P$
    $P := P \backslash \{s'\}$
    $Q := \{s \in S_K \mid (s, s') \in T_K\}$
    while $Q \neq \emptyset$
      let $s \in Q$
      $Q := Q \backslash \{s\}$
      $L_{CTL}(s) := L_{CTL}(s) \cup \{\mathbf{EX}\ f\}$
    end while
  end while
end procedure
```

Figure 2.13: Procedure for labeling **EX** f

```
procedure labelEU(in $f_1 \in CTL, f_2 \in CTL$)
  $P := \{s \in S_K \mid f_2 \in L_{CTL}(s)\}$
  forall $s \in P$ do $L_{CTL}(s) := L_{CTL}(s) \cup \{\mathbf{E}(f_1\ \mathbf{U} f_2)\}$
  while $P \neq \emptyset$
    let $s' \in P$
    $P := P \backslash \{s'\}$
    $Q := \{s \in S_K \mid (s, s') \in T_K\}$
    while $Q \neq \emptyset$
      let $s \in Q$
      $Q := Q \backslash \{s\}$
      if $(\mathbf{E}(f_1\ \mathbf{U}\ f_2) \notin L_{CTL}(s)) \wedge (f_1 \in L_{CTL}(s))$
        $L_{CTL}(s) := L_{CTL}(s) \cup \{\mathbf{E}(f_1\ \mathbf{U}\ f_2)\}$
        $P := P \cup \{s\}$
      end if
    end while
  end while
end procedure
```

Figure 2.14: Procedure for labeling $\mathbf{E}(f_1\ \mathbf{U}\ f_2)$

```
procedure labelEG(in $f \in CTL$)
  $O := \{s \in S_K \mid f \in L_{CTL}(s)\}$
  $SCC := \{C \subseteq S_K \mid C \text{ is a nontrivial SCC of } O\}$
  $P := \bigcup_{C \in SCC} \{s \in S_K \mid s \in C\}$
  forall $s \in P$ do $L_{CTL}(s) := L_{CTL}(s) \cup \{\mathbf{EG}\ f\}$
  while $P \neq \emptyset$
    let $s' \in P$
    $P := P \backslash \{s'\}$
    $Q := \{s \in S_K \mid (s, s') \in T_K\}$
    while $Q \neq \emptyset$
      let $s \in Q$
      $Q := Q \backslash \{s\}$
      if $\mathbf{EG}\ f \notin L_{CTL}(s)$
        $L_{CTL}(s) := L_{CTL}(s) \cup \{\mathbf{EG}\ f\}$
        $P := P \cup \{s\}$
      end if
    end while
  end while
end procedure
```

Figure 2.15: Procedure for labeling **EG** f

Finally, procedure *labelEG*() from figure 2.15 performs the labeling for formulas of the form **EG** f. This involves calculation using the algorithm presented in [Tar71] of all nontrivial strongly connected components of S_K, for which the component members are labeled with f. All states within these components are labeled with formula **EG** f. The procedure then loops over all states s', which are already labeled with **EG** f. Each predecessor state s of s' is then labeled with **EG** f, if it is also already labeled with f. The process continues until no more suitable predecessor states s can be found.

The building blocks for model checking a Timed Moore Automaton M against a given CTL formula f can now be combined to specify the entire process:

1. From Timed Moore Automaton M, create Kripke structure $K(M)$ using procedure *createKripkeStructure*(M) from figure 2.3.
2. Using f and definition 14, create the initial CTL labeling function L_{CTL}.
3. Expand the labeling function L_{CTL} using procedure *labelCTL*(f) from

figure 2.10.

4. Check, if the initial Kripke state s_{K_0} is labeled with f, i.e. return $f \in L_{CTL}(s_{K_0})$.

2.3.4 Checking for Live-Locks

Within a Timed Moore Automaton a live-lock situation is characterized by an execution, which at some point remains eternally unstable. In such a situation an automaton will continually execute discrete location transitions and never accept new inputs again. Within an automaton implementation a live-lock generally corresponds to an infinite loop without any timer delays, and it is therefore useful to preclude such faulty behavior on the specification level.

From a model checking standpoint, an automaton must always be able to become idle eventually, so that it may accept new inputs. The following CTL constraint expresses this situation:

$$\mathbf{AF}\ \mathit{idle}$$

However, it is not sufficient to say, that starting from an automaton's initial state, the automaton will always become stable. Rather, the same must be true for all subsequent runs to completion as well. In other words, the formula stated above must not only hold for the initial execution state of an automaton, but must rather be fulfilled by all execution states, that are reachable from the automaton's initial state. Live-lock freedom is therefore expressed as:

$$\mathbf{AG}\ (\mathbf{AF}\ \mathit{idle})$$

Application of the transformation tautologies from section 2.3.3 yield the following form:

$$\begin{array}{ll} \mathbf{AG}\ (\mathbf{AF}\ \mathit{idle}) & \Leftrightarrow \\ \neg\mathbf{EF}\ (\neg\mathbf{AF}\ \mathit{idle}) & \Leftrightarrow \\ \neg\mathbf{EF}\ (\mathbf{EG}\ \neg \mathit{idle}) & \end{array}$$

In essence, this form re-expresses live-lock freedom as the following natural language statement: There is no execution, which ever reaches a state, from which onward the automaton will never be idle again.

The above representation of live-lock freedom illustrates, that model checking for live-lock freedom involves the costly calculation of nontrivial strongly connected components due to the occurrence of the operator combination **EG**. However, a special handling for this situation allows us to avoid this.

Consider the part of the source formula **AF** *idle*, which gave rise to the occurrence of the **EG** operator combination. States, which are already stable, trivially fulfill this requirement. For unstable states, this formula requires all possible subsequent computations to eventually reach a stable state.

However, recall that unstable Timed Moore Automaton execution states always have precisely one successor state. This was shown in proof 2.2.6. It follows, that within the corresponding Kripke structure an unstable system state can only have a single emanating path of states, which can only begin to branch off again once it has reached a stable state. The following CTL expressions are therefore equivalent within Timed Moore Automata:

$$\mathbf{AF}\ \mathit{idle} \ \Leftrightarrow\ \mathbf{EF}\ \mathit{idle}$$

Applying this special equivalence, live-lock freedom for Timed Moore Automata can then be expressed as:

$$\mathbf{AG}\ (\mathbf{EF}\ \mathit{idle})$$

Again transforming this into the form used for model checking, we see, that only the cheaper operators $\neg$ and **EU** need to be checked:

$$\begin{array}{ll} \mathbf{AG}\ (\mathbf{EF}\ \mathit{idle}) & \Leftrightarrow \\ \neg\mathbf{EF}\ (\neg\mathbf{EF}\ \mathit{idle}) & \Leftrightarrow \\ \neg\mathbf{E}(\mathit{true}\ \mathbf{U}\ (\neg\mathbf{E}(\mathit{true}\ \mathbf{U}\ \mathit{idle}))) & \end{array}$$

Another consideration further helps reduce the cost of checking for live-locks: from a debugging point of view a witness for a possible live-lock situation is far more valuable than the history of how that situation was reached from the initial automaton execution state. It is sufficiently worrying to know, that such a situation is reachable.

In model checking terms, this means, that it is sufficient to label a Kripke structure with the CTL formula:

$$\neg \mathbf{EF}\ idle$$

Any state within a Kripke structure, which can be labeled with this formula is then a suitable witness to analyze and debug a live-lock situation.

2.4 Test Data Generation for Timed Moore Automata

This section introduces algorithms used in the test data generation for Timed Moore Automata. While in theory the generation of test data for any given test goal can be reduced to a model checking reachability problem, it is impractical to do so for real-world testing campaigns.

This section defines the notion of (symbolic) target traces through a Timed Automaton, for which test data should be generated. Subsequently, the algorithms used to generate test data for a given target trace are presented. Finally, some consideration is given to the selection of (sets of) target traces and the implications on test object code coverage criteria.

2.4.1 Test Data Generation for Single Traces

In order to generate test data for Timed Moore Automata, we begin by defining the notion of target traces through automata, for which we intend to generate test data. Conceptually, a target trace can be viewed as a sequence of location transitions to be taken by an execution. The goal of test data generation for such a trace is then to produce a sequence of input assignments, which will enforce the selected trace to be executed and to produce a sequence of output valuations to be expected from the automaton. Additionally, the execution of the automaton along a selected target trace should become stable as often as possible, so that the test environment may assert as many output valuations as possible.

In order to formalize target traces, we consider sequences of pairs of location symbols and natural numbers. Each pair $(loc, p) \in (LOC \times \mathbb{N})$ denotes a start location and the priority of an emanating transition to be taken. A sequence of pairs hence formalizes a sequence of location transitions to be enforced by input assignments.

Definition 15. *The set of test data generation target traces is defined as:*

$$TT = (LOC \times \mathbb{N})^*$$

Given a trace $\pi \in TT$, two predicates become relevant for test data generation. The predicate $C_{trace}(\pi)$ will collect all conditions over input- and timer status variable valuations, which have to hold in order for a subsequent run to completion to take the given trace π.

The predicate $C_{stable}(\pi)$ is even stronger. In addition to enforcing the specified trace it collects necessary constraints, which have to hold in order for a computation to become stable in the location immediately following the last transition of π.

The predicates $C_{trace}(\pi)$ and $C_{stable}(\pi)$ are defined below.

Definition 16. *Given a single target location transition $(loc, p) \in (LOC \times \mathbb{N})$ the predicate $C_{trans}(loc, p)$ encapsulates constraints, that have to hold for a transition emanating from loc with priority p to be enabled:*

$$\begin{aligned}C_{trans}(loc \in LOC, p \in dom(R(loc))) :=& \\ &\textstyle\bigwedge_{\{v \in dom(\Pi_1(R(loc)(p)))\}} (v = \Pi_1(R(loc)(p))(v))\end{aligned}$$

Given a location $loc \in LOC$, the predicate $C_{timer}(loc)$ encapsulates the effects of timer activation entry actions when entering location loc:

$$C_{timer}(loc \in LOC) := \bigwedge_{\{v \in dom(L_{ta}(loc))\}} (\beta(v) = L_{ta}(loc)(v)))$$

Given a location $loc \in LOC$, the predicate $C_{stop}(loc)$ contains all constraints, which disable all transitions emanating from loc, i.e. enforce loc to remain stable:

$$\begin{aligned}C_{stop}(loc \in LOC) :=\ & C_{timer}(loc) \wedge \\ & \textstyle\bigwedge_{\{p \in dom(R(loc))\}} (\neg C_{trans}(loc, p))\end{aligned}$$

Given a single target location transition $(loc, p) \in (LOC \times \mathbb{N})$ the predicate $C_{force}(loc, p)$ encapsulates constraints, that enforce the given target transition to be taken:

$$\begin{array}{rl} C_{force}(loc \in LOC, p \in dom(R(loc))) := & C_{timer}(loc) \wedge \\ & C_{trans}(loc, p) \wedge \\ & \bigwedge_{\{p' \in dom(R(loc)) | p' < p\}} (\neg C_{trans}(loc, p')) \end{array}$$

Given a trace $\pi \in TT$, the predicate $C_{trace}(\pi)$ contains all constraints, which have to hold in order for an execution to take all transitions specified in trace π:

$$C_{trace}(\pi \in (LOC \times \mathbb{N})^*) := \bigwedge_{\{(loc,p) \in \pi\}} (C_{force}(loc, p))$$

Finally, given a trace $\pi \in TT$, the predicate $C_{stable}(\pi)$ contains all constraints, which have to hold in order for an execution to take the trace π and become stable after the last transition was taken:

$$\begin{array}{rl} C_{stable}(\pi \in (LOC \times \mathbb{N})^*) := & C_{trace}(\pi) \wedge \\ & C_{stop}(last(\pi)) \end{array}$$

Before assembling the algorithms for test data generation, we need to consider another building block. In analogy to the model checking approach discussed in the previous section, we need some notion of concrete system states. We utilize the state space $\overline{S_{TMA}}$ as defined below:

Definition 17. *The state space $\overline{S_{TMA}}$ used for test data generation for Timed Moore Automata is defined as:*

$$\begin{array}{rl} \overline{S_{TMA}} = & LOC \times \\ & (VAR_{in} \longrightarrow \mathbb{B}) \times \\ & (VAR_{out} \longrightarrow \mathbb{B}) \times \\ & (VAR_{ts} \longrightarrow \mathbb{B}) \times \\ & (VAR_{ta} \longrightarrow \mathbb{B}) \end{array}$$

Note, that $\overline{S_{TMA}}$ closely resembles the state space S_{TMA} from model checking, except that (1) we do not require a Boolean valuation for the stability of a state since we will only consider stable states, and that (2) we need not consider don't-care valuations any more. We can now introduce following state transition system:

Definition 18. *The state transition system used for test data generation is defined as a 3-tuple:*

$$(S_T, s_{T_0}, T_T)$$

Its elements are:

(1) The set of states $S_T \subseteq \overline{S_{TMA}}$

(2) The initial state $s_{T_0} \in S_T$

(3) The state transition relation $T_T : S_T \times S_T$

Note, that as opposed to the model checking approach presented previously, we will not explicitly enumerate all elements of S_T and T_T. Instead, we will only need to explicitly construct the initial state s_{T_0} and use a concrete interpreter to calculate successor states.

Using the predicates C_{trace}, C_{stable} and the initial state s_{T_0}, we can now specify the algorithms used to generate test data for a given trace π.

Function $generateTestCase()$ from figure 2.16 starts by calculating the initial state of an execution using function $createInitialTestState()$ and assigns it as the current stable state. It then performs a loop, which partitions the target trace π into partial traces.

Each partial trace must start in the current stable state, so that new inputs can be assigned. These inputs must then force the execution to travel along the target trace π. Function $generateTestStep()$ calculates input- and timer status assignments, which enforce the shortest such partial trace possible starting from the current stable state. Function $interpret()$ is then used to calculate the next current stable state (along target trace π) as indicated by the calculated inputs.

An entire trace π is considered feasible, if all invocations of function $generateTestStep()$ were successful. As outputs, the function $generateTestCase()$ collects all input assignments mandated by $generateTestStep()$ as well as all expected output valuations as predicted by $interpret()$.

```
**function** $generateTestCase(\textbf{in}\ \pi \in (LOC \times \mathbb{N}),$
                    $\textbf{out}\ data_{in} \in ((VAR_{in} \longrightarrow \mathbb{B}) \times (VAR_{ts} \longrightarrow \mathbb{B}))^*),$
                    $\textbf{out}\ data_{out} \in ((VAR_{out} \longrightarrow \mathbb{B}) \times (VAR_{ts} \longrightarrow \mathbb{B}))^*) : \mathbb{B}$
  $data_{in} :=<>$
  $data_{out} :=<>$
  $s_T := createInitialTestState()$
  **let** $s_T = (loc, \sigma_{in}, \sigma_{out}, \sigma_{ts}, \sigma_{ta})$
  **while** $\mid \pi \mid > 0$
    $\sigma'_{in} := \sigma_{in}$
    $\sigma'_{ts} := \sigma_{ts}$
    $sat := generateTestStep(\pi, \sigma'_{in}, \sigma'_{ts})$
    **if** $sat$
      $push\_back(data_{in}, (\sigma'_{in}, \sigma'_{ts}))$
      $s_T := (loc, \sigma'_{in}, \sigma_{out}, \sigma'_{ts}, \sigma_{ta})$
      $s'_T := interpret(s_T)$
      **let** $s'_T = (loc', \sigma''_{in}, \sigma'_{out}, \sigma''_{ts}, \sigma'_{ta})$
      $push\_back(data_{out}, (\sigma'_{out}, \sigma''_{ts}))$
    **else**
      **break**
    **end if**
  **end while**
  $generateTestCase := sat$
**end function**
```

Figure 2.16: Function for generating test case data

```
function $createInitialTestState() : \overline{S_{TMA}}$
  let $\sigma_{in} \in (VAR_{in} \longrightarrow \mathbb{B})$
  let $\sigma_{out} \in (VAR_{out} \longrightarrow \mathbb{B})$
  let $\sigma_{ts} \in (VAR_{ts} \longrightarrow \mathbb{B})$
  let $\sigma_{ta} \in (VAR_{ta} \longrightarrow \mathbb{B})$
  forall $v \in VAR_{in}$ do
    $\sigma_{in} := \sigma_{in} \oplus \{v \mapsto false\}$
  end forall
  forall $v \in VAR_{out}$ do
    $\sigma_{out} := \sigma_{out} \oplus \{v \mapsto false\}$
  end forall
  forall $v \in VAR_{ta}$ do
    $\sigma_{ta} := \sigma_{ta} \oplus \{v \mapsto false\}$
    $\sigma_{ts} := \sigma_{ts} \oplus \{\beta(v) \mapsto false\}$
  end forall
  $s_{T_0} := (loc_0, \sigma_{in}, \sigma_{out}, \sigma_{ts}, \sigma_{ta})$
  $createInitialTestState := s_{T_0}$
end function
```

Figure 2.17: Function for creating initial test data generation state

The function $createInitialTestState()$ from figure 2.17 is used to calculate an initial system state, which can then be used by function $generateTestCase()$ as a basis. Note, that the initial assignments of of input- and timer status valuations are arbitrary, since they may be overwritten by a subsequent invocation of $generateTestStep()$.

Function $generateTestStep()$ from figure 2.18 performs the trace partitioning for a given trace π, which is at the heart of the test data generation process. Starting with a prefix trace containing only the first transition from π, it iteratively attempts to find the shortest feasible prefix trace of π, which ends in a stable state. For each prefix trace candidate π_{step}, it calculates the predicate $C_{stable}(\pi)$ containing all constraints over input- and timer status valuations, which have to hold in order for an execution to take the trace π_{step} and become stable after the last transition is taken. The predicate is then passed to *MINISat* ([SE02]), a SAT-Solver well suited for solving Boolean satisfiability problem instances.

Should function $generateTestStep()$ be unable to find a feasible stable prefix trace for π, the function attempts as a last resort to find suitable input- and

```
function $generateTestStep$(inout $\pi \in (LOC \times \mathbb{N})^*$,
                            out $\sigma_{in} \in (VAR_{in} \longrightarrow \mathbb{B})$,
                            out $\sigma_{ts} \in (VAR_{ts} \longrightarrow \mathbb{B})) : \mathbb{B}$
  $\pi_{step} :=<>$
  $\pi_{remain} := \pi$
  $sat := false$
  while $\mid \pi_{remain} \mid > 0$
    $push\_back(\pi_{step}, head(\pi_{remain}))$
    $pop\_front(\pi_{remain})$
    if $solve(C_{stable}(\pi_{step}))$
      $sat := true$
      $(\sigma_{in}, \sigma_{ts}) := solution()$
      $\pi := \pi_{remain}$
      break
    end if
  end while
  if $\neg sat$
    if $solve(C_{trace}(\pi))$
      $sat := true$
      $(\sigma_{in}, \sigma_{ts}) := solution()$
      $\pi :=<>$
    end if
  end if
  $generateTestStep := sat$
end function
```

Figure 2.18: Function for generating test step data

timer status valuations to enforce the entire trace π. Note, that in this case predicate C_{trace} is utilized since it is unclear which subsequent state might become stable.

Function $generateTestStep()$ returns the feasibility of finding inputs enforcing a (partial) trace π ending in a stable state. If suitable input- and timer status valuations could be found, then the function returns the remainder of trace π as well as the input- and timer status valuations themselves.

Function $interpret()$ from figure 2.19 performs concrete interpretation for a given input state s_T. It is used to calculate a stable successor state for a given input state s_T. As such, it is a close cousin to the function $performTransition()$ (figure 2.9) from subsection 2.3.1. It evaluates all transitions leaving the current location loc in order of their priorities.

If an enabled transition could be found, the successor state contains the target location of the transition as new current location. The entry actions of that location are applied to valuation functions σ_{out}, σ_{ta} and σ_{ts}. The successor state is then subjected to another invocation of $interpret()$ since it is still unstable.

If no enabled transition could be found, the successor state is identical to the predecessor state, except that it now becomes stable.

2.4.2 Trace Selection

Up to this point, this section on test data generation was focused on generating data for single target traces. However, in order to test entire Timed Moore Automata, multiple traces need to be considered.

Using the nomenclature from [SLS05], we consider a *test procedure* to contain a sequence of *test cases*, which in turn may contain a sequence of *test steps*. In mapping these notions to our discussion up to this point, each partial target trace constructed by function $generateTestStep()$ constitutes a *test step*, since for each partial trace we have:

(1) Test pre-conditions — the inputs to be assigned

(2) A test event — a run to completion of the automaton

(3) Test post-conditions – the expected outputs of the automaton

```
function $interpret$(in $s_T \in \overline{S_{TMA}}$) : $\overline{S_{TMA}}$
  let $s_T = (loc, \sigma_{in}, \sigma_{out}, \sigma_{ts}, \sigma_{ta})$
  forall $p \in dom(R(loc))$ from $min(dom(R(loc)))$ to $max(dom(R(loc)))$ do
    $\gamma := \Pi_1(R(loc)(p))$
    $loc' := \Pi_2(R(loc)(p))$
    $guard := true$
    forall $v \in VAR_{in}$ do
      if $v \in dom(\gamma)$
        $guard := guard \wedge (\gamma(v) = \sigma_{in}(v))$
      end if
    end forall
    forall $v \in VAR_{ts}$ do
      if $v \in dom(\gamma)$
        $guard := guard \wedge (\gamma(v) = \sigma_{ts}(v))$
      end if
    end forall
    if $guard$ then break
  end forall
  if $guard$
    $\sigma'_{out} := \sigma_{out}$
    $\sigma'_{ts} := \sigma_{ts}$
    $\sigma'_{ta} := \sigma_{ta}$
    forall $v \in VAR_{out}$ do
      $\sigma'_{out} := \sigma'_{out} \oplus \{v \mapsto L_{out}(loc')(v)\}$
    end forall
    forall $v \in VAR_{ta}$ do
      if $v \in dom(L_{ta}(loc'))$
        $\sigma'_{ta} := \sigma'_{ta} \oplus \{v \mapsto L_{ta}(loc')(v)\}$
        $\sigma'_{ts} := \sigma'_{ts} \oplus \{\beta(v) \mapsto L_{ta}(loc')(v)\}$
      end if
    end forall
    $s'_T := interpret((loc', \sigma_{in}, \sigma'_{out}, \sigma'_{ts}, \sigma'_{ta}))$
  else
    $s'_T := (loc, \sigma_{in}, \sigma_{out}, \sigma_{ts}, \sigma_{ta})$
  end if
  $interpret := s'_T$
end function
```

Figure 2.19: Concrete interpretation function

Each target trace, for which function $generateTestCase()$ has constructed test data, then gives rise to a *test case*, since the function returns sequences of input- and output assignments, i.e. a sequence of *test steps*. It remains to group multiple *test cases* into *test procedures*. While this grouping is arbitrary, it is important to consider the set of test cases in terms of test coverage criteria.

Given a standard implementation for Timed Moore Automata, where current locations are handled within a top-level **switch**-statement, and transitions are handled within each location **case**, generated test data for a set of feasible target traces, which exercises each location transition of an automaton, will only fulfill the *statement coverage* test end criterion.

Given a set of feasible target traces, which – in addition to exercising all location transitions – ensures, that for each location there is at least one target trace, where the respective location becomes stable, it is possible to amend the given test case data generation algorithm to include robustness test cases. Given a stable state, such robustness test cases might calculate (using the SAT-solver) all input combinations, which cause the state to remain stable. Such a test strategy would then yield test cases, which fulfill the *condition coverage* criterion, since for each location each transition has evaluated to $true$ as well as to $false$. Additionally, the robustness test cases might even lead to complete *condition decision coverage*.

Within this thesis, we consider the set of test cases, which enforce all location transitions and a maximum of stable states. And while some robustness test cases are added to each stable state situation, this still only yields partial *condition coverage*. A more concise coverage criterion is impractical, since (1) it is rarely possible to force all locations in an automaton to become stable, and (2) the pertinent safety standards for the railway domain only require *branch coverage*.

The pragmatic selection of target trace candidates is inspired by the calculation of the *transition cover* set from Chow's influential paper [Cho78] on the W-Method. However, while the W-method assumes all target traces to be feasible, this is not true for Timed Moore Automata, and we need to enable dynamic expansion of the unrolled transition graph in order to be able to produce multiple trace candidates for any given target transition.

2.5 Benchmarks

Model checking for live-locks and generation of test cases for Timed Moore Automata was performed for a real world railway level crossing application consisting of a collection of 27 automata using a 2.0 GHz Intel Core 2 Duo processor with 2 GB of RAM. Table 2.1 shows the results. Columns are labeled as follows:

1. $\#L$ — Number of locations in the automaton
2. $\#T$ — Number of transitions in the automaton
3. $\#IN$ — Number of inputs in the automaton
4. $\#TM$ — Number of timers in the automaton
5. $\#S$ — Number of states in the Kripke structure
6. t_{KS} — Time in milliseconds to construct the Kripke structure
7. t_{MC} — Time in milliseconds to check for live-locks
8. $\#TC$ — Number of test cases generated to cover the automaton
9. t_{TC} — Time to construct test cases

Checking automata for live-locks was always possible on the given hardware and never took more than 430 milliseconds in total. Since all live-locks were removed in earlier software iterations, this constitutes a worst-case execution time, since all Kripke states need to be considered during model checking.

The construction of test cases yielded complete condition coverage for all locations, which could be made stable, and complete statement coverage otherwise. The generation of test cases never took more than 60 milliseconds.

Both model checking for live-locks and test case generation could be performed instantaneously for smaller automata with approximately less locations than 10.

#L	#T	#IN	#TM	#S	t_{KS}	t_{MC}	#TC	t_{TC}
2	2	1	0	6	< 1	< 1	2	< 1
3	5	2	0	18	< 1	< 1	5	< 1
5	6	4	0	62	< 1	< 1	6	< 1
5	12	4	1	91	< 1	< 1	11	< 1
7	14	4	0	111	< 1	< 1	14	< 1
7	19	5	0	270	< 1	10	17	< 1
8	10	4	1	91	< 1	< 1	10	< 1
8	12	3	0	66	< 1	< 1	12	< 1
9	13	7	0	226	< 1	< 1	11	< 1
8	15	6	0	346	< 1	< 1	15	< 1
10	15	4	1	165	< 1	< 1	15	< 1
10	19	5	2	358	< 1	10	16	10
10	19	7	2	289	< 1	< 1	19	10
12	34	5	0	310	< 1	< 1	28	< 1
12	37	7	0	501	10	< 1	29	20
12	38	7	0	442	10	< 1	28	60
13	35	8	0	1071	20	20	35	20
16	28	8	1	756	10	10	27	20
18	29	6	2	574	10	10	27	10
18	41	10	0	1485	40	20	38	50
19	28	4	4	306	< 1	< 1	24	10
19	34	7	0	291	< 1	10	33	20
22	38	10	3	513	10	10	37	10
22	40	5	0	455	< 1	< 1	31	10
24	54	14	2	1613	30	30	43	30
25	73	8	3	5095	120	310	63	40
33	48	9	5	3090	50	110	43	20

Table 2.1: Performance results for Timed Moore Automata benchmark

Chapter 3

Interactive Model-Based Testing

This chapter discusses an existing framework for model-based test data and test procedure generation. The chapter in turn describes extensions to the framework, which allow interactive interventions into the generation process to fine-tune results according to user application domain expertise.

Section 3.1 gives an overview over the framework under consideration in its current form.

Section 3.2 describes modifications to the control flow, which allow for more user guided influence on the generation process. Subsequently, section 3.3 outlines the user interface used to control the presented interactive test generation paradigm.

Section 3.4 gives an application example for the interactive generation paradigm in the form of a small case study. Finally, section 3.5 evaluates the interactive generation paradigm in comparison to existing tools.

3.1 Model-Based Testing Framework

Within this thesis, we consider the framework for model-based test data generation as presented in [Pel13] and [PHL+11]. In this framework, test models may be specified using several different modeling formalisms. Using an intermediate model representation – an abstract syntax suitable to represent test models of different modeling formalisms – concrete instances of test models are represented in memory for further use.

As such, the intermediate model representation of a given test model specifies, which valuations must be considered within each system state of a test model execution. Furthermore, the intermediate model representation is used to manufacture the transition relation, which explicitly correlates predecessor and successor system states. The operational semantics of a test model is then implicitly given as a state transition system over all system states, which are reachable from the test model's initial state.

Test generation goals are specified as Linear Temporal Logic (LTL) expressions over test model elements. Combined with the transition relation of a test model, this yields bounded model checking problem instances, which are passed to a Satisfiability-Modulo-Theory solver. Resulting test model computations are then stored within a tree of system states. Finally, each trace through such a computation tree is then refined into an executable test procedure.

3.1.1 Intermediate Model Representation

Test models for the framework under consideration are most commonly given as collections of UML2 composite structure and state chart diagrams. Starting with a root component, composite structure diagrams are employed to partition the model into a hierarchy of components. Each leaf component is then associated with a state chart diagram in order to assign it it's behavior.

Variables and timers needed to specify behavior may be declared as attributes of components within the component hierarchy. This implies the scope of each variable and timer, all child components of a declaring component may read and write the variable or timer in question.

Note, that assignable inputs and observable outputs of a system under test are modeled in the same way. They are characterized as part of the system input/output interface using stereotypes designated for that purpose. While inputs and outputs may be declared on any level of the component hierarchy, common modeling practice usually places them on the top level of the component hierarchy.

The abstract syntax – referred to as intermediate model representation within the framework – consists of data structures to reflect hierarchies of components with associated variables, timers and hierarchic state charts in memory.

While the intermediate model representation primarily caters to UML2 models, it is suitable to represent test models specified in a variety of modeling formalisms. Currently, front-end parser components exist for subsets of UML2, SysML, MatLab Simulink as well as some other domain-specific modeling formalisms.

3.1.2 Operational Semantics

The behavior of a given test model is specified as a state transition system (S, s_0, R) with system state space S, such that each system State $s \in S$ is comprised of a valuation function $S \in V \mapsto D$.

Here, $V = I \cup O \cup V \cup T \cup L$ denotes the set of variables necessary to describe the behavior of the test model. It contains the model's input variables I, its output variables O, internal variables V, timers T as well as variables L to encode the state chart locations, which are active within a given system state. The set D denotes the corresponding valuation domain for each variable.

The initial system state $s_0 \in S$ can immediately be constructed using just the abstract syntax of a given test model, since this specifies start locations for all state charts as well as initial valuations for all variables and timers.

In order to define the state transition relation $R \in S \times S$, we construct the state transition predicate $\Phi(s, s')$, which argues over all variable valuations from any two system states $s, s' \in S$. For any system states s the predicate $\Phi(s, s')$ becomes true if and only if s' is a possible successor state for s. Using Φ, the system state transition R is then defined as:

$$R := \{(s, s') \in S \times S \mid (s, s') \models \Phi(s, s')\}$$

As described in [PHL+11], the state transition predicate is generated to closely reflect the semantics of state charts as posed by Harel in [HN96].

All state charts posses urgency, they must perform transitions between their respective locations immediately whenever these transitions are enabled. All state charts execute these discrete transitions in parallel and in zero time. State charts may assign new variable valuations, however, input variable valuations must remain unchanged. Conflicting valuation assignments of more than one state chart to the same variable constitute a dead-lock in the model, since the state transition predicate Φ can never accommodate such assignments.

Whenever no discrete transitions are possible, the system must perform a delay transition. Within a delay transition (1) time must elapse and (2) no state chart transition may become urgently enabled. All variable valuations must remain unchanged. New input variable valuations may be assigned to the system at the end of a delay transition.

The global system time tick $t_{system} \in T$ serves as indication of elapsed execution time and is used as reference for all running timers of a system during delay transitions. Test models can accommodate dense real-time as long as the valuation domain $D(t_{system})$ is rational, i.e. $D(t_{system}) \subset \mathbb{Q}$. Test models may then be abstracted to use clock regions or clock zones as given by [JGP99].

3.1.3 Generation Goals

The generation process is guided by generation goals specified by the user. Such generation goals generally correspond to specific test cases, for which test data is needed. Generally, generation goals specify properties, which a generated computation – a sequence of system states – π must fulfill. Generation goal properties are specified using Linear Temporal Logic (LTL).

We consider predicate logic expressions over V to be our atomic propositions. An expression $exp \in AP$ can be evaluated in a system state $s \in S$ by replacing all occurrences of variables from $v \in V$ with their valuations $s(v)$. To indicate that an expression exp evaluates to true when replacing all variables with their valuations from state s we use the notation $s \models exp$.

As opposed to CTL, LTL does not contain path quantifiers, since it only argues about the properties of a single path. Its syntax is defined below.

Definition 19. *Consider a set of atomic propositions AP. The syntax of a path formula is then defined inductively:*

- *If $p \in AP$, then p is a path formula.*
- *If f and g are path formulas, then $\neg f$, $f \vee g$ and $f \wedge g$ are path formulas.*

- *If f and g are path formulas, then* $\mathbf{X}\ f$, $\mathbf{F}\ f$, $\mathbf{G}\ f$, $f\ \mathbf{U}\ g$ *and* $f\ \mathbf{R}\ g$ *are path formulas.*

Given a sequence of states $\pi \in S^*$, a LTL formula is considered to hold under the following conditions:

Definition 20. *The 'models' relation* $\models: S^* \times LTL$ *defines, which LTL formulas hold within specific paths* $\pi \in S^*$*. Given paths* π*, atomic propositions* exp *and generic LTL formulas* f *and* g*, it is defined according to the structure of the formula in question:*

- $\pi \models exp \quad \Leftrightarrow \quad s \models exp$ *where* s *is first state of* π
- $\pi \models \neg f \quad \Leftrightarrow \quad \pi \not\models f$
- $\pi \models f \vee g \quad \Leftrightarrow \quad \pi \models f$ *or* $\pi \models g$
- $\pi \models f \wedge g \quad \Leftrightarrow \quad \pi \models f$ *and* $\pi \models g$
- $\pi \models \mathbf{X}\ f \quad \Leftrightarrow \quad \pi^1 \models f$
- $\pi \models \mathbf{F}\ f \quad \Leftrightarrow \quad \exists k \geq 0 : \pi^k \models f$
- $\pi \models \mathbf{G}\ f \quad \Leftrightarrow \quad \forall k \geq 0 : \pi^k \models f$
- $\pi \models f\ \mathbf{U}\ g \quad \Leftrightarrow \quad \exists k \geq 0 : \pi^k \models g \wedge \forall 0 \leq j < k : \pi^j \models f$
- $\pi \models f\ \mathbf{R}\ g \quad \Leftrightarrow \quad \forall j \geq 0 : ((\forall i < j : \pi^i \not\models f) \Rightarrow \pi^j \models g)$

In their simplest form, generation goals formalize a reachability problem. It may for instance be desirable to generate a computation, which eventually assigns a specific value val_1 to an output variable out_1. Such a generation goal would then be specified as:

$$\mathbf{F}(out_1 = val_1)$$

A more complex example could be a test case, which requires a specific sequence of locations loc_1, loc_2, loc_3 to be fulfilled by a computation. Such a generation goal might be specified as:

$$\mathbf{F}(loc_1 \wedge (\mathbf{X}(loc_2 \wedge (\mathbf{X}(loc_3)))))$$

Generally, any LTL property arguing over symbols of a test model may be used as generation goal. However, some properties – while syntactically valid – yield unsatisfying results. This is due to the fact, that the test generation process relies on bounded model checking. More specifically, given a generation goal LTL property g the test case generation will always produce the shortest possible computation π, which fulfills g.

While reachability properties of the form

$$\mathbf{F}(exp)$$

are well suited for test generation, invariants of the form

$$\mathbf{G}(exp)$$

are not very useful because the generation process would – at best – yield a computation π consisting of a single state s, such that $s \models exp$.

However, this apparent flaw is to be expected since testing is rarely the methodology of choice to prove invariants in a system under consideration. From the testers perspective, a witness of a violated invariant should be the aim. For the given example, a tester would hence pose the reachability property

$$\mathbf{F}(\neg exp)$$

as generation goal.

Generally, liveness properties may be used as generation goals while safety properties should be tested by attempting to construct witnesses for their violation.

3.1.4 Bounded Model Checking

It is natural to extend the above definition of our state transition predicate to encompass unrolling of the transition relation. The predicate $\Phi(s, s')$ specifies the transition from one predecessor state to one successor state. We can then define the corresponding predicate for a sequence of possible system evolutions:

$$\Phi(i) := \Phi(s_0, s_1) \wedge \Phi(s_1, s_2) \wedge \ldots \wedge \Phi(s_{i-1}, s_i)$$

Additionally, a given LTL property p may be transformed into a predicate $G(p, i)$ arguing over a finite sequence of system states of length i. Such an unrolling of LTL formulas into a bounded model checking problem is given in [BHJ$^+$06].

The test generation problem $tc(i)$ for generation goal p can now be described as the following bounded model checking instance of length i:

$$tc(i) := s_0 \wedge \Phi(i) \wedge G(p, i)$$

$\Phi(i)$ contains the transition relation of the test model, unrolled to reflect i possible transitions (delay or discrete). $G(p, i)$ contains the generation goal as encoded when only checking computations of length i. Finally, s_0 contains all valuations of the current system state, which is to be the basis of the generation process.

Given a maximum bound of system evolutions, the bounded model checking problem instance $tc(i)$ is then repeatedly passed to a satisfiability-modulo-theory solver in order to determine a solution for the smallest possible unrolling i. The framework under consideration utilizes SONOLAR – as presented in[PVL11] – as solver.

3.1.5 Computation Tree

Typically, the test generation process involves multiple generation goals. Experience shows, that it is unnecessarily complex to attempt to solve each generation goal individually and using the test model's initial system state as the basis for generation. Rather, the generation process will attempt to find a solution for any remaining generation goals. Using a back-tracking strategy system states created from previous generation results serve as basis for following generation steps.

While each solved generation goal results in a computation in the form of a sequence of system states, the generation result for multiple generation goals is then a tree of system states, such that each tree leaf corresponds to a computation, which realizes a generation goal. The back-tracking algorithm used to identify start states for generation steps thereby ensures, that generated computations share as many computation prefixes as possible.

Note, that executable code for the stimulation of a system under test must still be extracted from the computation tree. While the computation tree contains all valuations for all involved system states, stimulator code must only contain assignments to system inputs. Additionally, underlying delay transitions must be detected within the computation tree in order to sort input assignments into timed input assignment sequences.

Furthermore, note, that test oracles used to evaluate the correctness of system under test responses to generated stimuli are generic. Their construction is a purely syntactic transformation from the concrete test model syntax to executable test code.

3.1.6 Concrete Interpreter

A solution model for a given generation goal encodes all intermediate system states within a computation, which realizes the generation goal. However, such a computation must not necessary end in a stable system state, i.e. a system state, which can only perform a delay transition.

In case of solutions, which give rise to computations ending in an unstable state, we still need to extend that computation to the next stable state. In theory, this can be accomplished using the components already described. Given a deterministic transition relation predicate Φ, unstable state s and undetermined successor state s', the predicate $\Phi(s, s')$ should only have a single solution.

In practice it is more sensible to implement a concrete interpreter, which explicitly applies the rules of the operational semantics, rather than relying on a solver and Φ as implicit specification of the semantics. Additionally, a concrete interpreter provides code redundancy, which helps to validate the generation of Φ from the intermediate model representation. If a solution yielded by the solver cannot be reproduced by the concrete interpreter, there must be an intolerable discrepancy between these two semantics specifications.

3.1.7 Generation Control Flow

Figure 3.1 describes the control flow of the generation process. Given a test model and a set of generation goals, the generation framework firstly parses its inputs in order to construct the intermediate model representation, the state transition predicate Φ and the initial computation tree consisting only of a single system state reflecting the model's initial state.

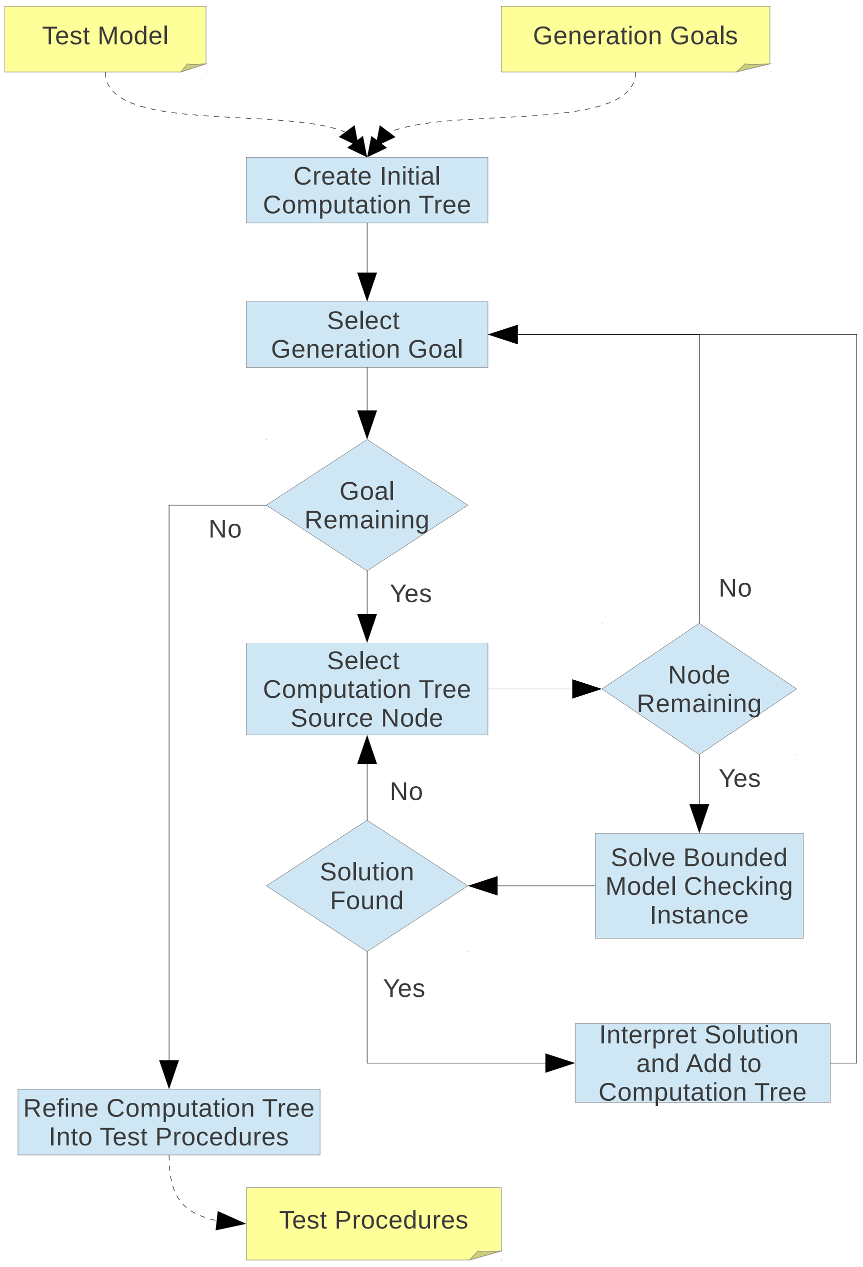

Test Model
Generation Goals
Create Initial Computation Tree
Select Generation Goal
Goal Remaining
No
Yes
Select Computation Tree Source Node
Node Remaining
No
Yes
Solve Bounded Model Checking Instance
Solution Found
No
Yes
Interpret Solution and Add to Computation Tree
Refine Computation Tree Into Test Procedures
Test Procedures

Figure 3.1: Control Flow - Fully Automated Approach

Depending on a configurable strategy, the framework will then iteratively select a generation goal from the set of generation goals to be solved. For each generation goal, a back-tracking algorithm then iteratively selects a source node from the computation tree to be used as a basis for a subsequent generation attempt.

Given a specific generation goal p, a computation tree source node s_0 and a maximum unrolling bound k, the framework then attempts to solve the bounded model checking problem

$$\bigvee_{i=1}^{k} tc(i)$$

using the satisfiability-modulo-theory solver SONOLAR.

If a solution could be found it is added to the computation tree. In order to do this the solution model is refined into a timed sequence of input assignments, which in turn serves as input to the concrete interpreter. The resulting computation is then inserted as a new child computation branch to system state s_0 within the computation tree.

Once the generation process cannot find any more additional generation goal solutions – either because all goals were solved, or because the remaining goals were infeasible – the computation tree is refined into test procedures. Since each trace starting from the computation tree root node to a leaf node represents a computation realizing a generation goal, each such trace is refined into executable code. This purely syntactical transformation consists of creating the appropriate sequence of executable statements for input assignments and time delays.

3.2 Interactive Generation Paradigm

The model-based test generation framework described thus far is afflicted by some weaknesses, which can be addressed by modifying its current control flow. This section presents an effort to do so by providing multiple opportunities for the user to interactively intervene in the generation process.

3.2.1 Critique of the Fully Automated Paradigm

The fully automated test generation paradigm as described above contains intrinsic weaknesses, which may cause generation results to be lacking with respect to readability and maintainability. Generated test procedures may successfully realize given generation goals, but they may do so in an unexpected fashion with respect to the intention of a generation goal.

Generation results may contain input assignments, which realize generation goals in unforeseen ways. Consider for example the turn indication function of an automobile. Given a generation goal, which postulates turning directional turn indication flashing on and subsequently off again, the generation process may turn flashing on by moving the turn indication lever, but then turn flashing off again by turning the ignition key. While this is a valid solution to the generation goal, it is probably not, what the test engineer had in mind.

Additionally, generation results may contain superfluous and unrelated input assignments. A generated test procedure may indeed turn a directional turn indication on and off again, but it may modify – for example – the battery voltage of the vehicle within a range, that has no influence on the turn indication. These input assignments are again valid with respect to a given generation goal, but modifications to unrelated inputs are distracting and unnecessary.

It is generally not sufficient to formulate generation goals to reflect what functionality of a system under test should be exercised. Rather, generation goals need to additionally exclude valuation changes in those parts of a system under test, which should remain untouched. The reason for this is, that the underlying satisfiability-modulo-theory solver has no notion of the easiest or most intuitive solution. Rather, it relies on solver-intrinsic heuristics to quickly find any solution for a given problem instance. The result is, that generated test input assignment sequences can be very hard to comprehend. This in turn impedes the validation of test results.

Another weakness arises from the fact, that back-tracking along an intermediate computation tree to find a suitable source node for a subsequent generation step is again heuristic and not guided by any application domain knowledge. This can cause difficulty in maintaining sets of generation goals, since modifications to a single generation goal can in turn influence multiple subsequent generation steps.

The fully automated test generation paradigm allows for the user to inject application domain knowledge into the generation process only through the postulation of very specific generation goals and the delicate configuration of the generation goal selection sequence and back-tracking algorithm. This in turn necessitates the time consuming process of repeatedly generating test procedures for *all* generation goals, until goals and configurations are sufficiently fine-tuned. This process is additionally complicated by the fact, that analysis of generation results must rely on generated executable code, since the underlying computation tree is discarded at the end of the generation process.

To address these weaknesses, this thesis introduces a modified control flow and a suitable user interface, which allow the user to inject application knowledge while the generation process is running. This allows for the following interactive features:

- Visualization of each intermediate state of the computation tree as well as each system state contained within.
- Model checking computation trees for LTL pre-conditions to facilitate the user-guided selection of generation source nodes.
- Interactive selection of computation sub-trees to be pruned from an existing intermediate computation tree.
- Interactive expansion of a computation tree by manually assigning input valuations to a new branch.
- Interactive selection of LTL generation goals to extend an intermediate computation tree using bounded model checking.
- Interactive selection of computation tree traces to be refined into test procedures.

3.2.2 Modifications to the Generation Control Flow

While re-using multiple components from the fully automated generation paradigm, the interactive generation paradigm features a new control flow with multiple break points, where user interaction is required. Figure 3.2 shows the modified control flow.

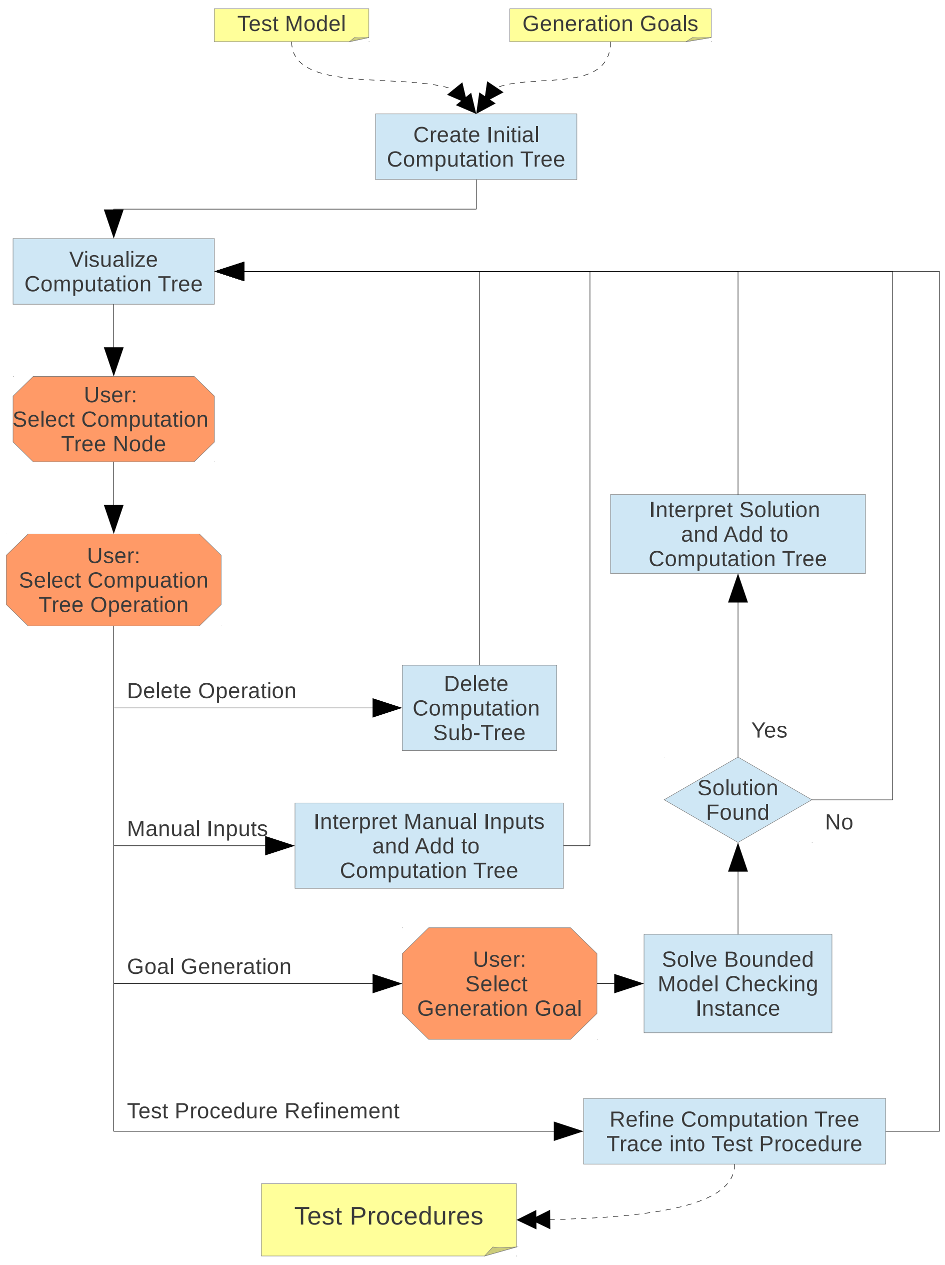

Test Model
Generation Goals
Create Initial Computation Tree
Visualize Computation Tree
User: Select Computation Tree Node
User: Select Computation Tree Operation
Delete Operation
Delete Computation Sub-Tree
Manual Inputs
Interpret Manual Inputs and Add to Computation Tree
Goal Generation
User: Select Generation Goal
Solve Bounded Model Checking Instance
Solution Found
Yes
No
Interpret Solution and Add to Computation Tree
Test Procedure Refinement
Refine Computation Tree Trace into Test Procedure
Test Procedures

Figure 3.2: Control Flow - Interactive Approach

As before, a given test model and a set of generation goals is parsed in order to construct an instance of the intermediate model representation and the initial trivial computation tree. The control flow then executes a main loop.

Firstly, the computation tree is visualized in a graph representation. Within the graph, the user may select any system state in order to analyze the contained input-, output-, variable-, location- and timer valuations.

Using the computation tree visualization, the user can then select a system state to be modified by a subsequent computation tree operation. While pure visual inspection of a computation tree might be sufficient for a user to decide where to modify the computation tree, additional support is given. The user may ask the generator to perform model checking for LTL properties on the computation tree in order to identify traces, which fulfill useful pre-conditions. As a result, all traces fulfilling a given LTL property are highlighted for further user inspection.

Once the user has selected a system state several operations are available to be performed on the computation tree. Firstly, the user may delete the computation sub-tree specified by the currently selected system state. This operation is generally performed whenever generation results were achieved using too coarse generation goals and therefore need to be discarded.

Secondly, the user may create a copy of a selected system state in order to modify its inputs manually. This operation causes the modified system state to be interpreted by the concrete interpreter, and the resulting computation is added to the selected (modified) computation tree node.

Additionally, the user may select a generation goal to be solved using the selected system state as a source state. This operation re-uses the bounded model checking approach from before.

Finally, the user may earmark the selected system state for test procedure generation. The computation from the computation tree root node to the selected computation tree node is then refined into executable test procedure code.

3.2.3 Model Checking for Computation Trees

The selection of a source computation tree node to be extended subsequently is facilitated by performing model checking of computation trees against LTL

properties. It is useful to identify traces within a computation tree, which fulfill given LTL properties, whenever pre-conditions for a test generation step are required, which cannot easily be identified by visual inspection of a – possibly quite large – computation tree.

Given an intermediate computation tree and an LTL property, the model checking process is twofold. Firstly, all candidate traces $\pi \in S^*$ within the computation tree are identified. Secondly, each trace is analyzed to determine, whether it fulfills the given LTL property.

The set of candidate traces is constructed using nested depth-first searches within the computation tree. The outer search begins with the currently user-selected computation tree node and enumerates candidate trace start nodes. The inner search begins with the current start node candidate and enumerates candidate trace end nodes.

Combined, the nested searches enumerate candidate start- and end nodes, and the intermediate trace nodes can be inferred. Note, that both the outer (start node) and inner (end node) searches terminate their respective recursions once a candidate trace actually fulfills the given LTL property. As a result, the algorithm yields the *closest shortest* matching traces along each computation tree branch starting from the currently user-selected computation tree node.

For each candidate trace π of length i we can check whether π fulfills LTL property p by again unrolling p into predicate $G(p, i)$ and substituting all variables for all states along π with the concrete valuations from π. If the resulting predicate simplifies to $true$, we have, that $\pi \models p$.

Note, that $G(p, i)$ must return predicate $false$ for all LTL properties p, which due to their structure cannot be unrolled in i steps.

3.2.4 Manual Computation Tree Extension

In order to assign inputs manually, the user must select a stable system state, since new inputs can only be assigned during delay transitions. Such a state is then cloned and added as a child to the originally selected state. As such, the newly created clone contains the same valuations as its successor, so that new input assignments can be accumulative rather than having to be exhaustive.

The user may then modify input valuations of the cloned system state at will. Additionally, the user may forward the system time up to the point, where the next running timer would elapse. In either case, the modified system state is considered to be provisionally unstable, since the modified input- and system time valuations may cause discrete transitions to be enabled.

Once the user is satisfied with the modifications, concrete interpretation is performed in order to calculate the resulting computation up to the next stable system state.

3.2.5 Generation Goal driven Computation Tree Extension

Generating computations according to given generation goals is performed as before.

Given a generation goal p, selected computation tree source node s_0 and a maximum unrolling bound k, the bounded model checking instance $\bigvee_{i=1}^{k} tc(i)$ is passed to the satisfiability-modulo-theory solver.

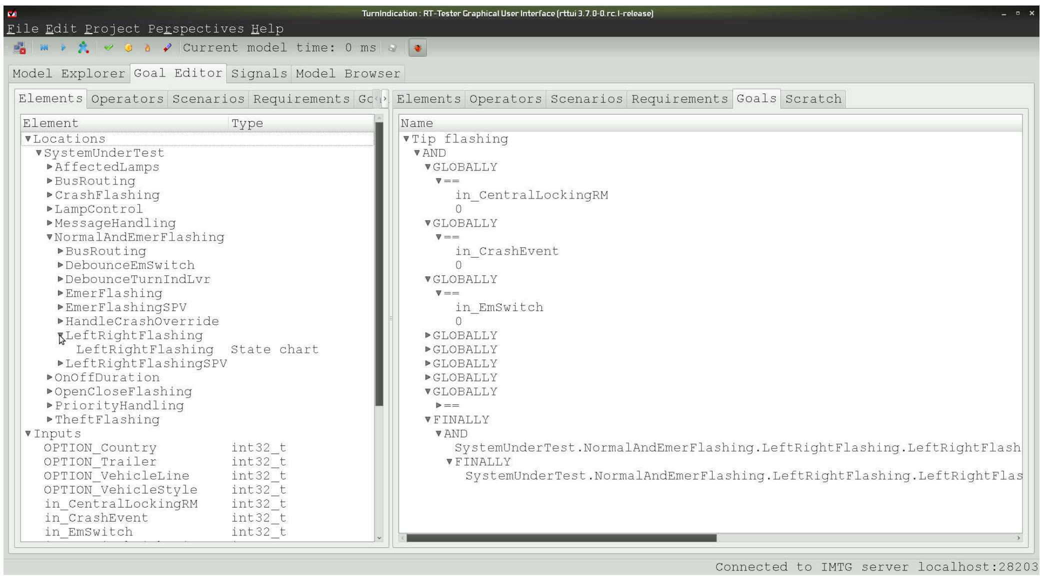
The potential solution is subsequently passed to the concrete interpreter, so that the solution can be refined into a computation ending in a stable state. The resulting computation is then added to the computation tree as a new branch of the selected source computation tree node.

Note, that back-tracking is explicitly disabled here. The selection of a source system state to be used as basis for the bounded model checking process is a wanted major decision for the user to inject domain expertise.

3.3 User Interface

This section complements the previous discussion of an interactive model-based test generation paradigm by providing a brief overview of how a corresponding user interface was implemented to function.

The described user interface widgets are readily available and represent the degree of interactivity currently supported by the model-based framework under consideration. Special emphasis is given to a prototypical widget used


TurnIndication : RT-Tester Graphical User Interface (rttui 3.7.0-0.rc.l-release)
File Edit Project Perspectives Help
Current model time: 0 ms
Model Explorer
Goal Editor
Signals
Model Browser
Elements
Operators
Scenarios
Requirements
Goals
Scratch
Connected to IMTG server localhost:28203

```
Element                        Type
▾Locations
  ▾SystemUnderTest
    ▸AffectedLamps
    ▸BusRouting
    ▸CrashFlashing
    ▸LampControl
    ▸MessageHandling
    ▾NormalAndEmerFlashing
      ▸BusRouting
      ▸DebounceEmSwitch
      ▸DebounceTurnIndLvr
      ▸EmerFlashing
      ▸EmerFlashingSPV
      ▸HandleCrashOverride
      ▾LeftRightFlashing
        LeftRightFlashing      State chart
      ▸LeftRightFlashingSPV
    ▸OnOffDuration
    ▸OpenCloseFlashing
    ▸PriorityHandling
    ▸TheftFlashing
▾Inputs
   OPTION_Country              int32_t
   OPTION_Trailer              int32_t
   OPTION_VehicleLine          int32_t
   OPTION_VehicleStyle         int32_t
   in_CentralLockingRM         int32_t
   in_CrashEvent               int32_t
   in_EmSwitch                 int32_t
```

```
Name
▾Tip flashing
  ▾AND
    ▾GLOBALLY
      ▾==
         in_CentralLockingRM
         0
    ▾GLOBALLY
      ▾==
         in_CrashEvent
         0
    ▾GLOBALLY
      ▾==
         in_EmSwitch
         0
    ▸GLOBALLY
    ▸GLOBALLY
    ▸GLOBALLY
    ▸GLOBALLY
    ▾GLOBALLY
      ▸==
    ▾FINALLY
      ▾AND
         SystemUnderTest.NormalAndEmerFlashing.LeftRightFlashing.LeftRightFlash
        ▾FINALLY
          SystemUnderTest.NormalAndEmerFlashing.LeftRightFlashing.LeftRightFlas
```

Figure 3.3: User Interface - Goal Editor

for creating and editing of LTL formulas.

3.3.1 Goal Editor

The Goal Editor Widget – as the name might suggest – is employed to edit LTL properties used for either model checking intermediate computation trees or for expanding them using generation goals. It is available during the entirety of the work-flow described in section 3.2.2.

As opposed to simply utilizing a text widget, the LTL editor presented here provides some noteworthy features. Figure 3.3 shows a typical screen-shot.

Using LTL to specify generation goal- or model checking properties for a large system under test is usually a highly complex undertaking, since it is rather difficult for practitioners to think of encoding all necessary pre- and side-constraints when postulating an intended property. Some generic tool-based reasons for this phenomenon have been given in section 3.2.1. However, some pragmatic reasons may solely be due to using simple text editors.

Real-world LTL properties tend to be deeply nested, precisely because in order to have specific expressive power they need to incorporate pre-conditions and invariants. While parentheses are a suitable means to allow machine parsing of such properties, they do not truly facilitate human readability. Rather, most users will – when reduced to utilizing a text editor – depend on line breaks and indentation to reflect the nested tree structure of any given LTL property. It is therefore a nearly self-evident course of action to move from text editors to a tree visualization reminiscent of the well-established Polish Notation.

Using tree widgets for the representation of LTL propositions within a graphical user interface allows for some additional advantages. Most notably, a tree view widget provides the possibility for a separate text column, which can be used for natural language annotations of LTL formula tree nodes. Annotations are very useful for orientation purposes whenever a (sub-)formula has to be (re-)used for or moved to new or modified formula tree vacancies.

Using a tree representation of LTL formulas with additional natural language annotations for tree nodes becomes especially powerful when combined with drag-and-drop and cut-copy-paste functionality of modern user interface frameworks.

The goal editor presented here is split into two identical data view halves displaying the same data model. This is done so as to encourage drag-and-drop operations. Each data view provides identical/mirrored tabs containing the basic building blocks for LTL propositions, i.e. inputs, outputs, variables, locations, timers, requirement LTL representations parsed from the test model and LTL- and predicate logic operators. The special-purpose tabs "Goals" and "Scratch" are the central writable tabs, which can be used for drag-and-drop construction of generation goals.

Additionally, the following LTL formula stubs and annotations are provided within tab "Scenarios":

- Reachability — Some condition eventually holds:
  $\mathbf{F}(f)$
- Invariant — Some condition always holds:
  $\mathbf{G}(f)$
- Never —Some condition never holds:
  $\mathbf{G}(\neg f)$
- Parallel composition —Conditions hold in parallel:
  $(f \ \wedge \ g)$

- Alternatives — At least one condition holds:
  $(f \;\lor\; g)$
- Choice — Exactly one condition holds:
  $((f \;\land\; \neg g) \;\lor\; (\neg f \;\land\; g)$
- Strict step sequence — Sequence of step conditions:
  $(f \;\land\; \mathbf{X}(g))$
- Strict path sequence — Sequence of multi-step conditions:
  $(f \;\land\; (f \;\mathbf{U}\; g))$
- Weak sequence — Sequence of intermittent conditions:
  $(f \;\land\; \mathbf{F}(g))$

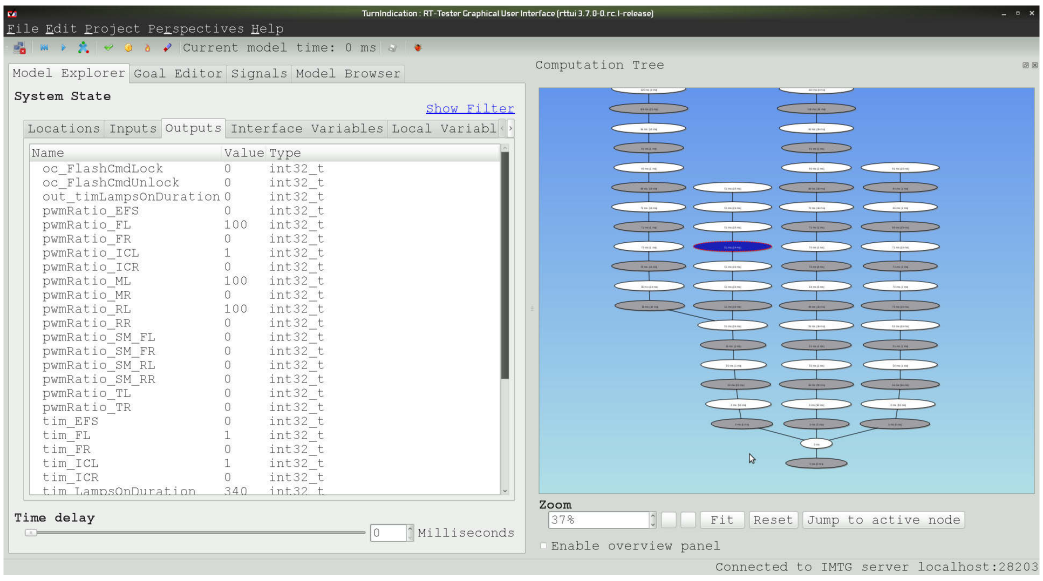
Again, the tab “Scenarios” is provided to facilitate editing of *LTL* propositions using drag-and-drop mechanisms. Each contained formula stub may be dragged into an incomplete generation goal or sub-formula to structure a proposition argument. Its unspecified operand sub-trees f and g in turn remain empty until specified.

3.3.2 Model Explorer

The visualization of a given computation trees and the system states it is comprised of is performed by the Model Explorer Widget. Its concepts, functionality and development considerations may be found in [Lan11]. It consists of a computation tree view and a system state view. Figure 3.4 contains a typical screen-shot of the Model Explorer Widget with a computation tree visualization in its right half and a system state visualization in its left half.

The computation tree visualization reflects the structure of the tree. Each node visualization contains only the current system time of the respective system state. Grey system states indicate unstable system states, white system states are stable.

Whenever the user selects a computation tree node from the computation tree visualization, the left half of the Model Explorer Widget is updated to contain all valuations of the corresponding system state. A user may hence inspect all system states of a computation tree by simply clicking on all nodes in the tree visualization.



Name	Value	Type
oc_FlashCmdLock	0	int32_t
oc_FlashCmdUnlock	0	int32_t
out_timLampsOnDuration	0	int32_t
pwmRatio_EFS	0	int32_t
pwmRatio_FL	100	int32_t
pwmRatio_FR	0	int32_t
pwmRatio_ICL	1	int32_t
pwmRatio_ICR	0	int32_t
pwmRatio_ML	100	int32_t
pwmRatio_MR	0	int32_t
pwmRatio_RL	100	int32_t
pwmRatio_RR	0	int32_t
pwmRatio_SM_FL	0	int32_t
pwmRatio_SM_FR	0	int32_t
pwmRatio_SM_RL	0	int32_t
pwmRatio_SM_RR	0	int32_t
pwmRatio_TL	0	int32_t
pwmRatio_TR	0	int32_t
tim_EFS	0	int32_t
tim_FL	1	int32_t
tim_FR	0	int32_t
tim_ICL	1	int32_t
tim_ICR	0	int32_t
tim_LampsOnDuration	340	int32_t

Time delay: 0 Milliseconds

Connected to IMTG server localhost:28203

Figure 3.4: User Interface - Model Explorer and Computation Tree

Additionally, a user may modify input- or global system time valuations for stable system states within the system state view. This causes a modified system state clone to be created, and the Model Explorer Widget is blocked until either the clone is finalized and interpreted or discarded.

The toolbar displayed above the computation tree- and system state view contains all commands used to evolve an intermediate computation tree. However, all computation tree manipulation commands depend on the selection of a current user-selected computation tree node. The user may select such a node by double-clicking on it. As a result, the selected node is distinguished using a blue background.

Finally, figure 3.5 provides a closer look at the computation tree view of the Model Explorer Widget. Note, that multiple traces may be highlighted by use of red node edges. As such, figure 3.5 gives an example of how model checking results of LTL properties against an intermediate computation tree are visualized.

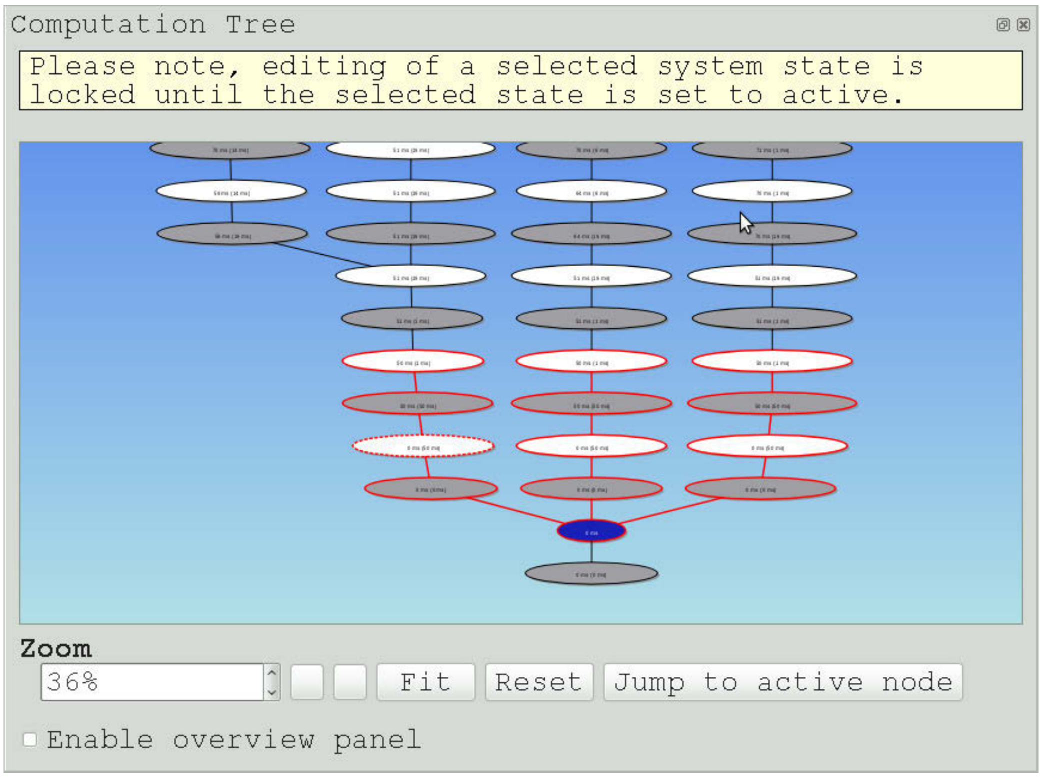

Computation Tree
Please note, editing of a selected system state is locked until the selected state is set to active.
Zoom
36%
Fit
Reset
Jump to active node
Enable overview panel

Figure 3.5: User Interface - Model Checking Results

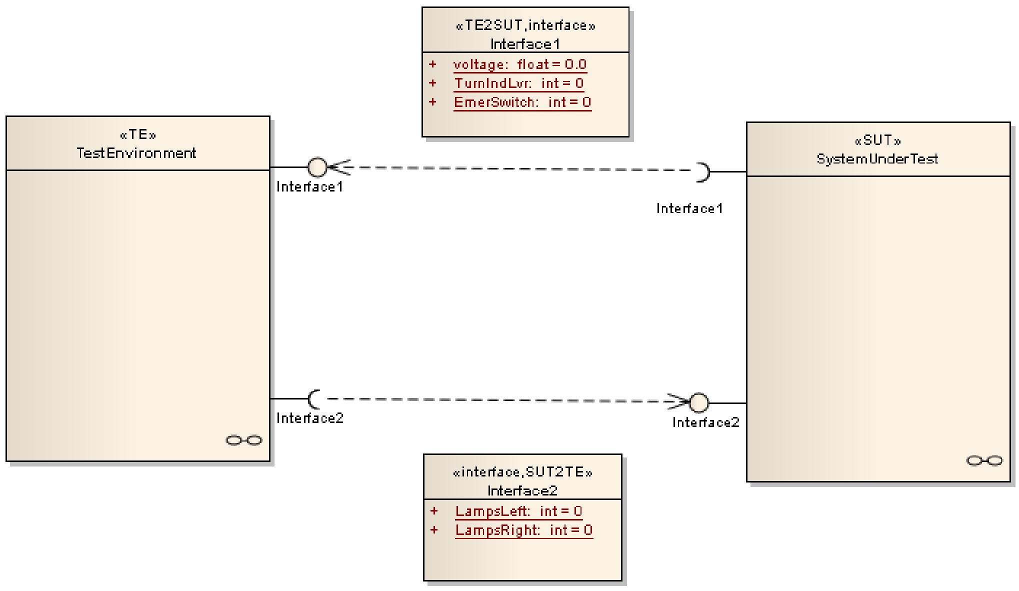


Figure 3.6: Turn Indicator - System Overview

3.4 Case Study - Turn Indication

This section provides a small example test model and a selection of matching test scenarios. For each scenario, different possible approaches for generating test data are discussed in order to illuminate, how the interactive test generation paradigm can be used depending on the user's preference and domain knowledge.

3.4.1 Test Model

The system under test used in this case study is an example turn indication automotive controller. Its inputs consist of battery voltage, turn indication lever position and emergency switch position. As outputs, the system computes lamp commands for the left and right side turn indicators. Figure 3.6 shows the interface stimuli and observables of the controller.

Internally, the system is split into two sub-systems running in parallel. Subsystem FLASH_CTRL handles priorities between directional flashing and emergency flashing. Emergency flashing will override directional flashing. However, if the turn indication lever is moved into a setting different from

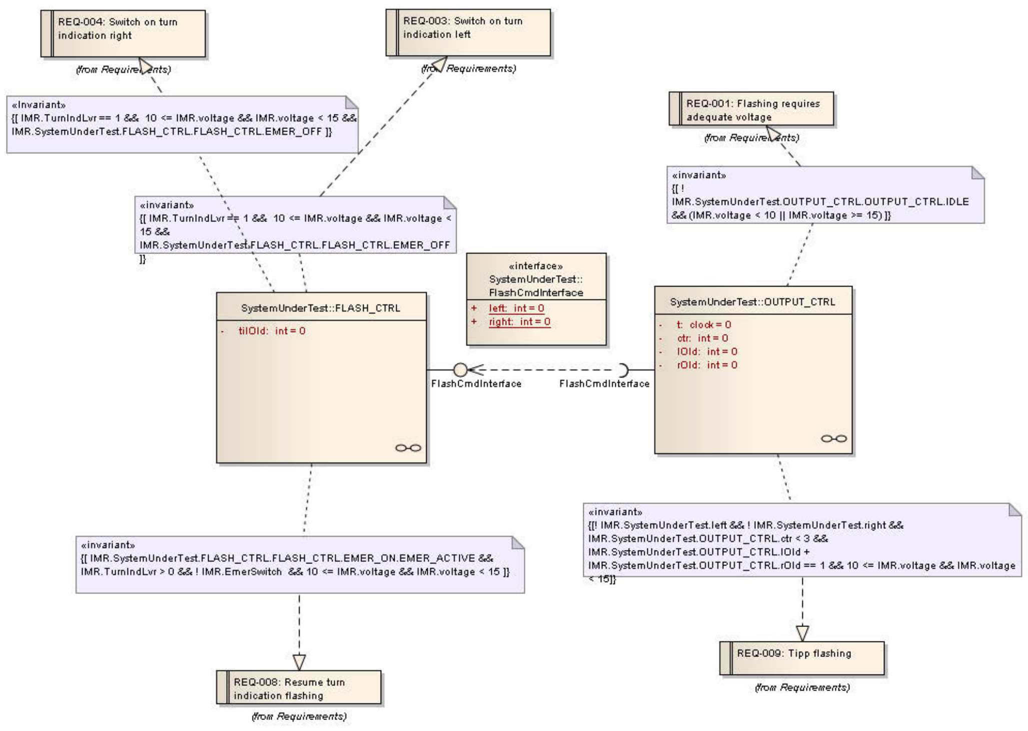
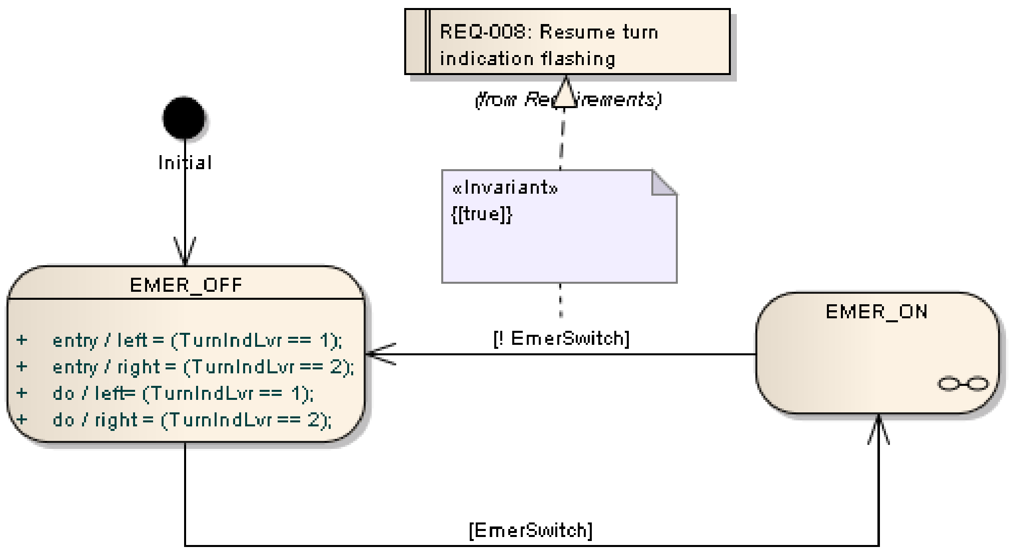

REQ-004: Switch on turn indication right
(from Requirements)
REQ-003: Switch on turn indication left
(from Requirements)
REQ-001: Flashing requires adequate voltage
(from Requirements)
«Invariant»
{[IMR.TurnIndLvr == 1 && 10 <= IMR.voltage && IMR.voltage < 15 && IMR.SystemUnderTest.FLASH_CTRL.FLASH_CTRL.EMER_OFF]}
«invariant»
{[IMR.TurnIndLvr == 1 && 10 <= IMR.voltage && IMR.voltage < 15 && IMR.SystemUnderTest.FLASH_CTRL.FLASH_CTRL.EMER_OFF]}
«invariant»
{[! IMR.SystemUnderTest.OUTPUT_CTRL.OUTPUT_CTRL.IDLE && (IMR.voltage < 10 || IMR.voltage >= 15)]}
«interface»
SystemUnderTest::FlashCmdInterface
+ left: int = 0
+ right: int = 0
SystemUnderTest::FLASH_CTRL
- tilOld: int = 0
FlashCmdInterface
FlashCmdInterface
SystemUnderTest::OUTPUT_CTRL
- t: clock = 0
- ctr: int = 0
- lOld: int = 0
- rOld: int = 0
«invariant»
{[IMR.SystemUnderTest.FLASH_CTRL.FLASH_CTRL.EMER_ON.EMER_ACTIVE && IMR.TurnIndLvr > 0 && ! IMR.EmerSwitch && 10 <= IMR.voltage && IMR.voltage < 15]}
«invariant»
{[! IMR.SystemUnderTest.left && ! IMR.SystemUnderTest.right && IMR.SystemUnderTest.OUTPUT_CTRL.ctr < 3 && IMR.SystemUnderTest.OUTPUT_CTRL.lOld + IMR.SystemUnderTest.OUTPUT_CTRL.rOld == 1 && 10 <= IMR.voltage && IMR.voltage < 15]}
REQ-008: Resume turn indication flashing
(from Requirements)
REQ-009: Tipp flashing
(from Requirements)

Figure 3.7: Turn Indicator - System Under Test

its position when emergency flashing began, direction flashing may again override emergency flashing.

As a result, the sub-system FLASH_CTRL computes on/off statuses for left and right side turn indicators and passes these to sub-system OUTPUT_CTRL. Sub-system OUTPUT_CTRL is responsible for transforming on/off statuses for left and right turn indicators into concrete output commands with corresponding timed on/off sequences. Additionally, sub-system OUTPUT_CTRL is responsible for realizing tip flashing. Tip flashing is activated whenever flashing is enabled for a brief period of time and ensures, that every detected flashing state will result in at least three turn indicator on/off sequences. Figure 3.7 shows the decomposition into sub-systems.

Sub-system FLASH_CTRL is modeled using a state chart with two states. In state EMER_OFF, no emergency flashing is active and turn indication statuses are calculated simply as a function of the turn indication lever position. State EMER_ON is reached once the emergency switch is pressed. Figure 3.8 shows the state chart.



REQ-008: Resume turn indication flashing
(from Requirements)
Initial
«Invariant»
{[true]}
EMER_OFF
+ entry / left = (TurnIndLvr == 1);
+ entry / right = (TurnIndLvr == 2);
+ do / left= (TurnIndLvr == 1);
+ do / right = (TurnIndLvr == 2);
[! EmerSwitch]
EMER_ON
[EmerSwitch]

Figure 3.8: Turn Indicator - Flash Control

State EMER_ON is a hierarchic state and contains the state chart shown in figure 3.9. In sub-state EMER_ACTIVE turn indication statuses are set to one as a result of activated emergency flashing. Sub-state EMER_OVERRIDE is reached, if the turn indication lever position is changed to a new direction. As a result, emergency flashing is overridden by directional flashing, and turn indication statuses are again computed using the turn indication lever position.

Sub-system OUTPUT_CTRL consists of the state chart shown in figure 3.10. In state IDLE no flashing takes place. State FLASHING performs flashing only, if it is indicated by the turn indication statuses calculated in sub-system FLASH_CTRL, and if the battery voltage is within a valid range. The state chart can transition from FLASHING back to IDLE if either the battery voltage becomes invalid, or both side flashing statuses return to zero. However, in the latter case the sub-system ensures, that three mandatory tip flashing cycles are observed.

State FLASHING is again hierarchic and consists of the state chart shown in figure 3.11. It contains states ON and OFF, which realize the concrete turn indicator on and off periods of 340ms and 320ms respectively.

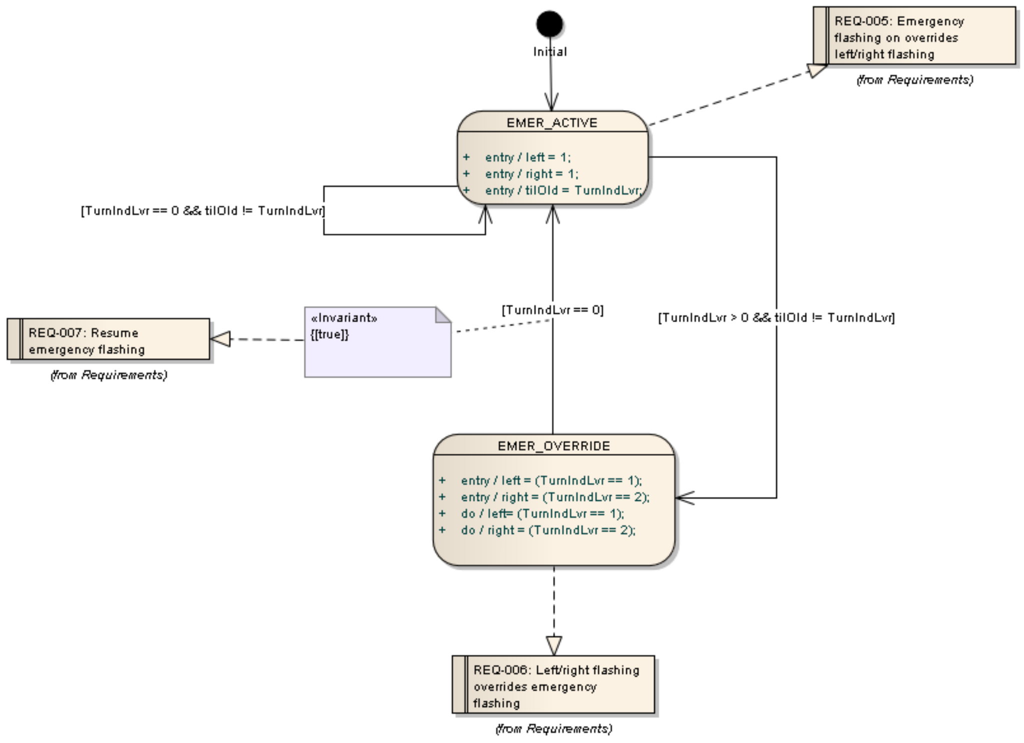

REQ-005: Emergency flashing on overrides left/right flashing
(from Requirements)
Initial
EMER_ACTIVE
+ entry / left = 1;
+ entry / right = 1;
+ entry / tilOld = TurnIndLvr;
[TurnIndLvr == 0 && tilOld != TurnIndLvr]
[TurnIndLvr == 0]
[TurnIndLvr > 0 && tilOld != TurnIndLvr]
REQ-007: Resume emergency flashing
(from Requirements)
«Invariant»
{[true]}
EMER_OVERRIDE
+ entry / left = (TurnIndLvr == 1);
+ entry / right = (TurnIndLvr == 2);
+ do / left= (TurnIndLvr == 1);
+ do / right = (TurnIndLvr == 2);
REQ-006: Left/right flashing overrides emergency flashing
(from Requirements)

Figure 3.9: Turn Indicator - Flash Control - Emergency On

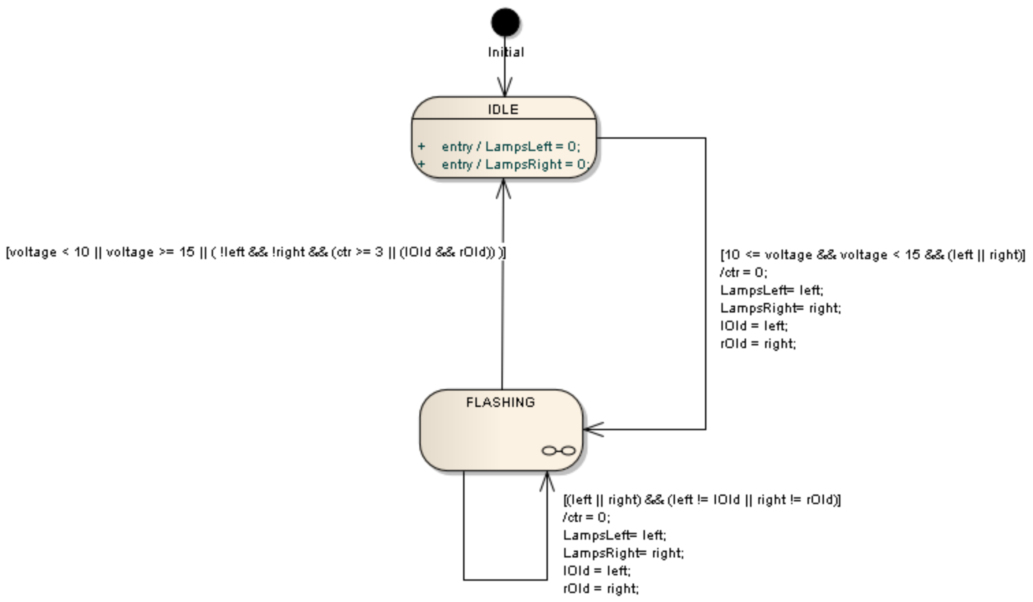

Initial
IDLE
+ entry / LampsLeft = 0;
+ entry / LampsRight = 0;
[voltage < 10 || voltage >= 15 || (!left && !right && (ctr >= 3 || (lOld && rOld)))]
[10 <= voltage && voltage < 15 && (left || right)]
/ctr = 0;
LampsLeft= left;
LampsRight= right;
lOld = left;
rOld = right;
FLASHING
[(left || right) && (left != lOld || right != rOld)]
/ctr = 0;
LampsLeft= left;
LampsRight= right;
lOld = left;
rOld = right;

Figure 3.10: Turn Indicator - Output Control

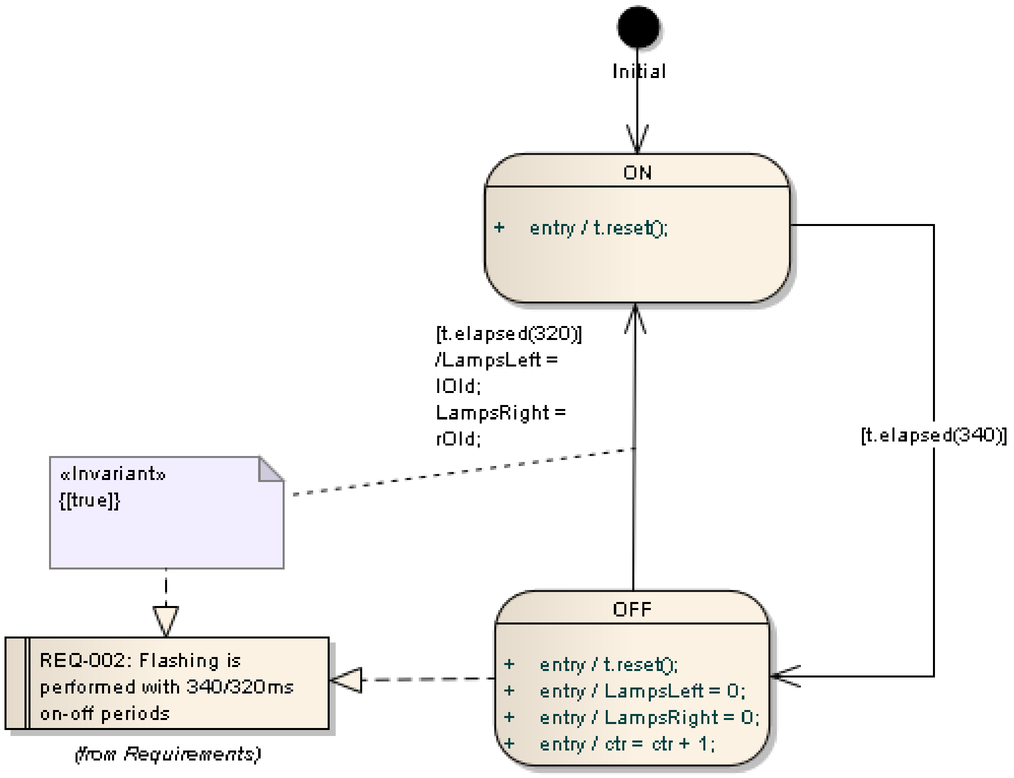


Figure 3.11: Turn Indicator - Output Control - Flashing

3.4.2 Example Scenarios

Using the test model described above, we can now generate computations in a computation tree, which in turn might be refined into executable test procedures. The following subsections give examples of different test scenarios and different approaches in constructing suitable computations.

Stable Flashing

Consider creating a test procedure, which exercises stable direction left side flashing. No emergency flashing should take place, and the battery voltage should remain in range. The following LTL generation goal might be used.

$$\begin{aligned}
&(\mathbf{G}\ ((\mathit{voltage} \geq 10) \wedge (\mathit{voltage} < 15))) \wedge \\
&(\mathbf{G}\ (\mathit{EmerSwitch} = 0)) \wedge \\
&(\mathbf{G}\ (\mathit{TurnIndLvr} \neq 2)) \wedge \\
&(\mathbf{F}\ (\mathit{OUTPUT\_CTRL} :: \mathit{FLASHING} :: \mathit{ON} \wedge \\
&\quad (\mathbf{F}\ (\mathit{OUTPUT\_CTRL} :: \mathit{IDLE} \wedge \\
&\qquad (\mathit{OUTPUT\_CTRL} :: \mathit{ctr} > 3)))))
\end{aligned}$$

The goal firstly enforces, that the voltage remains in a valid range during the entire computation. Secondly, the goal disables any movement of the emergency switch. Additionally, the turn indication lever position is restricted to be in the neutral or left position. Finally, the goal requires, that eventually state OUTPUT_CTRL::FLASHING::ON is reached, that after that state OUTPUT_CTRL::IDLE is reached, and that OUTPUT_CTRL::ctr has a value above the tip flashing threshold.

The generation results yield the following timed input assignment sequence:

1. System time: 0 ms
    - $\mathit{EmerSwitch} \mapsto 0$
    - $\mathit{TurnIndLvr} \mapsto 1$
    - $\mathit{voltage} \mapsto 12.0$
2. System time: 2320 ms
    - $\mathit{EmerSwitch} \mapsto 0$

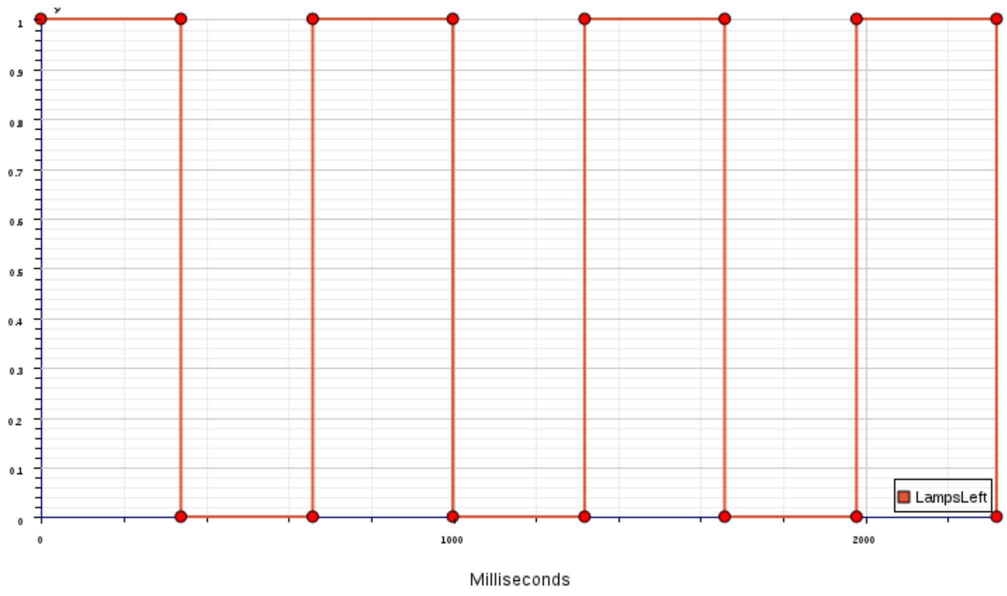


Figure 3.12: Stable Flashing Signal View

- $TurnIndLvr \mapsto 0$
- $voltage \mapsto 12.0$

Figure 3.12 shows a signal graph for the left side turn indicator. The generated computation successfully causes left side flashes for four cycles before turning it off again.

A more interactive approach to generating the same computation might be to generate a partial computation enabling left side flashing using a simpler generation goal and then manually letting time elapse, until tip flashing is no longer active. A suitable initial generation goal might be:

$$\mathbf{F}\ ((LampsLeft = 1) \wedge \\ (LampsRight = 0))$$

The resulting computation contains only the initial input assignments to initiate left side flashing:

1. System time: 0 ms
   - $EmerSwitch \mapsto 0$

- $TurnIndLvr \mapsto 1$
- $voltage \mapsto 12.0$

Using unchanged input assignments, the user can then let four flashing cycle periods elapse manually. Finally, the user might reset the turn indication lever position:

1. System time: 2320 ms
   - $EmerSwitch \mapsto 0$
   - $TurnIndLvr \mapsto 0$
   - $voltage \mapsto 12.0$

Finally, a completely manual approach can also be utilized. A user might create the entire computation by manually turning left side flashing on and off again.

Tip Flashing Direction Change

Consider a scenario, where the system is in stable left flashing for five cycles, then the left stable flashing is overridden by a right flashing request. The turn indication lever is subsequently returned to the neutral position in order to exercise right side tip flashing.

The following generation goal postulates this scenario:

$$\begin{array}{l}(\mathbf{G}\ ((voltage \geq 10) \wedge (voltage < 15))) \wedge \\ (\mathbf{G}\ (EmerSwitch = 0)) \wedge \\ (\mathbf{F}\ (OUTPUT\_CTRL :: FLASHING :: ON \wedge \\ \quad (LampsLeft = 1) \wedge \\ \quad (OUTPUT\_CTRL :: ctr \geq 5) \\ \quad \mathbf{F}\ (OUTPUT\_CTRL :: FLASHING :: ON \wedge \\ \quad\quad (LampsRight = 1) \wedge \\ \quad\quad \mathbf{F}\ (OUTPUT\_CTRL :: IDLE))))\end{array}$$

Again, voltage is always kept in a valid range and the emergency switch is locked into off position. The remaining property contains multiple applications of the weak path sequence property stub $f\ \wedge \mathbf{F}\ (g)$. Stable flashing is enforced by requiring a large enough value of $OUTPUT\_CTRL :: ctr$ during left flashing. Tip flashing is implicitly postulated for right flashing by omitting a similar $OUTPUT\_CTRL :: ctr$ constraint.

The generation process yields the following computation:

1. System time: 0 ms
   - $EmerSwitch \mapsto 0$
   - $TurnIndLvr \mapsto 1$
   - $voltage \mapsto 12.0$
2. System time: 3300 ms
   - $EmerSwitch \mapsto 0$
   - $TurnIndLvr \mapsto 2$
   - $voltage \mapsto 12.0$
3. System time: 4620 ms
   - $EmerSwitch \mapsto 0$
   - $TurnIndLvr \mapsto 0$
   - $voltage \mapsto 12.0$

Figure 3.13 shows the corresponding outputs as a signal graphs. The first graph indicates the turn indication lever position. The second and third graphs show the left and right turn indicator values respectively.

Again, the same computation might be constructed using multiple generation steps. And again, the entire computation might be reproduced manually. However, a more sensible approach might be to use multiple simpler generation goals:

1. Generate stable left flashing

$$\begin{aligned}
&(\mathbf{G}\ ((voltage \geq 10) \wedge (voltage < 15))) \wedge \\
&(\mathbf{G}\ (EmerSwitch = 0)) \wedge \\
&(\mathbf{F}\ (OUTPUT\_CTRL :: FLASHING :: ON \wedge \\
&\quad (LampsLeft = 1) \wedge \\
&\quad (OUTPUT\_CTRL :: ctr \geq 5)))
\end{aligned}$$

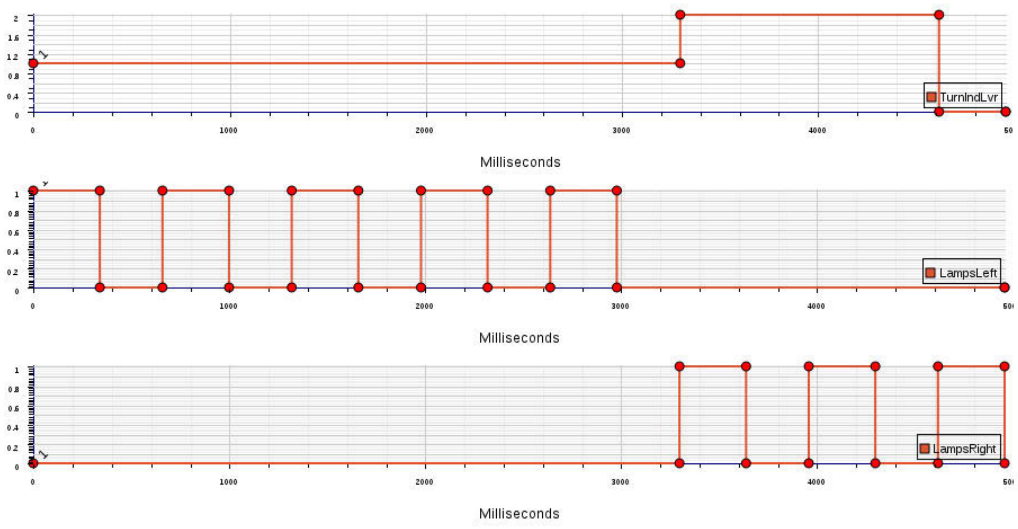


Figure 3.13: Tip Flashing Override Signal View

2. Generate right tip flashing

$$\begin{aligned}&(\mathbf{G}\ ((voltage \geq 10) \wedge (voltage < 15))) \wedge\\&(\mathbf{G}\ (EmerSwitch = 0)) \wedge\\&(\mathbf{F}\ (OUTPUT\_CTRL :: FLASHING :: ON \wedge\\&\quad (LampsRight = 1)\end{aligned}$$

3. Generate flashing idle

$$\begin{aligned}&(\mathbf{G}\ ((voltage \geq 10) \wedge (voltage < 15))) \wedge\\&(\mathbf{F}\ (OUTPUT\_CTRL :: IDLE))\end{aligned}$$

Emergency Flashing Direction Override

In this scenario the system is driven into left directional flashing. Next, the emergency switch is activated in order to override the directional flashing. Finally, the turn indication lever is moved from left position to right position in order to override emergency flashing by directional flashing again.

The following generation goal might be used in order to calculate a suitable computation:

$$
\begin{array}{l}
(\mathbf{G}\ ((voltage \geq 10) \wedge (voltage < 15))) \wedge \\
(\mathbf{G}\ (EmerSwitch = 0)) \wedge \\
(\mathbf{F}\ (OUTPUT\_CTRL :: FLASHING :: ON \wedge \\
\quad FLASH\_CTRL :: EMER\_OFF \wedge \\
\quad (OUTPUT\_CTRL :: ctr \geq 3) \\
\quad \mathbf{F}\ (OUTPUT\_CTRL :: FLASHING :: ON \wedge \\
\qquad FLASH\_CTRL :: EMER\_ON :: EMER\_ACTIVE \wedge \\
\qquad (OUTPUT\_CTRL :: ctr \geq 3) \\
\qquad \mathbf{F}\ (OUTPUT\_CTRL :: FLASHING :: ON \wedge \\
\qquad\quad FLASH\_CTRL :: EMER\_ON :: EMER\_OVERRIDE \wedge \\
\qquad\quad (OUTPUT\_CTRL :: ctr \geq 3) \\
\qquad\quad \mathbf{F}\ (OUTPUT\_CTRL :: IDLE)))))
\end{array}
$$

The resulting computation looks as follows:

1. System time: 0 ms
   - $EmerSwitch \mapsto 0$
   - $TurnIndLvr \mapsto 1$
   - $voltage \mapsto 12.0$
2. System time: 1980 ms
   - $EmerSwitch \mapsto 1$
   - $TurnIndLvr \mapsto 1$
   - $voltage \mapsto 12.0$
3. System time: 3960 ms
   - $EmerSwitch \mapsto 1$
   - $TurnIndLvr \mapsto 2$
   - $voltage \mapsto 12.0$
4. System time: 5620 ms
   - $EmerSwitch \mapsto 0$
   - $TurnIndLvr \mapsto 0$
   - $voltage \mapsto 12.0$

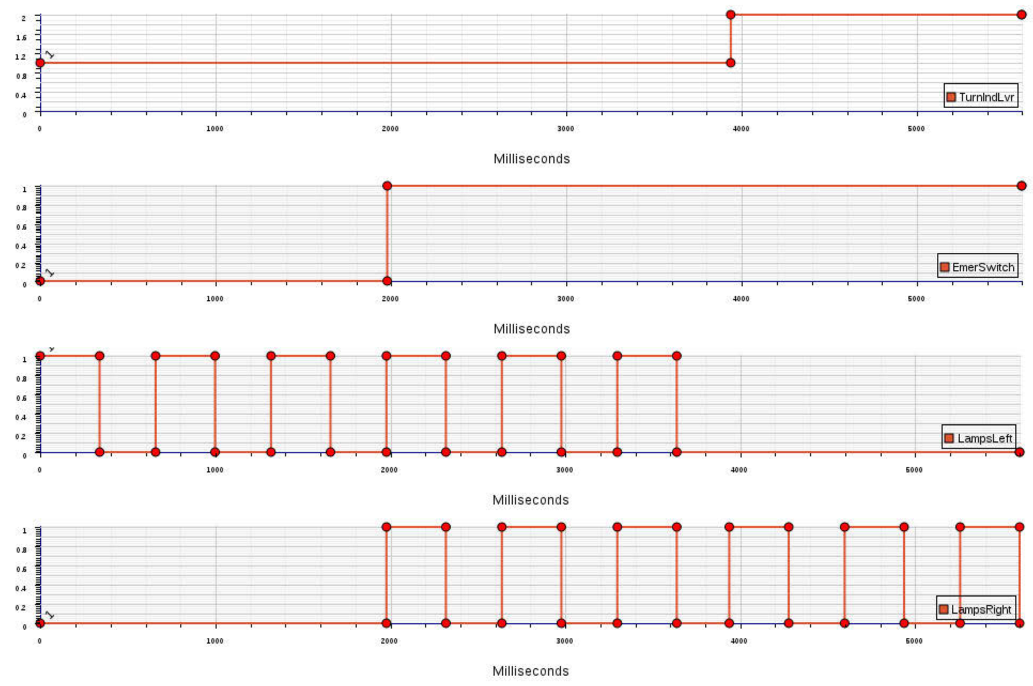


Figure 3.14: Emergency Flashing Direction Override Signal View

Figure 3.14 shows signal graphs to visualize the computation. The first graph shows the turn indication lever movement. The second graph shows the emergency switch position. The third and fourth graphs show left and right turn indications respectively.

Again, the same computation can be created using a range of different approaches. A user might perform multiple manual input assignments or have the generator solve a sequence of smaller generation goals.

Note, that the creation of this computation can be simplified if the previous scenarios have already been created. A user does not have to generate each scenario starting from scratch – the initial state of the test model. Rather, an existing computation tree might be re-used.

Given an (intermediate) computation tree generated using one of the above scenarios, a system state already providing stable left flashing may be identified and used as a pre-condition for simpler subsequent generation steps. The following LTL property might yield a suitable state:

$$\begin{aligned}
&OUTPUT\_CTRL :: FLASHING :: ON \wedge \\
&FLASH\_CTRL :: EMER\_OFF \wedge \\
&(LampsLeft = 1) \wedge \\
&(OUTPUT\_CTRL :: ctr \geq 3)
\end{aligned}$$

Using such a state as the user-selected basis for further generation steps, the following simpler generation goal completes the computation:

$$\begin{aligned}
&(\mathbf{G}\ ((voltage \geq 10) \wedge (voltage < 15))) \wedge \\
&(\mathbf{G}\ (EmerSwitch = 0)) \wedge \\
&(\mathbf{F}\ (OUTPUT\_CTRL :: FLASHING :: ON \wedge \\
&\quad FLASH\_CTRL :: EMER\_ON :: EMER\_ACTIVE \wedge \\
&\quad (OUTPUT\_CTRL :: ctr \geq 3) \\
&\quad \mathbf{F}\ (OUTPUT\_CTRL :: FLASHING :: ON \wedge \\
&\qquad FLASH\_CTRL :: EMER\_ON :: EMER\_OVERRIDE \wedge \\
&\qquad (OUTPUT\_CTRL :: ctr \geq 3) \\
&\qquad \mathbf{F}\ (OUTPUT\_CTRL :: IDLE))))
\end{aligned}$$

3.5 Evaluation

This chapter introduced a modification to the work-flow of an existing framework for model-based test generation. This modification was performed in order to enable an expert to interactively infuse application domain knowledge into the test generation process. As a result, the presented modified framework is well suited for the development of test cases corresponding to specific application scenarios.

Several algorithms, tools and publications from the scientific community can be found, which examine different aspects of model-based testing, scenario testing and user interaction paradigms within test generation processes. However, the combination presented in this thesis seems to be unique.

In [RK11] system requirements are formalized as test scenarios, which in turn are transformed into Petri nets. System-level tests are then automatically generated from the resulting formal models.

[AQ13] perform model-based testing as well, but focuses on utilizing scenarios for regression testing of functional differences between specific software versions. Petri nets are used to analyze the particular differences between baselines.

[DKT08] examine B machine models as a basis for fully automated test generation. The fully automated test generation approach is then extended to include test scenarios to guide the generation process.

In [DCT12] customized regular expressions are used to generate tests on the basis of behavioral B machine models. Scenarios can then be executed on the test model in order to establish expected results.

In [CDJ11], the authors present an approach to generating test suites based on UML behavioral models additionally constrained using OCL. The publication examines the power of a completely automated generation approach and concludes, that test scenarios expressed as regular expressions are useful to complement automatically generated functional tests.

[MLL09] describe a model-based testing approach based on Event-B test models. Test scenarios are specified as CSP expressions, and are in turn used to generate test cases. The authors show, how test scenarios can automatically be adapted to changes in the test model.

[LK01] introduce an approach on using UML sequence diagrams to formalize functional and real-time requirements as scenarios. The paper presents an extension to a UML CASE-tool, which provides support for continuously testing a system implementation against specified scenarios during the entire software development process.

[CDKM11] and [BW05] utilize interactive theorem provers to facilitate the generation of tests in order to verify programs on a unit testing integration level. Single test cases are used to verify simple properties, while the theorem provers are used to judge the state of proofs for higher-level properties. Domain expertise is used to interactively guide the overall proof evolution.

In [GKP00] the authors present an interactive test generation approach for sequential or concurrent programs. The generation process prepares visualizations of control flow graphs and (interleavings of) paths through the control flow. The user can then interactively select and edit paths to be refined into test cases.

[MFT12] propose a concept for a generic work-flow, which couples search-based software testing approaches with the ability to have domain specialists interact with the generation process. The concept is focused on separating software engineering concerns from domain knowledge useful during testing.

Chapter 4

Conclusion

There exists a wide variety of graphical specification formalisms used to specify embedded systems. While several formalisms such as UML are widely used and have become standardized, they cannot completely eliminate the need for domain specific specification languages.

While at first glance it may appear to be an unnecessary effort to develop a complete specification formalism just for a specific application domain, it is sometimes more efficient to do so, than to attempt to force an application specification into a standardized specification formalism, which in some cases might well be too complex and ill-suited to the task. Designing a specification formalism and tailoring its semantics for a specific task is often the preferred option, and this choice impacts the field of model-based software development in general, and model-based testing and model checking in particular.

This thesis elaborated challenges in model-based testing from both ends of the spectrum. The first part of the thesis considered model-based testing and model checking for a domain specific specification formalism. The second part evaluated and extended a general purpose model-based testing approach for standardized specification formalisms.

4.1 Timed Moore Automata

In the first part of the thesis, Timed Moore Automata, a specification formalism used for modeling embedded systems were introduced. The formalism is used in real-world applications from the railway domain, particularly for modeling the behavior of (components of) level crossings. As such, systems specified using the Timed Moore Automata formalism are safety-critical, and measures to verify their behavior need to be taken.

Timed Moore Automata are an extension of the classical Moore Automata. Just as classical Moore Automata, they are finite state machines, for which the outputs are only dependent on the location the automaton resides in. However, with respect to classical Moore Automata, a number of features have been added to Timed Moore Automata.

While classical Moore Automata view their inputs as queues of events, Timed Moore Automata process inputs asynchronously as a vector of Boolean input valuations. This, in turn, induces a run-to-completion semantics for Timed Automata. They perform discrete transitions immediately and for as long as such transitions are enabled. New input valuations are only accepted while an automaton rests (i.e. performs a delay transition).

The eponymous addition to Timed Moore Automata in comparison to classical Moore Automata is the introduction of abstract timers into the formalism. Within Timed Moore Automata, abstract timers do not entail concrete time durations, after which they must elapse. Instead, they are realized as special kinds of inputs and outputs of automata, which are interdependent and restricted in their evolution of valuations to formalize the abstract notions of *timer running*, *timer elapsed* (as inputs to the automaton) and *timer started*, *timer stopped* (as outputs of the automaton).

The thesis presented approaches and algorithms to performing model-checking and test data generation for test models specified as Timed Moore Automata.

Explicit model checking Timed Moore Automata against CTL properties was done by constructing Kripke structures and employing separate algorithms for each unique CTL operator pairing to label Kripke states with fulfilling (sub-)properties. This approach was made more efficient by introduction of delayed non-determinism into the automata's execution semantics in order to reduce the number of Kripke states needed.

As a practical application, the thesis showed how Timed Moore Automata can be checked to be live-lock free. Special consideration was given to the execution semantics to show, that this task can be achieved by reducing the property in question (**AF** *idle*) to the much cheaper (**EF** *idle*).

While in theory, the generation of test data for Timed Moore Automata can be achieved by using the model checking algorithms summarized above to show the existence of executions, which fulfill reachability properties, the thesis provided a specialized approach to test data generation.

Test data generation was done by (1) selection of a target trace through an automaton, for which test data is to be generated, (2) formulating predicates over input-, output-, timer status- and timer action variables, which enforce a trace with a maximum number of intermittent stable states, and (3) using a SAT constraint solver to produce a concrete test input sequence.

Selection of target traces to be enforced by generated test data was shown to be the determining factor with respect to model-coverage criteria achievable by this approach. In practice, the transition cover known from [Cho78] is used for the selection of target traces in order to achieve branch coverage.

Several open questions on Timed Moore Automata warrant further research. In practice, railway control systems are specified as entire collections of Timed Moore Automata running in parallel. While it seems entirely possible to extend the semantics given thus far to accommodate suitable interleavings of automata execution, this would branch the formalism from its real-world application. In currently existing systems, Timed Moore Automata trigger connected neighbor automata executions whenever any outputs change, that serve as inputs to the respective neighbor. This usually causes cascades of triggered automata executions, and research on a model checking approach to validate live-lock freedom from automata cyclically triggering each other would be interesting in theory and in practice.

4.2 Interactive Model-Based Testing

The second part of the thesis described an existing framework for the automatic model-based test generation for larger scale test models. The framework utilizes an internal intermediate model representation to accommodate multiple standardized modeling formalisms. As such, it serves as a basis to derive the transition relation of a test model to formalize its behavior.

Test generation goals are specified using LTL formulas over model elements. The test generation process then combines the transition relation of the test model with a generation goal by unrolling each into a predicate encapsulating multiple test model execution steps, which fulfill the given generation goal. In effect, this yields a bounded model checking problem instance, which in turn is passed to an SMT-Solver.

Since within an invocation of the generation framework there are usually multiple generation goals to consider, results of the generation process are not merely a sequence of timed input assignments, but rather an entire tree of system computation states. Within the generation process, the computation tree allows reuse of previous generation results, and is eventually refined into individual executable test procedures.

This fully automatic generation control flow contains some weaknesses with respect to readability and maintainability of generation goals as well as generation results. Therefore, the thesis introduced a modification to the generation control flow, which allows the user to interactively visualize and construct a computation tree, while still being able to utilize the automated generation capabilities of the framework. This allows the user to inject domain knowledge and thereby guide the generation process.

Pragmatically, the thesis introduced a new interactive generation work-flow and a corresponding user interface, which was designed and implemented specifically to interface with the generation framework. A scientific test model depicting an automotive controller was used to develop various testable scenarios, and to demonstrate exemplary use of the interactive generation paradigm.

The thesis leaves some unanswered questions with regard to a more complete integration of the interactive control flow with the generation framework.

While within the fully automated generation paradigm a testable requirement will correspond to a single generation goal, this is not the case in the interactive generation paradigm. Instead, the user may utilize different techniques and multiple generation (sub-)goals to expand a computation tree.

As a consequence, it would be necessary to bookkeep entire sequences of interactions in order to achieve requirements tracing in the interactive generation approach. Currently, only LTL generation (sub-)goals are stored persistently.

Generally, it would be useful to research an approach to expressing a sequence of user interactions as a single LTL property. This would allow a user to develop useful scenario tests within the interactive generation work-flow, and to store them for later reuse within the fully automatic paradigm. This would promote a more seamless interaction between both paradigms.

List of Figures

List of Tables

Bibliography

[AADO00] A. Abdurazik, P. Ammann, Wei Ding, and J. Offutt. Evaluation of three specification-based testing criteria. In *Engineering of Complex Computer Systems, 2000. ICECCS 2000. Proceedings. Sixth IEEE International Conference on*, pages 179–187, 2000.

[AB00] Paul E. Ammann and Paul E. Black. Test generation and recognition with formal methods. In *The First International Workshop on Automated Program Analysis, Testing and Verification*, pages 64–67, 2000.

[ABLN06] J.H. Andrews, L.C. Briand, Y. Labiche, and A.S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *Software Engineering, IEEE Transactions on*, 32(8):608–624, Aug 2006.

[ABM98] Paul E. Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In *In Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98*, pages 46–54. IEEE Computer Society, 1998.

[ABuR$^+$07] Shaukat Ali, Lionel C. Briand, Muhammad Jaffar ur Rehman, Hajra Asghar, Muhammad Zohaib Z. Iqbal, and Aamer Nadeem. A state-based approach to integration testing based on {UML} models. *Information and Software Technology*, 49(1112):1087 – 1106, 2007.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[APW08] Emine G. Aydal, Richard F. Paige, and Jim Woodcock. Observations for assertion-based scenarios in the context of model validation and extension to test case generation. In *ICST Workshops*, pages 11–20. IEEE Computer Society, 2008.

[AQ13] Farooq Ahmad and Zahid Hussain Qaisar. Scenario based functional regression testing using petri net models. In *ICMLA (2)*, pages 572–577. IEEE, 2013.

[AS05] B.K. Aichernig and P.A.P. Salas. Test case generation by ocl mutation and constraint solving. In *Quality Software, 2005. (QSIC 2005). Fifth International Conference on*, pages 64–71, Sept 2005.

[ATF09] Wasif Afzal, Richard Torkar, and Robert Feldt. A systematic review of search-based testing for non-functional system properties. *Inf. Softw. Technol.*, 51(6):957–976, June 2009.

[Bar03] Chitta Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving.* Cambridge University Press, New York, NY, USA, 2003.

[BBM02] Francesca Basanieri, Antonia Bertolino, and Eda Marchetti. The cow_suite approach to planning and deriving test suites in uml projects. In Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook, editors, *UML 2002 — The Unified Modeling Language*, volume 2460 of *Lecture Notes in Computer Science*, pages 383–397. Springer Berlin Heidelberg, 2002.

[BCES10] David Basin, Manuel Clavel, Marina Egea, and Michael Schläpfer. Automatic generation of smart, security-aware gui models. In Fabio Massacci, Dan Wallach, and Nicola Zannone, editors, *Engineering Secure Software and Systems*, volume 5965 of *Lecture Notes in Computer Science*, pages 201–217. Springer Berlin Heidelberg, 2010.

[BDL05] David Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security. In Manfred Broy, Johannes Grünbauer, David Harel, and Tony Hoare, editors, *Engineering Theories of Software Intensive Systems*, volume 195 of *NATO Science Series*, pages 353–398. Springer Netherlands, 2005.

[BDL$^+$06] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Håkansson, Paul Pettersson, Wang Yi 0001, and Martijn Hendriks. Uppaal 4.0. In *QEST*, pages 125–126. IEEE Computer Society, 2006.

[Bel10] Axel Belinfante. Jtorx: A tool for on-line model-driven test derivation and execution. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 266–270. Springer Berlin Heidelberg, 2010.

[BFJLT02] B. Baudry, F. Fleurey, J.-M. Jezequel, and Y. Le Traon. Automatic test case optimization using a bacteriological adaptation model: application to .net components. In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, pages 253–256, 2002.

[BGLP08] Fabrice Bouquet, Christophe Grandpierre, Bruno Legeard, and Fabien Peureux. A test generation solution to automate software testing. In *Proceedings of the 3rd International Workshop on Automation of Software Test*, AST '08, pages 45–48, New York, NY, USA, 2008. ACM.

[BHH$^+$94] Thomas Burch, Joachim Hartmann, Günter Hotz, M. Krallmann, U. Nikolaus, Sudhakar M. Reddy, and Uwe Sparmann. A hierarchical environment for interactive test engineering. In *ITC*, pages 461–470. IEEE Computer Society, 1994.

[BHJ$^+$06] Armin Biere, Keijo Heljanko, Tommi A. Junttila, Timo Latvala, and Viktor Schuppan. Linear encodings of bounded ltl model checking. *CoRR*, abs/cs/0611029, 2006.

[BHvMW09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[BJ07] Paul Baker and Clive Jervis. Testing uml2.0 models using ttcn-3 and the uml2.0 testing profile. In Emmanuel Gaudin, Elie Najm, and Rick Reed, editors, *SDL Forum*, volume 4745 of *Lecture Notes in Computer Science*, pages 86–100. Springer, 2007.

[BLL05] L.C. Briand, Y. Labiche, and Q. Lin. Improving statechart testing criteria using data flow information. In *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*, pages 10 pp.–104, Nov 2005.

[BLLP04] Eddy Bernard, Bruno Legeard, Xavier Luck, and Fabien Peureux. Generation of test sequences from formal specifications: Gsm 11-11 standard case study. *Software: Practice and Experience*, 34(10):915–948, 2004.

[BLR05] Gerd Behrmann, Kim G. Larsen, and Jacob I. Rasmussen. Optimal scheduling using priced timed automata. *SIGMETRICS Perform. Eval. Rev.*, 32(4):34–40, March 2005.

[BM83] D. L. Bird and C.U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[BRST08] Clark Barrett, Silvio Ranise, Aaron Stump, and Cesare Tinelli. The satisfiability modulo theories library (smt-lib). www.smt-lib.org, 2008.

[BSST09] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.

[BW05] Achim D. Brucker and Burkhart Wolff. Interactive testing with hol-testgen. In Wolfgang Grieskamp and Carsten Weise, editors, *FATES*, volume 3997 of *Lecture Notes in Computer Science*, pages 87–102. Springer, 2005.

[CBL$^+$14] Gustavo Carvalho, Flávia Barros, Florian Lapschies, Uwe Schulze, and Jan Peleska. Model-based testing from controlled natural language requirements. In Cyrille Artho and Peter Csaba Ölveczky, editors, *Formal Techniques for Safety-Critical Systems*, volume 419 of *Communications in Computer and Information Science*, pages 19–35. Springer International Publishing, 2014.

[CCGR00] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2:2000, 2000.

[CDJ11] Kalou Cabrera Castillos, Frédéric Dadeau, and Jacques Julliand. Scenario-based testing from uml/ocl behavioral models - application to posix compliance. *STTT*, 13(5):431–448, 2011.

[CDKM11] Harsh Raju Chamarthi, Peter C. Dillinger, Matt Kaufmann, and Panagiotis Manolios. Integrating testing and interactive theorem proving. In David Hardin and Julien Schmaltz, editors, *ACL2*, volume 70 of *EPTCS*, pages 4–19, 2011.

[Cen11] Cenelec. Railway applications - Communications, signalling and processing systems - Software for railway control and protection systems (EN 50128). Technical report, CENELEC, September 2011.

[CGP99] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, Cambridge, MA, USA, 1999.

[Cho78] T.S. Chow. Testing software design modeled by finite-state machines. *Software Engineering, IEEE Transactions on*, SE-4(3):178–187, 1978.

[CIvdPS05] J.R. Calame, N. Ioustinova, J. van de Pol, and N. Sidorova. Data abstraction and constraint solving for conformance testing. In *Software Engineering Conference, 2005. APSEC '05. 12th Asia-Pacific*, pages 8 pp.–, Dec 2005.

[CJRZ02] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. Stg: A symbolic test generation tool. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 470–475. Springer Berlin Heidelberg, 2002.

[CLOM06] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Object distance and its application to adaptive random testing of object-oriented programs. In *Proceedings of the 1st International Workshop on Random Testing*, RT '06, pages 55–63, New York, NY, USA, 2006. ACM.

[CLOM07] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Experimental assessment of random testing for object-oriented software. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 84–94, New York, NY, USA, 2007. ACM.

[CM94] J.J. Chilenski and S.P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, Sep 1994.

[Coo89] Walter A. Cook. *Case Grammar Theory.* Georgetown University Press, Washington, DC, 1989.

[CPL$^+$08] I. Ciupa, A. Pretschner, A. Leitner, M. Oriol, and B. Meyer. On the predictability of random tests for object-oriented software. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 72–81, April 2008.

[CSE96] John Callahan, Francis Schneider, and Steve Easterbrook. Automated software testing using model-checking, 1996.

[DBI12] Dimitris Dranidis, Konstantinos Bratanis, and Florentin Ipate. Jsxm: A tool for automated test generation. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *Software Engineering and Formal Methods*, volume 7504 of *Lecture Notes in Computer Science*, pages 352–366. Springer Berlin Heidelberg, 2012.

[DCT12] Frédéric Dadeau, Kalou Cabrera Castillos, and Régis Tissot. Scenario-based testing using symbolic animation of b models. *Softw. Test., Verif. Reliab.*, 22(6):407–434, 2012.

[DEFT09] Rolf Drechsler, Stephan Eggersgl, Grschwin Fey, and Daniel Tille. *Test Pattern Generation Using Boolean Proof Engines.* Springer Publishing Company, Incorporated, 1st edition, 2009.

[Din04] George Din. Ttcn-3. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 465–496. Springer, 2004.

[DJK$^+$99] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 285–294, New York, NY, USA, 1999. ACM.

[DKT08] Frédéric Dadeau, Adrien De Kermadec, and Régis Tissot. Combining scenario- and model-based testing to ensure posix compliance. In Egon Börger, Michael J. Butler, Jonathan P. Bowen, and Paul Boca, editors, *ABZ*, volume 5238 of *Lecture Notes in Computer Science*, pages 153–166. Springer, 2008.

[DZ03] W. Dulz and Fenhua Zhen. Matelo - statistical usage testing by annotated sequence diagrams, markov chains and ttcn-3. In *Quality Software, 2003. Proceedings. Third International Conference on*, pages 336–342, Nov 2003.

[EKRV06] Juhan-P. Ernits, Andres Kull, Kullo Raiend, and Jüri Vain. Generating tests from efsm models using guided model checking and iterated search refinement. In Klaus Havelund, Manuel Núñez, Grigore Roşu, and Burkhart Wolff, editors, *Formal Approaches to Software Testing and Runtime Verification*, volume 4262 of *Lecture Notes in Computer Science*, pages 85–99. Springer Berlin Heidelberg, 2006.

[Eng05] Dawson Engler. Concur 2005 - concurrency theory. chapter Static Analysis Versus Model Checking for Bug Finding, pages 1–1. Springer-Verlag, London, UK, UK, 2005.

[EP11] C. Efkemann and J. Peleska. Model-based testing for the second generation of integrated modular avionics. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 55–62, March 2011.

[FW08] Gordon Fraser and Franz Wotawa. Using model-checkers to generate and analyze property relevant test-cases. *Software Quality Journal*, 16(2):161–183, 2008.

[Gel07] Michael Gelfond. *In Handbook of Knowledge Representation*, chapter Answer Sets. Elsevier Science, 2007.

[GH99] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-7, pages 146–162, London, UK, UK, 1999. Springer-Verlag.

[GKP00] Elsa L. Gunter, Robert P. Kurshan, and Doron Peled. Pet: An interactive software testing tool. In E. Allen Emerson and A. Prasad Sistla, editors, *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 552–556. Springer, 2000.

[GKSB11] Wolfgang Grieskamp, Nicolas Kicillof, Keith Stobie, and Victor Braberman. Model-based quality assurance of protocol documentation: tools and methodology. *Software Testing, Verification and Reliability*, 21(1):55–71, 2011.

[GMS98] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. Automated test data generation using an iterative relaxation method. *SIGSOFT Softw. Eng. Notes*, 23(6):231–244, November 1998.

[GMS99] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. Una based iterative test data generation and its evaluation. In *14th IEEE International Conference on Automated Software Engineering(ASE'99*, pages 224–232, 1999.

[Gut99] Walter J. Gutjahr. Partition testing vs. random testing: The influence of uncertainty. *IEEE Trans. Softw. Eng.*, 25(5):661–674, September 1999.

[Hau06] Matthew Hause. The sysml modelling language. In *Fifteenth European Systems Engineering Conference*, 2006.

[HdMR05] Grégoire Hamon, Leonardo de Moura, and John Rushby. Automated test generation with sal. Technical report, Computer Science Laboratory SRI International, SRI International 333 Ravenswood Avenue, Menlo Park, CA 94025-3493, 2005.

[HHL$^+$07] Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, and Joachim Wegener. The impact of input domain reduction on search-based test data generation. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 155–164, New York, NY, USA, 2007. ACM.

[HLM$^+$08] Anders Hessel, KimG. Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using uppaal. In RobertM. Hierons, JonathanP. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 77–117. Springer Berlin Heidelberg, 2008.

[HLSC01] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Sung Deok Cha. Automatic test generation from statecharts using model checking. In *In Proceedings of FATES'01, Workshop on Formal Approaches to Testing of Software, volume NS-01-4 of BRICS Notes Series*, pages 15–30, 2001.

[HLSU02] HyoungSeok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A temporal logic based theory of test coverage and generation. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 327–341. Springer Berlin Heidelberg, 2002.

[HM07] Mark Harman and Phil McMinn. A theoretical & empirical znalysis of evolutionary testing and hill climbing for structural test data generation. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 73–83, New York, NY, USA, 2007. ACM.

[HN96] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, October 1996.

[HN04] A. Hartman and K. Nagin. The agedis tools for model based testing. *SIGSOFT Softw. Eng. Notes*, 29(4):129–132, July 2004.

[Hol03] Gerard Holzmann. *Spin Model Checker, the: Primer and Reference Manual.* Addison-Wesley Professional, first edition, 2003.

[HP00] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[HP13] Wen-ling Huang and Jan Peleska. Exhaustive model-based equivalence class testing. In Hüsnü Yenigün, Cemal Yilmaz, and Andreas Ulrich, editors, *Testing Software and Systems*, volume 8254 of *Lecture Notes in Computer Science*, pages 49–64. Springer Berlin Heidelberg, 2013.

[Hui07] Antti Huima. Implementing conformiq qtronic. In Alexandre Petrenko, Margus Veanes, Jan Tretmans, and Wolfgang Grieskamp, editors, *Testing of Software and Communicating Systems*, volume 4581 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin Heidelberg, 2007.

[Jac11] Jonathan Jacky. Pymodel: Model-based testing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 10th Python in Science Conference*, pages 43 – 48, 2011.

[JGP99] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[JJ05] Claude Jard and Thierry Jéron. Tgv: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer*, 7(4):297–315, 2005.

[JM99] Thierry Jeron and Pierre Morel. Test generation derived from model-checking. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 108–122. Springer Berlin Heidelberg, 1999.

[JVCS] Jonathan Jacky, Margus Veanes, Colin Campbell, and Wolfram Schulte. *Model-Based Software Testing and Analysis with C#*. 1 edition.

[KG04] Susan Khor and Peter Grogono. Using a genetic algorithm and formal concept analysis to generate branch coverage test data automatically. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, ASE '04, pages 346–349, Washington, DC, USA, 2004. IEEE Computer Society.

[Kor90] B. Korel. Automated software test data generation. *Software Engineering, IEEE Transactions on*, 16(8):870–879, Aug 1990.

[KP12] Teemu Kanstrén and Olli-Pekka Puolitaival. Using built-in domain-specific modeling support to guide model-based test generation. In *MBT*, pages 58–72, 2012.

[Lan11] Sebastian Langer. Visualisierung von transitionsbäumen für model- und szenariobasiertes testen. Bachelor report, Universität Bremen, Germany, September 2011.

[LHM08] Kiran Lakhotia, Mark Harman, and Phil McMinn. Handling dynamic data structures in search based testing. In *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, GECCO '08, pages 1759–1766, New York, NY, USA, 2008. ACM.

[LK01] Marc Lettrari and Jochen Klose. Scenario-based monitoring and testing of real-time uml models. In Martin Gogolla and Cris Kobryn, editors, *UML*, volume 2185 of *Lecture Notes in Computer Science*, pages 317–328. Springer, 2001.

[LP08] Helge Löding and Jan Peleska. Symbolic and abstract interpretation for C/C++ programs. *Electr. Notes Theor. Comput. Sci.*, 217:113–131, 2008.

[LP10] Helge Löding and Jan Peleska. Timed moore automata: Test data generation and model checking. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 449–458, Washington, DC, USA, 2010. IEEE Computer Society.

[MB05] Bruno Marre and Benjamin Blanc. Test selection strategies for lustre descriptions in gatel. *Electron. Notes Theor. Comput. Sci.*, 111:93–111, January 2005.

[McM04] Phil McMinn. Search-based software test data generation: A survey. *SOFTWARE TESTING, VERIFICATION AND RELIABILITY*, 14:105–156, 2004.

[MFT12] Bogdan Marculescu, Robert Feldt, and Richard Torkar. A concept for an interactive search-based software testing system. In Gordon Fraser and Jerffeson Teixeira de Souza, editors, *SSBSE*, volume 7515 of *Lecture Notes in Computer Science*, pages 273–278. Springer, 2012.

[MGP$^+$12] Stefan Milius, Henning Günther, Jan Peleska, Oliver Möller, Helge Löding, Martin Sulzmann, Ramin Hedayati, and Axel Zechner. A framework for formal verification of systems of synchronous components. In Holger Giese, Michaela Huhn, Jan Phillips, and Bernhard Schätz, editors, *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VIII, Schloss Dagstuhl, Germany, 2012, Tagungsband Modellbasierte Entwicklung eingebetteter Systeme*, pages 145–154. fortiss GmbH, München, 2012.

[MHL$^+$12] P. McMinn, M. Harman, K. Lakhotia, Y. Hassoun, and J. Wegener. Input domain reduction through irrelevant variable removal and its effect on local, global, and hybrid search-based structural test data generation. *Software Engineering, IEEE Transactions on*, 38(2):453–477, March 2012.

[MIU05] Johannes Mayer, Abteilung Angewandte Informationsverarbeitung, and Universität Ulm. On testing image processing applications with statistical methods. In *In Software Engineering (SE 2005), Lecture Notes in Informatics*, pages 69–78, 2005.

[MLL09] Qaisar A. Malik, Johan Lilius, and Linas Laibinis. Model-based testing using scenarios and event-b refinements. In Michael J. Butler, Cliff B. Jones, Alexander Romanovsky, and Elena Troubitsyna, editors, *Methods, Models and Tools for Fault Tolerance*, volume 5454 of *Lecture Notes in Computer Science*, pages 177–195. Springer, 2009.

[MMS01] C.C. Michael, Gary McGraw, and M.A. Schatz. Generating software test data by evolution. *Software Engineering, IEEE Transactions on*, 27(12):1085–1110, Dec 2001.

[Moo56] Edward F. Moore. Gedanken experiments on sequential machines. In *Automata Studies*, pages 129–153. Princeton U., 1956.

[MS04] Nashat Mansour and Miran Salame. Data generation for path testing. *Software Quality Journal*, 12(2):121–136, 2004.

[MS06] Johannes Mayer and Christoph Schneckenburger. An empirical analysis and comparison of random testing techniques. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, pages 105–114, New York, NY, USA, 2006. ACM.

[NS08] Thomas Noll and Bastian Schlich. Delayed nondeterminism in model checking embedded systems assembly code. In *Proceedings of the 3rd International Haifa Verification Conference on Hardware and Software: Verification and Testing*, HVC'07, pages 185–201, Berlin, Heidelberg, 2008. Springer-Verlag.

[OA99] Jeff Offutt and Aynur Abdurazik. Generating tests from uml specifications. In Robert France and Bernhard Rumpe, editors, *UML 99 - The Unified Modeling Language*, volume 1723 of *Lecture Notes in Computer Science*, pages 416–429. Springer Berlin Heidelberg, 1999.

[Obj10] Object Management Group. OCL 2.2 Specification, 2010.

[ODC06] David Owen, Dejan Desovski, and Bojan Cukic. Random testing of formal software models and induced coverage. In *Proceedings of the 1st International Workshop on Random Testing*, RT '06, pages 20–27, New York, NY, USA, 2006. ACM.

[OMG11a] OMG. OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.4.1. Technical report, Object Management Group, August 2011.

[OMG11b] OMG. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1. Technical report, Object Management Group, August 2011.

[oR11] Special C. of RTCA. DO-178C, software considerations in airborne systems and equipment certification, 2011.

[Pel13] Jan Peleska. Industrial-strength model-based testing - state of the art and current challenges. In Alexander K. Petrenko and Holger Schlingloff, editors, *MBT*, volume 111 of *EPTCS*, pages 3–28, 2013.

[PHL$^+$11] Jan Peleska, Artur Honisch, Florian Lapschies, Helge Loeding, Hermann Schmid, Peer Smuda, Elena Vorobev, and Cornelia Zahlten. A real-world benchmark model for testing concurrent real-time systems in the automotive domain. In Burkhart Wolff and Fatiha Zaidi, editors, *Testing Software and Systems*, volume 7019 of *Lecture Notes in Computer Science*, pages 146–161. Springer Berlin Heidelberg, 2011.

[PHP99] Roy P. Pargas, Mary Jean Harrold, and Robert R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification And Reliability*, 9:263–282, 1999.

[PLK07] Jan Peleska, Helge Löding, and Tatiana Kotas. Test automation meets static analysis. In Rainer Koschke, Otthein Herzog, Karl-Heinz Rödiger, and Marc Ronthaler, editors, *INFORMATIK 2007: Informatik trifft Logistik. Band 2. Beiträge der 37. Jahrestagung der Gesellschaft für Informatik e. V. (GI), 24.-27. September 2007 in Bremen*, volume 110 of *LNI*, pages 280–290. GI, 2007.

[Plo81] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[PPW$^+$05] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 392–401, New York, NY, USA, 2005. ACM.

[Pre06] Alexander Pretschner. Zur kosteneffektivität des modellbasierten testens. In *MBEES*, pages 85–94, 2006.

[Pro03] S.J. Prowell. Jumbl: a tool for model-based statistical testing. In *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, pages 9 pp.–, Jan 2003.

[Pro05] S. J. Prowell. Using markov chain usage models to test complex systems. In *Proc. 38th Annual Hawaii International DATA MUTATION TESTING: ACASE STUDY Page 17 of 18 THE COMPUTER JOURNAL, 2007 Conf. System Sciences (HICSS'05), Big Island, HI, January 3–6*, page 318, 2005.

[PSK08] Stephan Pietsch and Bogdan Stanca-Kaposta. Model-based testing with utp and ttcn-3 and its application to hl7. Technical report, Testing Technologies IST GmbH, 2008.

[PVL11] Jan Peleska, Elena Vorobev, and Florian Lapschies. Automated test case generation with smt-solving and abstract interpretation. In Mihaela Bobaru, Klaus Havelund, GerardJ. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 298–312. Springer Berlin Heidelberg, 2011.

[Rav07] A.R. Ravindran. *Operations Research and Management Science Handbook*. Operations Research Series. Taylor & Francis, 2007.

[RBW06] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[RK11] Hassan Reza and Scott D. Kerlin. A model-based testing using scenarios and constraints-based modular petri nets. In *ITNG*, pages 568–573. IEEE Computer Society, 2011.

[SE02] Niklas Sörensson and Niklas Een. Minisat v1.13 - a sat solver with conflict-clause minimization. 2005. sat-2005 poster. 1 perhaps under a generous notion of "part-time", but still concurrently taking a statistics course and leading a normal life. Technical report, 2002.

[SLS05] Andreas Spillner, Tilo Linz, and Hans Schaefer. *Software testing foundations*. dpunkt., Heidelberg, 2005.

[SML06] Manoranjan Satpathy, Qaisar A. Malik, and Johan Lilius. Synthesis of scenario based test cases from b models. In Klaus Havelund, Manuel Núñez, Grigore Rosu, and Burkhart Wolff, editors, *FATES/RV*, volume 4262 of *Lecture Notes in Computer Science*, pages 133–147. Springer, 2006.

[SWB02] Harmen Sthamer, Joachim Wegener, and Andre Baresel. Using evolutionary testing to improve efficiency and quality in software testing. In *In Proceedings of the 2nd Asia-Pacific Conference on Software Testing Analysis and Review (AsiaSTAR*, pages 22–24, 2002.

[Tar71] Robert Tarjan. Depth-first search and linear graph algorithms. In *Switching and Automata Theory, 1971., 12th Annual Symposium on*, pages 114–121, 1971.

[TKSL04] L. Tan, Jesung Kim, O. Sokolsky, and I. Lee. Model-based testing and monitoring for hybrid embedded systems. In *Information Reuse and Integration, 2004. IRI 2004. Proceedings of the 2004 IEEE International Conference on*, pages 487–492, Nov 2004.

[Tre08] Jan Tretmans. Model based testing with labelled transition systems. In RobertM. Hierons, JonathanP. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer Berlin Heidelberg, 2008.

[Tre11] Jan Tretmans. Model-based testing and some steps towards test-based modelling. In Marco Bernardo and Valerie Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 297–326. Springer Berlin Heidelberg, 2011.

[TSL04] Li Tan, Oleg Sokolsky, and Insup Lee. Specification-based testing with linear temporal logic. In Du Zhang, Éric Grégoire, and Doug DeGroot, editors, *IRI*, pages 493–498. IEEE Systems, Man, and Cybernetics Society, 2004.

[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.

[Utt08] Mark Utting. The role of model-based testing. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, volume 4171 of *Lecture Notes in Computer Science*, pages 510–517. Springer Berlin Heidelberg, 2008.

[VCG$^+$08] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. In RobertM. Hierons, JonathanP. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 39–76. Springer Berlin Heidelberg, 2008.

[WB04] Joachim Wegener and Oliver Bühler. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In *GECCO (2)*, pages 1400–1412, 2004.

[Weg03] Joachim Wegener. Evolutionary testing of embedded systems. In Rolf Drechsler and Nicole Drechsler, editors, *Evolutionary Algorithms for Embedded System Design*, volume 10 of *Genetic Algorithms and Evolutionary Computation*, pages 1–33. Springer US, 2003.

[WJ91] E.J. Weyuker and Bingchiang Jeng. Analyzing partition testing strategies. *Software Engineering, IEEE Transactions on*, 17(7):703–711, Jul 1991.

[WL05] Stefan Wappler and Frank Lammermann. Using evolutionary algorithms for the unit testing of object-oriented software. In

Proceedings of the 2005 Conference on Genetic and Evolutionary Computation, GECCO '05, pages 1053–1060, New York, NY, USA, 2005. ACM.

[WS07] Stefan Wappler and Ina Schieferdecker. Improving evolutionary class testing in the presence of non-public methods. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 381–384, New York, NY, USA, 2007. ACM.