Universiteit van Amsterdam

MSc Software Engineering
Master Thesis

Trace Collection and Data Coverage for Model Based Testing

by
Sjoerd van der Heijden
10336001

October 4, 2019

18 EC

Supervisor:
dr. Ana Oprescu
Daily supervisor:
dr. ir. Machiel van der Bijl

Second Reader:
dr. Clemens Grelck

The work for this thesis was hosted at

axini

Abstract

We propose a method for generating complete test suites using model based testing (MBT), and using the formalism of MBT we prove that this method can indeed produce complete test suites. We also implemented a test suite generator following our proposed method, which aims to generate test suites that can ensure both trace and data coverage. To keep our algorithm from generating infinitely large test suites, we limit the number of action labels in our test cases. As such, test suites generated by our algorithm are able, when executed on a system under test (SUT), to verify whether a SUT conforms to its specification up to the given number of actions. Furthermore, we analyzed the performance of our algorithm by generating test suites from a complex STS. We find that our algorithm generates test suites at a rate of the order of one trace per second. We also identified parts of our code where improvements would greatly enhance the performance of our algorithm.

Contents

1 Introduction **2**

2 Background **3**
- 2.1 Formalizing testing . . . 3
- 2.2 Formalizing models . . . 3
- 2.3 Implementation relation **ioco** . . . 6
- 2.4 Coverage criteria . . . 6
- 2.5 Implementations of MBT . . . 7

3 Complete test suite derivation **8**
- 3.1 Proof of algorithm functionality . . . 8
- 3.2 Maximum number of traces . . . 9

4 Implementation of test suite generator with data coverage **11**
- 4.1 Trace collection . . . 11
- 4.2 Constraint collection . . . 12
- 4.3 Solving for domains . . . 12
- 4.4 Variable valuations . . . 12

5 Experiments on test suite generator **13**
- 5.1 Experimental method . . . 13
- 5.2 Results . . . 13
- 5.3 Discussion . . . 14

6 Overview of MBT tools **15**

7 Conclusions **17**
- 7.1 Future work . . . 17

Appendix A Algorithm for complete test suite generation **22**
- A.1 Initialization . . . 22
- A.2 Trace collection . . . 24
- A.3 Constraint collection . . . 25
- A.4 Constraint solving . . . 29

Chapter 1

Introduction

Software testing is a tedious activity, and requires significant effort and time without contributing directly to the functionality of software. While programming allows us to automate and scale up many processes, writing tests is often still done by hand, and these tests are usually specific to a small section of code such as a function. Furthermore, changes to code often require changing the tests as well.

Model based testing (MBT) is a black-box testing technique that allows for automated testing; it allows for both automated test generation and execution. MBT is usually done by creating a model of the requirements of a system under test (SUT), then deriving test cases from the model and executing them on the SUT. In this way, any part of the SUT that does not function conform the specification of the SUT can be found. While the models are still written by hand, as long as the specifications of a system do not change the model and test cases can remain the same. When the specifications do change the model must be manually updated, but afterwards new tests can be automatically derived and executed again. A survey reports that MBT allowed respondents to find more bugs, and reduced testing costs and duration[1].

The purpose of software testing is to verify that SUTs work as specified, and to find deviations from the specified behavior. MBT is based on a formal framework, in this framework the models take the form of directed graphs. Thanks to this framework, the (non-)conformance of a SUT to its specification can also be expressed formally. In this thesis we use this formalism to propose a method to generate test cases, and we prove that that collection of test cases can be used to show whether or not a SUT behaves conform its specification.

When implementing an MBT tool, one must take into consideration that many systems would require infinite testing to show that it always behaves as specified. To test a system thoroughly but also in a finite amount of time, one may judge a system sufficiently tested once certain coverage criteria are met. Coverage criteria used existing MBT tools are aimed at generating sets of test cases that cover certain portions of models, or cover different variable valuations and valuation combinations. For this thesis, we built an algorithm that follows our formally proposed and proven method for test case generation. In order to generate only a finite number of test cases we limit our path coverage to paths of a given length, and our variable value coverage to only the minimum and maximum values of variables, of only those variables of which the values can be chosen during testing.

We have studied existing MBT tools, and we show that our approach is novel. We find only one other tool that generates tests to cover both all paths and multiple variable valuations, but this tool lacks in other aspects.

In Chapter 2 we formalize software testing, detail the formal framework of MBT and elaborate on coverage criteria. In Chapter 3 we discuss and prove our method of test case generation, and reason about the number of test cases required to reach our coverage criteria. In Chapter 4 we describe the implementation of our test case generator. We have performed experiments to show the performance of our implementation, which we discuss in Chapter 5. In Chapter 6 we discuss existing MBT tools. We conclude this thesis in Chapter 7.

Chapter 2

Background

In this section we will first provide an introduction into formal software testing (Section 2.1), after that we give background information on formal MBT (Sections 2.2 to 2.4), and finally we discuss previous work on MBT that forms the basis for this thesis (Section 2.5).

2.1 Formalizing testing

The goal of software testing is to verify the proper functioning of a SUT. SUTs will also be referred to as *implementations*, and their proper function is their *specification*. MBT is based on the **ioco** relation, which expresses whether an implementation conforms to its specification. We go into detail about the **ioco** relation in Section 2.3. To verify the conformance of an implementation to its specification using MBT, one can check the specification for inputs the implementation can receive and the outputs it should then give, and verify that these outputs are indeed given as response to the inputs. A sequence of such inputs and outputs is a *test case*, and when an implementation shows the behaviour prescribed by the test case, the implementation is said to pass the test. Otherwise, the implementation fails the test and we will call it a faulty implementation with respect to the specification. If an implementation passes all test cases derivable from its specification, the implementation conforms to its specification. A collection of test cases is a *test suite*. For a given specification, a test suite is considered *exhaustive* when it correctly points out any faulty implementations as faulty, and maybe also some correct implementations, and a test suite is *sound* when it points out only faulty implementations, but maybe not all. When a test suite is both sound and exhaustive, i.e. it can identify all faulty implementations and only faulty implementations, the test suite is *complete*[2, 3].

2.2 Formalizing models

The models used in MBT are generally models of specifications. More specifically, the specifications are often modeled as labeled transition systems (LTSs) (e.g. Fig. 2.1). These and similar types of systems are used in the literature as well[3, 4, 5, 6]. Labeled transition systems are directed graphs where the nodes are called states, which reflect possible states a SUT can be in, and the edges are labeled transitions between these states, where the labels reflect actions that can change the state of a SUT. For example in the case of the coffee machine, the user can perform the action of pushing a button which causes a change of state of the machine so that it becomes ready to dispense coffee. LTSs are formally defined as follows:

Definition 1. A labeled transition system (LTS) is a tuple $\langle Q, q_0, L, T\rangle$ where Q is a set of states, $q_0 \in Q$ is the start state of the LTS, L is a set of labels, and $T \subseteq Q \times (L \cup \{\tau\}) \times Q$ is the transition relation, where τ denotes unobservable labels, with $\tau \notin L$. Further explanation and implications of the τ label are discussed at a later point in this section.

An LTS can also contain variables, but the system then requires states for all possible variable valuations. To add variables without increasing the size of the model, the variables can be kept symbolic. An LTS with symbolic variables is also called a symbolic transition system (STS):

Definition 2. A symbolic transition system (STS) is a tuple $\langle Q, q_0, Var, Var_0, L, T\rangle$ where Var is the set of variables and Var_0 the initial valuations of the variables. With the introduction of variables, the transition relation becomes $T \subseteq Q \times (L \cup \{\tau\}) \times G(Var) \times A(Var) \times Q$, where $G(Var)$ is the set of possible constraints over the variables and $A(Var)$ is the set of assignments. The significance of a constraint $g \in G$ is that a transition with g can only be followed when g is satisfied. An assignment $a \in A$ on a transition means that variable values change according to a when following that transition. The other symbols are defined as in Definition 1.

To illustrate how transitions look in an STS, an example transition in an STS would be $(q_0, \mu, x < 10, x \mathrel{+}= 5, q_1) \in T$, where $q_0, q_1 \in Q$, $\mu \in L \cup \{\tau\}$ and $x \in Var$. This transition would originate from the start state, have action μ, only be followable if $x < 10$, add 5 to the value of x if followed, and leave the STS in state q_1 when followed.

Implementations behave similar to transition systems, but require an environment (e.g. a user or another system) to interact with. These interactions can either be performed by the environment or by the implementation, we refer to these interactions as input and output actions respectively. Additionally, input actions can be performed on implementations at any time; a user may press the button on a coffee machine repeatedly while the machine is still dispensing coffee. Due to these differences, implementations can be viewed as input-output symbolic transition system (IOSTS) (e.g. Fig. 2.2):

Definition 3. An input-output symbolic transition system (IOSTS) is a tuple $\langle Q, q_0, \mathcal{V}, \mathcal{I}, \mathcal{V}_0, L_I, L_U, T\rangle$, with $\mathcal{V}$ the set of state variables, $\mathcal{V}_0$ the initial valuation of the state variables and $\mathcal{I}$ the set of label parameters, L_I the set of input actions and L_U the set of output actions, and $\langle Q, q_0, \mathcal{V} \cup \mathcal{I}, \mathcal{V}_0, L_I \cup L_U, T\rangle$ is an STS. Note that label parameters do not have initial valuations, as the values of label parameters are fleeting; label parameters only have values while following transitions on which they are assigned. Input actions are actions performed on implementations by their environment and output actions are performed on the environment by implementations. State variables are variables that belong to the STS and can be accessed and changed by assignments on transitions. Label parameters are values that are assigned by the environment when following transitions where they are used. Furthermore, $\mathcal{V}$ is disjoint from $\mathcal{I}$ and L_I is disjoint from L_U; $\mathcal{V} \cap \mathcal{I} = \emptyset$ and $L_I \cap L_U = \emptyset$. Additionally, IOSTSs are input-enabled, which means that

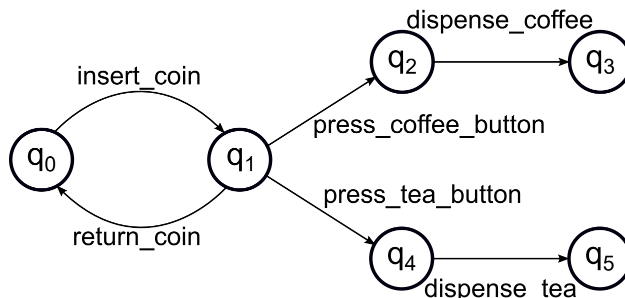

Figure 2.1: *Example of an LTS, representing a simplified specification of a coffee machine. The circles are the states, where state q_0 is the start state. The arrows are the transitions. The phrases next to the transitions are labels, and reflect actions that can be performed to change the state of the coffee machine.*

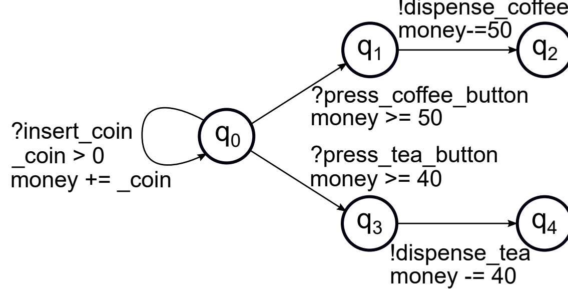

Figure 2.2: *Example of an IOSTS, representing a simplified specification of a coffee machine. The circles are the states, where state q_0 is the start state. The arrows are the transitions. The phrases following question marks represent input actions, actions which can be performed by an environment of an implementation. Phrases following exclamation marks represent output actions, which should be performed by the implementation. Transitions may have constraints, which need to be satisfied in order to follow that transition, and assignments, which change the value of the specified variables. Variables preceded by an underscore are label parameters, and are chosen by the environment while following a transition on which it is used. Note that IOSTSs are input-enabled, meaning that every input action should be allowed from any state. In this example that would require adding transitions with each input action going from each state from where that action is not possible yet. These transitions would loop back to the state from which the transition originates and would not be constrained nor influence variable values. We refrain from showing these transitions to prevent further cluttering the figure.*

any input action can be performed from any state.

Note that in literature different names may be used for state variables and label parameters, for example Frantzen *et al.* [7, 4] refer to them as location variables and interaction variables respectively, and Jeannet *et al.* [6] call them internal and external variables respectively.

Definition 4. Given an STS $s = \langle Q, q_0, Var, Var_0, L, T\rangle$ with states $q, q' \in Q$ and $\mu \in (L \cup \{\tau\}) \times G(Var) \times A(Var)$, we use the following shorthand notations for transitions in this thesis:
$q \xrightarrow{\mu} q' \quad \stackrel{\text{def}}{=} (q, \mu, q') \in T$
$q \xrightarrow{\mu} \quad \stackrel{\text{def}}{=} \exists q'' \in Q : q \xrightarrow{\mu} q''$
$q \xrightarrow{\mu_1 \cdot \mu_2 \cdot \ldots \cdot \mu_n} q' \quad \stackrel{\text{def}}{=} \exists q_1, q_2 \ldots q_{n-1} \in Q : q \xrightarrow{\mu_1} q_1 \xrightarrow{\mu_2} q_2 \ldots \xrightarrow{\mu_{n-1}} q_{n-1} \xrightarrow{\mu_n} q'$
$q \xrightarrow{\mu_1 \cdot \mu_2 \cdot \ldots \cdot \mu_n} \quad \stackrel{\text{def}}{=} \exists q'' \in Q : q \xrightarrow{\mu_1 \cdot \mu_2 \cdot \ldots \cdot \mu_n} q''$

From a state q, we refer to those $\mu \in (L \cup \{\tau\}) \times G(Var) \times A(Var)$ for which $q \xrightarrow{\mu}$ holds as legal actions. Similarly we refer to sequences $\mu_1 \cdot \ldots \cdot \mu_n$ as legal from a state q when $q \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n}$ holds.

A sequence of action labels is called a *trace*, generally denoted by σ. We use $|\sigma|$ as the cardinality of σ. A sequence of transitions is called a *path*. The set of all sequences that can be made with an alphabet L is denoted as L^*. We write $L \cup \{\tau\}$ as L_τ, and accordingly L_τ^* denotes the set of all sequences that may also contain τ. When an STS reaches a state from which no transitions originate the STS is said to be in *deadlock*. The empty sequence of labels is denoted by ϵ.

The τ label is unobservable because $\tau \notin L_I \cup L_U$; there is no interaction between implementation and environment when a transition with the tau label is followed by the implementation. This means that when an implementation reaches a state with a τ-labelled transition, the environment cannot know whether the implementation takes that transition or remains in the current state. If it may be uncertain in which state a system is, the system is said to be *non-deterministic*. Non-deterministic implementations are difficult to test with predefined test traces, as the implementation may follow different transitions than prescribed by the trace, and the environment may only learn of this once an observable action is performed again. Systems where two transitions with the same label originate from the same state is also non-deterministic; one trace may result in multiple different states.

Since τ denotes unobservable actions it does not occur in traces used for testing. Hence, we use the following shorthand notations to abstract from τ labels:

Definition 5. Given an STS $s = \langle Q, q_0, \mathcal{V}, \mathcal{I}, L, T \rangle$, with $q, q' \in Q$ and $\mu_1 \dots \mu_n \in L$ we define the following shorthands:

$q \stackrel{\epsilon}{\Rightarrow} q'$ $\stackrel{\text{def}}{=} (q \stackrel{\tau}{\to} \dots \stackrel{\tau}{\to} q' \vee q = q')$

$q \xrightarrow{\mu_1 \cdot \mu_2 \cdot \dots \cdot \mu_n} q'$ $\stackrel{\text{def}}{=} q \stackrel{\epsilon}{\Rightarrow} \stackrel{\mu_1}{\longrightarrow} \stackrel{\epsilon}{\Rightarrow} \dots \stackrel{\epsilon}{\Rightarrow} \stackrel{\mu_n}{\longrightarrow} \stackrel{\epsilon}{\Rightarrow} q'$

Definition 6. Given an STS $s = \langle Q, q_0, \mathcal{V}, \mathcal{I}, L, T \rangle$, with $q \in Q$ and $\sigma \in L^*$ we define the following sets:

q **after** σ $\stackrel{\text{def}}{=} \{q' \mid q \stackrel{\sigma}{\Rightarrow} q'\}$

ini(q) $\stackrel{\text{def}}{=} \{\mu \in L \mid q \stackrel{\mu}{\to}\}$

ini(Q') $\stackrel{\text{def}}{=} \bigcup_{q \in Q'} \textbf{ini}(q)$

traces(s) $\stackrel{\text{def}}{=} \{\sigma \in L^* \mid q_0 \stackrel{\sigma}{\Rightarrow}\}$

2.3 Implementation relation ioco

MBT testing theory relies on the **ioco** implementation relation, where **ioco** stands for input-output conformance. The **ioco** relation is used to describe whether an implementation conforms to a specification: an implementation conforms to a specification when the implementation never performs output actions that should not be possible according to the specification. The formal definition of the **ioco** relation is as follows[3, 4, 6]:

Definition 7. Given a set of input labels L_I, a set of output labels L_U and a set of traces $\mathcal{F}$, we define the relation $\textbf{ioco}_{\mathcal{F}} \subseteq IOSTS(L_I, L_U) \times STS(L_I \cup L_U)$ as follows:

$$i \ \textbf{ioco}_{\mathcal{F}} \ s \stackrel{\text{def}}{\iff} \forall \sigma \in \mathcal{F} : \ out\ (i \ \textbf{after} \ \sigma) \subseteq \ out\ (s \ \textbf{after} \ \sigma) \tag{2.1}$$

In this thesis, we will use only $\textbf{ioco}_{traces(s)}$ as conformance relation, where we defined *traces(s)* in Definition 6 as the set of all legal traces through an STS s. Please note that test suites that are complete with respect to $\textbf{ioco}_{traces(s)}$ must be equal to *traces(s)*, in accordance with Definition 7.

Note that an implementation that can perform fewer output actions than a specification may still conform to that specification. This is because the **ioco** relation demands only that the output actions performed by the implementation are one of the actions that are prescribed by the specification. If the implementation never performs some output actions that are allowed by the specification, the implementation still conforms to the specification.

2.4 Coverage criteria

According to the $\textbf{ioco}_{traces(s)}$ relation, a SUT can only be verified to conform to its specification s when each trace $\in$ *traces*(s) has been executed on it and verified to yield specified behaviour. From a practical view this is not feasible, as *traces*(s) is infinitely big as soon as s contains any looping transitions, which would also require infinite testing time. Hence, we have to select a finite test suite from *traces*(s) which can test an implementation sufficiently, but also in a finite amount of time.

One approach to limiting test suite size is to select a set of tests that ensures *state coverage* instead of **ioco**. A test suite that ensures state coverage makes sure that the implementation enters every state at least once during testing. A stricter selection criterion is *transition coverage*, which yields test suites which ensure that every transition in an STS is followed at least once. Transition coverage is more strict than state coverage since all states in an STS are connected by transitions, and as such when all transitions are covered, all states

are covered too. The reverse is not true, e.g. in the case where two transitions originate and target the same states, both states can be covered by following only one transition. States without transitions are considered independent STSs in which the single state is the start state, and the start state is always immediately covered at the start of a test. A test suite complete according to the $\mathbf{ioco}_{traces(s)}$ relation is said to ensure *trace coverage*. Trace coverage is a stricter testing criterion than both state coverage and transition coverage. This is because all transitions that can be covered can also be covered by at least one trace, otherwise the transition could not have been covered. To retain the strictness of test suites that ensure trace coverage while building a finite test suite, one can select those traces from $traces(s)$ that have a given cardinality. Such a test suite will be able to find all faults in implementation for a given specification, as long as these faults are reachable within the specified number of actions. Note that if no variable assignments happen on any transition in an STS, trace coverage is not stricter than transition coverage; the only additional information that trace coverage gives compared to transition coverage is about different combinations of variable assignments and constraints.

State coverage, transitions coverage and trace coverage all concern the transition relation of an STS, but the variables and their valuations must also be taken into account; different variable valuations may elicit different behaviours in an STS, even for a single trace. *Data coverage* is a metric for how much of variable domains are covered during testing. Since data coverage criteria concern the variables but not the structure of STSs, data coverage is not stricter or looser than state, transition or trace coverage. Rather, it can be imposed alongside one of those three criteria. Since variable domains may be unbounded it is not feasible to test for all possible variable values and combinations thereof. As such, again a selection of test cases must be made. Three different aspects of data coverage can be selected for: which variable values to test for, what combinations of variable valuations to test for, and whether these previous selections should be made per transition or per trace.

As for the first aspect, we discussed that testing for all variable values is unfeasible. Instead three possible selection criteria are to valuate variables to their minimum and maximum value at least once, to do equivalence partitioning, or a combination of those two. Equivalence partitioning implies to test for at least one value from each subdomain of values that can be assigned to a variable. A combination of testing for minimum and maximum values and equivalence partitioning implies valuating variables as the boundary values of all its subdomains at least once across all tests.

For the second aspect, it could be deemed sufficient to valuate each variable independently, or one could choose to do e.g. pairwise testing or all-pairs testing.

The final aspect is whether the selection of variable values and selection of value combinations should happen once for each transition in an STS, or once for each individual path through an STS. The latter choice would make for a more rigorous test suite, but would also require far more test cases than the former.

2.5 Implementations of MBT

Aside from on theory, there has been previous work on implementing MBT tools. One such tool is the Axini Modelling Suite (AMS). This tool is currently capable of generating tests that ensure high transition coverage, and work has been done to enable it to ensure trace coverage for traces of a given length[8] and data coverage [9]. However, the work on trace coverage yielded an algorithm too computationally expensive to be included in AMS, and the algorithm to ensure data coverage could not reliably cover all transitions of STSs, and could therefore not ensure data coverage throughout those STSs. More MBT tools are discussed in Chapter 6, but out of those only one (Conformiq Designer) is able to meet the trace coverage and data coverage criteria at the same time, and that tool does not support non-deterministic models.

Chapter 3

Complete test suite derivation

In this section we suggest a means of deriving complete test suites, and continue to prove that such test suites are indeed complete (Section 3.1). Afterwards, we find theoretical upper bounds to test suite sizes, using means that require only little information about a given STS (Section 3.2).

3.1 Proof of algorithm functionality

Here we will propose a method for deriving test suites for an STS s that are complete with respect to the $\mathbf{ioco}_{traces(s)}$ relation, where we take s to have no deadlocks. We then prove that this method indeed yields complete test suites. The test suite we derive for s shall be referred to as $\mathit{Gentraces}(s, n)$, for generated traces.

The method we propose for deriving a complete test suite for and STS s of traces with length n is by recursively appending legal actions to all legal traces of length $n-1$, where the complete test suite of traces with $n = 0$ is $\{\epsilon\}$. That means that we derive $\mathit{Gentraces}(s, n)$ as follows:

$$\mathit{Gentraces}(s, 0) = \{\epsilon\} \tag{3.1}$$

$$\mathit{Gentraces}(s, n) = \{\sigma \cdot \mu \mid \sigma \in \mathit{Gentraces}(s, n-1),\ \mu \in \mathbf{ini}\,(q_0\,\mathbf{after}\,\sigma)\,\} \tag{3.2}$$

As we mentioned before, test suites that are complete with respect to $\mathbf{ioco}_{traces(s)}$ must be equal to $traces(s)$, which is the set of all legal traces through s. Since $\mathit{Gentraces}(s, n)$ contains traces only of a given length, we define $traces(s, n)$ as the set of all legal traces through s of length n:

$$traces(s, n) \stackrel{\text{def}}{=} \{\sigma \in L^* \mid q_0 \stackrel{\sigma}{\Rightarrow}, |\sigma| = n\} \tag{3.3}$$

Since $traces(s)$ can fully verify the conformance of an implementation to a specification, $traces(s, n)$ can verify whether an implementation conforms to a specification after following n observable transitions. As such we say that $traces(s, n)$ is a complete test suite of traces of length n. Traces with length $< n$ do not need to be included in $traces(s, n)$ to make it complete, as these traces are implicitly tested by the traces of length n that are extended from them. By showing that $\mathit{Gentraces}(s, n) = traces(s, n)$ we will therefore prove that $\mathit{Gentraces}(s, n)$ is a complete test suite of traces of length n. To prove the equality of these sets, we will show that both sets contain the other:

$$\mathit{Gentraces}(s, n) \subseteq traces(s, n) \tag{3.4}$$

$$\mathit{Gentraces}(s, n) \supseteq traces(s, n) \tag{3.5}$$

The truth of Eq. (3.4) is clear through the fact that *traces*(s, n) is the complete set of traces of length n by definition. The only way in which *Gentraces*(s, n) could contain elements not in *traces*(s, n) would be by containing illegal traces or by containing traces of length $\neq n$. However, the traces in *Gentraces*(s, n) are legal traces extended with a legal action, and are therefore all legal. Furthermore, the traces that are extended are of length $n - 1$ and are extended by a single action, resulting in traces with length $= n$.

Eq. (3.5) can only be untrue if *traces*(s, n) would contain traces not in *Gentraces*(s, n). These could only be traces that are not extended from *Gentraces*$(s, n-1)$, as these extended traces are included in *Gentraces*(s, n). However, when excluding the last action of any trace of length n, the result is a trace of length $n - 1$, which must be in *Gentraces*$(s, n - 1)$. Therefore, all traces in *traces*(s, n) must also be in *Gentraces*(s, n). This argument holds under the assumption that the transition relation of STS s does not change over time. If more transitions were added to s depending on trace length n, *traces*(s, n) would suddenly contain additional traces containing the associated label. *Gentraces*(s, n) would not contain these traces since the new transitions were absent during the derivation of earlier generations of *Gentraces*.

We have now proven that *Gentraces*$(s, n) =$ *traces*(s, n), and thereby shown that *Gentraces*(s, n) is a complete test suite of traces of length n through a given STS s. Moreover, since there is nothing that forbids $n = \infty$, $\bigcup_{n=0}^{\infty} \mathit{Gentraces}(s, n) = \mathit{traces}(s)$ and is a complete test suite according to the $\mathbf{ioco}_{traces(s)}$ relation.

3.2 Maximum number of traces

Here we will present methods for finding the theoretical maximum of traces in the complete test suite of traces of length n *traces*(s, n) for a given transition system s. If we can closely approximate the maximum number of legal traces through s using only little information of the system, we can gain information about a test suite without having to derive the complete test suite itself. We first discuss maximum test suite size for LTSs, and afterwards for STSs.

First we discuss our method for finding an upper bound to the size of a test suite uses only the number of transitions in an LTS(see Theorem 1). We take an LTS s with transition relation T. Since we consider only the number of transitions $|T|$ here, we must assume that T maximizes $|\mathit{traces}(s, n)|$ for a given number of transitions. Furthermore, we take each transition to have a unique label. The transition relation that maximizes $|\mathit{traces}(s, n)|$ prescribes that all transitions in s must originate from the start state. In this way, we see that $|\mathit{traces}(s, n = 1)|$ is maximized; any other transition relation with the same number of transitions would reduce $|\mathit{traces}(s, n = 1)|$ as there would be actions that could not be taken from the start state. To maximize $|\mathit{traces}(s, n = 2)|$, T must prescribe that all transitions in s should loop back to the start state, since in this way all traces in $|\mathit{traces}(s, n = 1)|$ can be extended by the maximum number of actions.

Theorem 1. *Take an LTS $s = \langle Q, q_0, L, T \rangle$ with one state and each transition both originating from and targeting that state. Take all transitions to have unique labels. An upper bound to the size of the complete test suite for s is $|traces(s, n)| = |T|^n$.*
Proof.

$$\text{Base case: } |\mathit{traces}(s, n = 0)| = |\{\epsilon\}| = 1$$
$$|\mathit{traces}(s, n = 1)| = |T|$$
$$\text{Induction hypothesis: } |\mathit{traces}(s, n)| = |T|^n$$
$$\text{Induction step: } |\mathit{traces}(s, n + 1)| = |T|^{n+1}$$

For an arbitrary path length n, each trace in *traces*(s, n) can be extended by one by adding any of the $|T|$ transitions to it, thus giving us

$$|\mathit{traces}(s, n + 1)| = |\mathit{traces}(s, n)| \cdot |T| = |T|^n \cdot |T| = |T|^{n+1}$$

□

Theorem 1 shows the upper limit to complete test suite size for LTSs, but when deriving the upper limit to complete test suite size for STSs variable valuations and constraints have to be taken into account as well. To find the maximum size of a complete test suite of traces of length n for an STS with a given number of transitions and variables but with unknown transition relation, we must assume again that there is only one state from which all transitions originate and loop back to. Furthermore, we must assume that each transition has a unique label, that there are no constraints on any transition, and that all variables may be re-valuated on every transition.

Theorem 2. *Take an STS $s = \langle Q, q_0, Var, Var_0, L, T\rangle$, consisting of a single state and each transition both originating from and targeting that state. Take every transition to have no constraints and to re-valuate every variable in Var. We use D_i as value domain for variable Var_i. An upper bound to complete test suite size for s is $|traces(s, n)| = (|T| \cdot \prod_{i=1}^{|Var|} D_i)^n$.*

Proof. For STS s we find that the size of $traces(s, n = 1)$ is $|T| \cdot \prod_{i=1}^{|Var|} D_i$, compared to $|T|$ in Theorem 1. Following the same proof as in Theorem 1, it is immediately clear that $|traces(s, n)| = (|T| \cdot \prod_{i=1}^{|Var|} D_i)^n$. □

The formulas in Theorems 1 and 2 give absolute maximum numbers of paths for LTSs and STSs respectively, using only the number of transitions in an STS and trace length. Using more information of an STS allows us to further constrain the upper bound of the size of test suites. A property that allows us to lower this upper bound is the maximum out degree of states in an STS, where the out degree of a state is the number of transitions originating from that state. The out degree of a state q is denoted by $\deg^+(q)$. The maximum out degree of all states in an STS s we will write as $\deg^+_{\max}(s)$.

Theorem 3. *Taking an STS s as described in Theorem 2 for which $deg^+_{max}(s)$ is known, an upper bound to complete test suite size for s is $|traces(s, n)| = (|deg^+_{max}(s)| \cdot \prod_{i=1}^{|Var|} D_i)^n$.*

Proof. When $\deg^+_{\max}(s)$ is known for an STS s, we see that $traces(s, n = 1) = \deg^+_{\max}(s) \cdot \prod_{i=1}^{|Var|} D_i$. Following the same proof as in Theorem 1 but replacing $|T|$ with $\deg^+_{\max}(s) \cdot \prod_{i=1}^{|Var|} D_i$, we find $|traces(s, n)| = (|\deg^+_{\max}(s)| \cdot \prod_{i=1}^{|Var|} D_i)^n$. □

Knowing the out degrees of all states does not help to find a significantly lower upper bound to test suite size. We illustrate this with the STSs shown in Fig. 3.1. These STSs have only one transition with high outgoing degree, but the outgoing degree of that state still dominates the expression for test suite size.

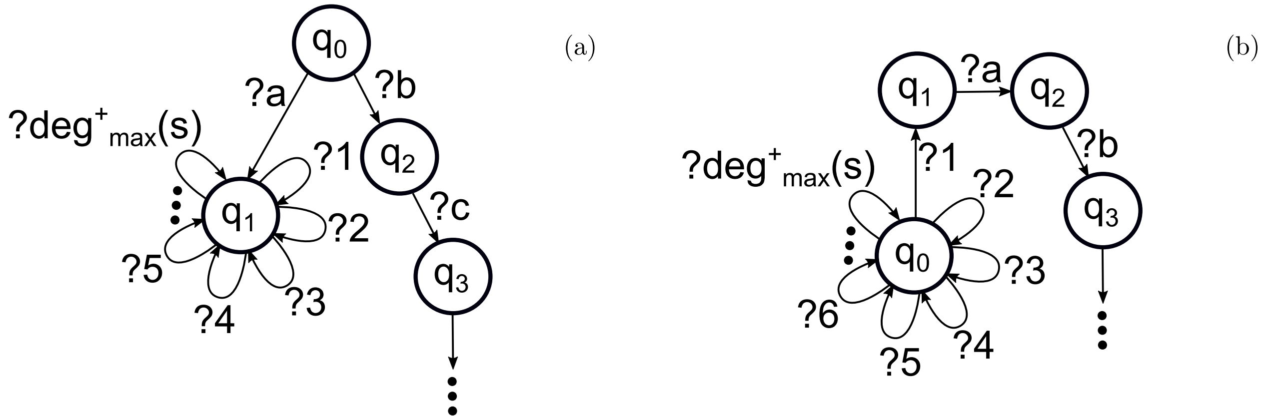

Figure 3.1: *Examples of STSs with low average outgoing degree, but large test suite size. For both STSs q_0 is the start state. For the STS in (a), $|traces(s, n)| = deg^+_{max}(s)^{n-1} + 1$, which is only off from the expression we found in Theorem 1 by a factor $deg^+_{max}(s)$. For the STS in (b), $|traces(s, n)| > (deg^+_{max}(s) - 1)^n$.*

Chapter 4

Implementation of test suite generator with data coverage

For this thesis we built an algorithm to generate complete test suites of traces of a given length and ensure data coverage, which we will describe in this chapter. For the code of the algorithm, see Appendix A. Our algorithm uses parts of the code-base of AMS. Our algorithm is divided into parts with distinct purposes: the first part collects all possible traces through a given STS (Section 4.1), the second collects the constraints on the variables for each trace (Section 4.2), the third solves for the domains of those variables (Section 4.3), and the last part generates the actual test cases and assigns values to the variables (Section 4.4). Our algorithm only interacts with an STS to derive the test suite, not with an implementation. Executing these tests cases on an implementation is outside of the scope of this work as that has been studied and implemented in previous works [8, 9].

4.1 Trace collection

In order to find all paths we perform a depth-first search on an STS. We perform this search by recursively calling a function that visits the states targeted by each of the transitions originating from the current state. When the given number of transitions with observable actions have been visited, an empty list is returned. The caller function then adds the last transition travelled to this list before returning it again. This is repeated until a list containing the full trace from start state up to that last state results, which is then added to the test suite. This is repeated for all states reachable through the given number of observable transitions. By doing a search through the STS, loops are unrolled implicitly. The test suite generated in this way is then an exhaustive and sound list of traces of the given length through the given STS (Section 3.1). At this point constraints are not taken into account yet, so some of the found traces may not be possible, e.g. if no variable values can be chosen to satisfy the constraints on the trace. Removing such traces becomes possible after solving the constraints on the traces. Furthermore, some traces may never be performed by a given SUT, e.g. if the SUT is not observed to perform the output action prescribed by a trace during testing. Since we do not execute our traces during the work for this thesis, these latter traces cannot be identified however.

4.2 Constraint collection

Once a trace has been found, its constraints are collected so they can later be solved. We do this by looping over the transitions comprising each trace from the last transition in the trace to the first. Any constraints encountered while iterating are copied, the copies are saved to a hash with the variable being constrained as key. Why the copying is necessary we discuss in the next paragraph. If multiple variables occur in a single constraint, the constraint is mapped to both variables.

When on a transition a variable is (re-)valuated, the domain of this variable is no longer constrained by constraints imposed by transitions that were taken earlier. To take this into account, we give the variable a unique name in those constraints that apply to that variable at the same time. For this it is necessary to save copies of constraints and assign the new name in the copies only; our constraints are objects, and renaming variables in a constraint for one trace would rename that variable for all traces. After renaming the variables in the constraints, the relevant constraints are removed from the hash and moved to the list of constraints on the trace. Constraints mapped to multiple variables are still renamed properly as the hash contains pointers to the copied constraints rather than individual instances of the constraints.

4.3 Solving for domains

For solving for the domains of our variables we use GNU Prolog, which is a native Prolog compiler which supports constraint solving over finite domains. The constraints on a trace are combined with AND-statements and translated to the format used by GNU Prolog. When the solver returns the domains for the variables in the constraints, the domains are mapped to the relevant variable name in a hash. If there is no solution, there are no variable valuations for which the trace is legal and the trace is removed from the test suite.

There is a number of flaws to GNU Prolog however, these are discussed by Mora Perdiguero [9], and he proposes means to circumvent them.

4.4 Variable valuations

Finally we assign values to our variables for each trace. For our data coverage criterion we seek to cover the minimum and maximum values of the label parameters on each transition they are (re)-valuated. We do not try to cover the minimum and maximum values of variables for each individual sequence of actions, nor do we necessarily want to test with different combinations of variable valuations. We do not attempt to cover values of state variables as they cannot be directly influenced and may not always be knowable. We do not attempt to cover all domain boundary values as this is discussed by Mora Perdiguero [9].

To have our test suite meet our data coverage criterion, we first find the maximum and minimum values for label parameters on each transition. For each trace we then pick the values we want to assign to the label parameters, where we first try to choose the minimum or maximum value for each label parameter. Per label parameter valuated per transition we keep track of whether the minimum and maximum values have been chosen for a trace. When a value must be chosen for a label parameter for which the minimum and maximum value are already covered, we choose a random value from its value domain instead. If there are minimum or maximum values of label parameters not covered after choosing label parameter values for all traces, we add more instances of traces that contain the relevant transition and ensure the coverage of the remaining values.

Chapter 5

Experiments on test suite generator

In this section we report on the performance of the algorithm developed as part of this thesis. First we discuss our experimental method (Section 5.1), then we present the results of the experiments (Section 5.2). Finally we discuss our results and compare them to results obtained in other works (Section 5.3).

5.1 Experimental method

With our algorithm we generated test suites for a complex STS, called the FEI model. This model represents the specification of a microscope. In this model the variables are tightly constrained, and specific combinations of variable values are required to follow certain paths. We perform our experiments on a Lenovo ThinkPad with 3.60 GiB RAM.

The metrics with which we analyzed the performance of our algorithm were test suite size and time required to build test suites. We also measured the time required to run the different parts of our algorithm.

Since we are only building a test suite but not running actual tests, we cannot report on results of testing. However, we have shown that our algorithm can generate complete test suites, and as such the tests should cover all possible traces up to the given depth.

5.2 Results

With our algorithm we generated test suites for the FEI model with traces of lengths $n = 0, 1, 2, 3$. We measured the time it took to generate test suites and measured the size of the test suites, and we present our findings in Table 5.1. Note that for $n = 3$ an error occured when requesting a response from Gnu Prolog, likely due to the response string being too long. As such that test suite was not completed and we report only on test suite size, and the time required to collect all traces and collect the constraints on all traces.

We see that our algorithm generates test suites at a rate in the order of one trace per second. Our measurements of time consumed by the different parts of our algorithm point out that by far the slowest process is constraint collection. After further measurements we found that specifically the copying of constraint objects takes a long time, taking 6.10 s of the total 6.35 s for test suites of traces with length $n = 1$, and 3.40×10^3 s of the total 3.48×10^3 s for test suites of traces with length $n = 2$.

We attempted to generate test suites with traces of length $n = 4$, but after collecting 329000 traces in $\sim$52 s the process froze and was killed by the kernel. Each time the process was terminated before the trace

Table 5.1: *Measurements on the generation of test suites for the FEI model by our algorithm.*

Number of observable transitions	Size of test suite	Total time	Trace collection time	Constraint collection time	Constraint solving time
0	1	2.69×10^{-4} s	9.08×10^{-5} s	6.18×10^{-5} s	1.16×10^{-4} s
1	15	6.35 s	4.99×10^{-4} s	6.12 s	0.23 s
2	3461	3.48×10^{3} s	0.25 s	3.42×10^{3} s	52.65 s
3	27835	–	2.43 s	3.31×10^{4} s	–

collection was completed, and as such the constraint collection and constraint solving were not started. The process was likely termination due to inefficient memory usage. We could reliably reproduce this result; on each run our algorithm would generate around 329000 traces in about 52 s, and then be terminated.

5.3 Discussion

In cases where new tests should be generated often, such as during the development of STSs, our algorithm is far too slow to be a viable testing solution. However, once the model has been verified to reflect the desired specifications, test suites only have to be generated once. In such cases the slowness of our algorithm becomes a smaller problem, and then our algorithm can grant certainty on whether an implementation conforms to its specification. Other testing methods, such as on-the-fly test generation, cannot grant such certainty.

We compare the performance of our algorithm to that of the algorithm produced by Ghaffari [8]. That work also describes a test suite generation algorithm, and the performance of that algorithm when generating test suites, also for the FEI model. When we compare our results, we find that our algorithms generate the same number of traces of a given length. Further, we find that our own algorithm takes 10–100 times longer to generate test suites than the algorithm produced by Ghaffari [8], while the latter algorithm also performed test execution during the reported periods of time.

We identified the part of our code that takes the greatest amount of time to be the copying of our constraints, during constraint collection. If we would not rename variables in the constraints themselves, we could remove the need for copying the constraints and thus significantly improve the performance of our algorithm. For this, we could rename the variables while creating constraints in the format accepted by GNU Prolog. To a degree, this would mix concerns, and as such produce smelly code, but the code would be far more functional than it currently is. Furthermore, the copying of objects can also be considered smelly.

Chapter 6

Overview of MBT tools

In this section we discuss a number of MBT tools. We focus tools that are still actively developed and maintained. For an overview of more MBT tools, see [10]. We find that no currently existing MBT tool can generate test suites that ensure high trace and data coverage at the same time.

The work in this thesis is based on *Axini Modelling Suite* (AMS)[11], which is a commercially available MBT tool. AMS generates test cases "on the fly"; test cases are generated during test execution. AMS generates these test cases by choosing one of the input actions that are possible according to the specification if there are any, and immediately sends that action to the SUT. Upon receiving an output action from the SUT, AMS verifies that the action is prescribed by the specification. The tool supports different test strategies, the most commonly used strategy achieves high transition coverage by choosing those transitions that have not been followed before during testing. This method allows for reaching high transition coverage, but the tool cannot ensure trace- or data coverage as yet. The work of this thesis aimed at enabling the tool to reach high trace- and data coverage, but our algorithms have not been added to the code base as yet.

The *free Model Based Testing tool* (fMBT) [12, 13, 14], like AMS, generates test cases on the fly. fMBT allows the user to choose from a number of testing heuristics to guide the test generation and coverage metrics to get reports on. The tool cannot not ensure nor measure data coverage, however.

Modbat[15, 16, 17, 18] is described by Memon [19] as "...an open-source tool based on extended finite-state machines specialized for testing the APIs of software. A Scala-based domain-specific language is used to create the models with features for probabilistic and non-deterministic transitions, component models with inheritance, and exceptions. Test cases are generated as sequences of method calls to the API that can be directly executed against the SUT." Modbat currently supports state- and transition coverage measurement, but seems to test randomly and does not ensure high coverages[15]. Data coverage is also not considered in this tool.

Conformiq Designer[20] is an MBT tool that supports multiple different testing goals[21], including all-path coverage and boundary value analysis[14, 22, 23]. However, it is unclear how Designer generates test to ensure all-path coverage without requiring infinite tests, and after performing boundary value analysis the tool does not valuate variables to their domain boundary values. How they do choose values is described in their manual[24]. Moreover, Designer does not support non-determinism[25], while our own algorithm is able to derive tests for non-deterministic systems.

TestCast is described by Marinescu *et al.* [26] as "...a commercial tool that provides testing for UML statecharts that encapsulate the system behavior, and it assumes structural test coverage criteria. Abstract test-cases are automatically designed and generated based on graph search techniques, which are converted to executable test-scripts. There is little scientific documentation about the tool...". On the website of the tool[27] it is specified that the tool uses test data imported and/or defined in model, as such the tool does not

generate its own variable values. Furthermore, TestCast is able to generate tests to cover all transitions[14], but does not appear able to generate tests as to cover trace coverage.

The company Smartesting has developed two MBT tools, *CertifyIt* (formerly called Test Designer[25]) and *Yest*. Yest is currently marketed on the Smartesting website[28], CertifyIt is no longer mentioned there and may have been deprecated. According to Jorgensen [14], CertifyIt generates test targets from a given model, then generates a test suite that covers each test target at least once. Test targets are generated according to specific coverage criteria, such as transition coverage[29] or decision coverage[30]. Jorgensen [14] does mention that test targets are defined for each of the paths in the model, suggesting that CertifyIt can generate test suites ensuring high trace coverage. However, it is not clear whether CertifyIt prevents indefinite testing by only accepting models with finite possible paths or in some other way. Yest is a fairly new tool and as such there is not a lot of information on it, in academic writing or otherwise. Yest is a light-weight MBT tool used for testing workflows and business rules[31]. For this purpose, the workflow is to be modeled in the tool as simply as possible, and the modelling happens with the testing goals in mind. Yest generates test cases based on different test selection criteria (decision tree or table coverage)[14]. It does not appear that either Yest or CertifyIt can generate tests to maximize data coverage.

MaTeLo by All4Tec[32], according to Costa *et al.* [33] "stands for Markov Test Logic and is a commercial tool that generates test sequences from a collection of states, transitions, classes of equivalence, types, sequences, global variables and test oracles using their user interface." The tool does support measuring transition coverage[34, 35], but otherwise there is little scientific documentation on this tool. The tool's website implies that the tool cannot generate its own variable values from the specification and needs user-defined data sets.

Graphwalker is an open-source model based test derivation tool, which cannot execute tests on its own. Graphwalker can derive tests for models that contain variables[36], however it is unclear whether variable values are determined by assignments on the transitions or whether Graphwalker is able to choose input values. While the tool can report on state- or transition coverage, the tool cannot report on data coverage, and as such tests derived by Graphwalker cannot ensure high data coverage either.

While the *BPM-X* tool of TransWare AG[37] does appear to be a MBT tool, it is used with business process models rather than for software testing. There is little information to be found on this tool, and while the BPM-X website does state that tests are derived with certain coverage criteria in mind, it is not specified which criteria these are. BPM-X appears to rely on other business process modelling tools to execute the tests it generates.

The *MBTsuite*[38] tool can generate tests using the "full path coverage" strategy[23, 14], but it is unclear how exactly this strategy works. For instance, it is nowhere discussed how a test suite generated by MBTsuite with the full path coverage strategy is prevented from becoming infinitely large. Furthermore, data coverage is never mentioned as one of the features of MBTsuite, which may mean that the tool is not able to generate test suites with the purpose of ensuring data coverage, or even that the tool does not support models with variables at all.

The *Tricentis Tosca*[39] testing platform uses models to automatically generate tests, and has integrated test data management and analytics tools[40]. Despite the apparent commercial success of this platform there is little documentation on this tool, academic or otherwise. The Tricentis Tosca website suggests that the platform focuses on testing with data from large databases, supplied by the user. With that data, Tricentis Tosca can then generate tests according to different data coverage strategies[41]. It appears that the platform uses fairly simple models that model a sequence of calculations rather than a sequence of actions performed upon or by a SUT. Futhermore, Tricentis Tosca is never mentioned to generate test suites that ensure state, transition or trace coverage.

Chapter 7

Conclusions

We have developed and implemented an algorithm to automatically generate test suites from STSs. We have proven that our approach yields test suites that can, when executed on an implementation, point out whether an implementation conforms to specifications modeled by a given STS. In practice we limited our test suites in order to make them finite, by limiting the length of our test traces. Furthermore, the test suites generated by our approach ensure coverage of the minimum and maximum values of label parameters in a given STS, on each transition where those parameters are valuated. Additionally, we have presented methods for finding upper bounds to complete test suite size, while using only little information of an STS, such as number of transitions or the maximum outgoing degree.

We have also looked into other MBT tools, and we found none that generates test suites to ensure both trace- and data coverage while allowing for non-determinsm in models.

We analyzed the performance of our implementation by generating test suites for a complex model and measuring the size of the generated test suites and the time it took to generate them. We find that our algorithm is slow in generating test suites, but we identified the problem to be the process of copying constraints.

7.1 Future work

Our algorithm is currently too slow to be a valuable addition to Axini TestManager. If we can eliminate the need of copying our constraints, we can improve the performance of our code to match or surpass that of the algorithm produced by Ghaffari [8].

In our current implementation, we only start building the test traces once our search algorithm reaches the predefined search depth. An alternative way to find traces and immediately take reachability into account would be to grow traces starting at the start state. At every step (or alternatively only at branching steps) the constraints could be solved, so traces that are not reachable can be identified and removed from the list of traces at that point. In this way, traces that are unreachable are not extended during the search and therefore do not clutter the test suite. This requires solving constraints multiple times for each trace, however, which is computationally expensive.

Growing traces from the start state also allows us to set a maximum to the number of times a loop is traversed, by keeping track of how often states appear in the trace. That would allow choosing to terminate traces that visit the same state too often, according to a user-defined metric.

The test suites generated by our code cannot be executed as yet. It would be an interesting challenge to implement test execute for our test suites, as such an implementation should handle any non-determinism in SUTs, and SUTs may simply happen to never follow specific test cases. While these problems have been considered in previous work[8], there is no approach that ensures data coverage in non-determinate systems to the best of our knowledge.

Our data coverage criterion is somewhat limited, since we only cover the minimum and maximum values of label parameters. It may be interesting to expand our algorithm to generate test suites that test data more exhaustively. We could simply follow the approach of Mora Perdiguero [9], for which the coverage criterion is to test with all variable domain boundary values. Alternatively, we could test with different combinations of variable valuations, or study the subject more thoroughly and find different meaningful data coverage criteria.

Acknowledgements

I would like to thank Axini for hosting the work done for this thesis, and its team for many insightful discussions. I would like to thank Ana Oprescu for the patience and insight with which she guided me during my work on this thesis.

Bibliography

[1] Robert V Binder. 2011 model-based testing user survey: Results and analysis', 2012.

[2] Jan Tretmans. Test generation with inputs, outputs, and quiescence. In *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 127–146. Springer, 1996.

[3] Jan Tretmans. Model based testing with labelled transition systems. In *Formal methods and testing*, pages 1–38. Springer, 2008.

[4] Lars Frantzen, Jan Tretmans, and Tim AC Willemse. A symbolic framework for model-based testing. In *Formal approaches to software testing and runtime verification*, pages 40–54. Springer, 2006.

[5] Vlad Rusu, Lydie Du Bousquet, and Thierry Jéron. An approach to symbolic test generation. In *International Conference on Integrated Formal Methods*, pages 338–357. Springer, 2000.

[6] Bertrand Jeannet, Thierry Jéron, and Vlad Rusu. Model-based test selection for infinite-state reactive systems. In *International Symposium on Formal Methods for Components and Objects*, pages 47–69. Springer, 2006.

[7] Lars Frantzen, Jan Tretmans, and Tim AC Willemse. Test generation based on symbolic specifications. In *International Workshop on Formal Approaches to Software Testing*, pages 1–15. Springer, 2004.

[8] Ardavan Ghaffari. *Trace Coverage Strategy for Symbolic Transition Systems*. Master's thesis, Universiteit van Amsterdam, the Netherlands, 2016.

[9] Jorge Mora Perdiguero. *Development of an efficient data coverage strategy for testmanager*. Master's thesis, ETSI_Informatica, 2016.

[10] Model based testing (mbt). URL: `http://home.mit.bme.hu/~micskeiz/pages/modelbased_testing.html`. Accessed: 2019-07-07.

[11] Axini. URL: `https://www.axini.com/nl/`. Accessed: 2019-07-07.

[12] free model based testing tool. URL: `https://github.com/intel/fMBT`. Accessed: 2019-07-07.

[13] Wenbin Li, Franck Le Gall, and Naum Spaseski. A survey on model-based testing tools for test case generation. In *International Conference on Tools and Methods for Program Analysis*, pages 77–89. Springer, 2017.

[14] Paul C Jorgensen. *The craft of Model-Based testing*. Auerbach Publications, 2017.

[15] Cyrille Valentin Artho, Armin Biere, Masami Hagiya, Eric Platon, Martina Seidl, Yoshinori Tanabe, and Mitsuharu Yamamoto. Modbat: A model-based api tester for event-driven systems. In *Haifa Verification Conference*, pages 112–128. Springer, 2013.

[16] Cyrille Artho, Martina Seidl, Quentin Gros, Eun-Hye Choi, Takashi Kitamura, Akira Mori, Rudolf Ramler, and Yoriyuki Yamagata. Model-based testing of stateful apis with modbat. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 858–863. IEEE, 2015.

[17] Modbat. URL: `http://fmv.jku.at/modbat/`. Accessed: 2019-07-07.

[18] Modbat. URL: `https://github.com/cyrille-artho/modbat/`. Accessed: 2019-07-07.

[19] A. Memon. *Advances in Computers*. Number v. 108 in Advances in Computers. Elsevier Science, 2018. ISBN 9780128151204. URL: `https://books.google.nl/books?id=tJY-DwAAQBAJ`.

[20] Conformiq designer. URL: `https://www.conformiq.com/products/conformiq-designer/`. Accessed: 2019-07-07.

[21] Muhammad Bilal Ahmad Khan and Song Shang. Evaluation of model based testing and conformiq qtronic, 2009.

[22] Hartmut Lackner, Martin Thomas, Florian Wartenberg, and Stephan Weißleder. Model-based test design of product lines: Raising test design to the product line level. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 51–60. IEEE, 2014.

[23] Jens Grabowski, Victor Kuliamin, Alain-Georges Vouffo Feudjio, Antal Wu-Hen-Chang, and Milan Zoric. Towards the usage of mbt at etsi. *arXiv preprint arXiv:1303.1007*, 2013.

[24] Conformiq manual. URL: `https://www.conformiq.com/wp-content/uploads/2015/02/conformiq-manual.pdf`. Accessed: 2019-07-07.

[25] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil Mcminn, Antonia Bertolino, *et al.* An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8): 1978–2001, 2013.

[26] Raluca Marinescu, Cristina Seceleanu, Hèléne Le Guen, and Paul Pettersson. A research overview of tool-supported model-based testing of requirements-based designs. In *Advances in Computers*, volume 98, pages 89–140. Elsevier, 2015.

[27] Testcast. URL: `https://elvior.com/model-based-testing/#section-features`. Accessed: 2019-07-07.

[28] Smartesting. URL: `https://www.smartesting.com/?lang=en`. Accessed: 2019-07-07.

[29] Frédéric Dadeau, Kalou Cabrera Castillos, Yves Ledru, Taha Triki, German Vega, Julien Botella, and Safouan Taha. Test generation and evaluation from high-level properties for common criteria evaluations–the tasccc testing tool. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 431–438. IEEE, 2013.

[30] Kalou Cabrera Castillos, Frédéric Dadeau, Jacques Julliand, and Safouan Taha. Measuring test properties coverage for evaluating uml/ocl model-based tests. In *IFIP International Conference on Testing Software and Systems*, pages 32–47. Springer, 2011.

[31] Bernard Elodie, Ambert Fabrice, Legeard Bruno, and Bouzy Arnaud. Lightweight model-based testing for enterprise it. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 224–230. IEEE, 2018.

[32] All4tec. URL: `https://all4tec.com/`. Accessed: 2019-07-07.

[33] Ulisses Araújo Costa, Daniela da Cruz, and Pedro Rangel Henriques. Automatic test generation for space. *arXiv preprint arXiv:1206.5104*, 2012.

[34] A Guiotto, B Acquaroli, and A Martelli. Matelo: automated testing suite for software validation. In *DASIA 2003-Data Systems In Aerospace*, volume 532, 2003.

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

[36] Graphwalker. URL: `http://graphwalker.github.io/`. Accessed: 2019-07-07.

[37] Transware ag. URL: `www.transwareag.com`. Accessed: 2019-07-07.

[38] Mbtsuite. URL: `https://mbtsuite.com/`. Accessed: 2019-07-07.

[39] Tricentis tosca. URL: `https://www.tricentis.com/products/automate-continuous-testing-tosca/`. Accessed: 2019-07-07.

[40] Joachim Herschmann and T Murphy. Magic quadrant for software test automation, 2016.

[41] Tricentis tosca documentation. URL: `https://documentation.tricentis.com/en/1130/content/testcase_design/instance_combinatorics.htm`. Accessed: 2019-07-07.

Appendix A

Algorithm for complete test suite generation

A.1 Initialization

```
class CompleteTraceBuilder
  attr_reader :traces
  attr_reader :sts
  # Given a source file name, prepares for extracting all traces
  def initialize(aml_source_filename_or_model, max_depth=30)
    if aml_source_filename_or_model.is_a? String
      aml_source = File.read(aml_source_filename_or_model)
      model, model_parts_to_labels = Aml::Interpreter.interpret(aml_source)
      @sts = model.stss[0]
    else
      @sts = aml_source_filename_or_model.stss[0]
    end

    @solver = Treesolver::Api.v(2)
    @solver.seed = rand(2**60)

    @max_depth = max_depth
    @traces = nil
  end

  def main
    collect_traces
    collect_constraints
    solve_constraints
    return @traces
  end
```

```
    def collect_traces
      trace_collector = OfflineTraceCollector::TraceCollector.new(@sts,
                                                                  @solver,
                                                                  @max_depth)
      @traces = trace_collector.recursive_walker
    end

    def collect_constraints
      constraint_collector = OfflineTraceCollector::ConstraintCollector.new(@sts)
      @traces.map do |trace|
        constraint_collector.assign_path_constraints_to_updates(trace)
      end
    end

    def solve_constraints
      constraintsolver = OfflineTraceCollector::ConstraintSolver.new(@solver)
      @traces.each do |trace|
        constraintsolver.main(trace, @sts)
      end
    end
  end

class Trace
  attr_accessor :steps
  attr_accessor :constraints
  attr_accessor :query
  attr_accessor :domains
  attr_accessor :update_parameter_names
  attr_accessor :previous_values
  def initialize(steps=[],
                 constraints=[],
                 query=nil,
                 domains={},
                 update_parameter_names=[])
    @steps = steps
    @constraints = constraints
    @query = query
    @domains = domains
    @update_parameter_names = update_parameter_names
    # source for the following workaround:
    # https://mensfeld.pl/2016/09/ruby-hash-default-value-be-cautious-when-you-use-it/
    @previous_values = Hash.new { |h, k| h[k] = [] }
  end
end

class Step
  attr_accessor :transition
  attr_accessor :constraints
  attr_accessor :query
  attr_accessor :domains
  attr_accessor :update_parameter_names
```

```
  def initialize(transition=nil,
                 constraints=[],
                 query=nil,
                 domains=nil,
                 update_parameter_names=[])
    @transition = transition
    @constraints = constraints
    @query = query
    @domains = domains
    @update_parameter_names = update_parameter_names
  end

  def set_constraints(constraints)
    @constraints = constraints
  end
end


# Source: https://stackoverflow.com/a/1465787
def deep_copy(obj)
  Marshal.load(Marshal.dump(obj))
end
```

A.2 Trace collection

```
  # The main function, recursive_walker, performs a depth first search through a
  # given STS, returning all possible paths through it, paths with a given
  # maximum length.
  class TraceCollector
    def initialize(sts, solver, max_depth = 30)
      @sts = sts
      @solver = solver
      @return_states = []
      @max_depth = max_depth
    end

    # Performs a depth first search into an sts. Finds all leaf states and
    # states at max_depth, and returns all paths leading to those states.
    # @return [Array<Array<Transition>>]: array of all paths leading to leaf
    # states and states at depth max_depth.
    def recursive_walker(state=@sts.start_states.first, current_depth=0)
      child_paths = []
      transitions = state.transitions.to_a
      if transitions.empty? or current_depth >= @max_depth
        trace = Trace.new
        return [trace]
      else
        transitions.each do |transition|
          trace_elongation = transition.observable? ? 1 : 0
          child_state = transition.target
          child_state = handle_state_type(child_state)
```

```
        recursive_walker(child_state, current_depth+trace_elongation).each do |trace|
          trace.steps.unshift(Step.new(transition = transition))
          child_paths.append(trace)
        end
      end
      child_paths
    end
  end

  # In recusive_walker, Switchstates need to be handled different from most
  # states, which happens here.
  # The child states of Switchstates have to be found through the child
  # sts of the Switchstate, instead of through child_transition.target
  # Return states are saved in order to return to them once a child sts
  # reaches its Finalstate.
  def handle_state_type(child_state)
    case child_state
    when Sts::SwitchState
      child_sts = child_state.to_sts
      if child_sts.terminating?
          @return_states.append(child_state.return_state)
      end
      child_sts.start_states.first
    when Sts::FinalState
      @return_states.pop
    else
      child_state
    end
  end
end
```

A.3 Constraint collection

```
class ConstraintCollector
  def initialize(sts)
    @sts = sts
  end

  # Collects and assigns constraints to transitions where the constrained
  # state variables are updated. Variables are renamed so that its value
  # domains before and after updates can be discerned.
  def assign_path_constraints_to_updates(trace)
    constraints = {}
    var_num = 0
    trace.steps.reverse_each do |step|
      transition = deep_copy(step.transition)

      constraints_containing_variable = []
      # Flattens struct nodes into ref nodes.
      structnode_finder(transition.parsed_constraint)
```

```ruby
if transition.update
  var_num += 1
  # Flattens struct nodes into ref nodes.
  structnode_finder(transition.parsed_update)
  # Splits constraints on &&'s.
  updates = constraint_flattener(transition.parsed_update)
  # Finds the variables that will be updated, and as such the variables
  # that need to be renamed in order to be solved, later.
  updated_variables = updates.map {|assign_node|
                                    assign_node.children[0].variable_name
                                  }

  update_variable_names = transition.parsed_update.variable_names.deep_dup
  update_constraints = constraint_flattener(transition.parsed_update)
  transition_constraints = constraint_flattener(transition.parsed_constraint)

  # Gathers constraints on transitions further along the trace that
  # contain variables that are updated on the current transition.
  previous_constraints = updated_variables.each do |variable|
    if constraints[variable]
      constraints_containing_variable.concat(constraints[variable])
      constraints[variable] = []
    end
  end

  # Renames the variables that will be updated during this transition in
  # the constraints further along the path.
  renamed_constraints = constraints_containing_variable.map do |constraint|
    normalize_parameters_and_variables(constraint, updated_variables, var_num)
  end

  # Renames the variables that will be updated during this transition in
  # the updates of this transition. Only the left hand side of these
  # 'constraints' should be renamed, hence the 'lhs = true'.
  renamed_updates = updates.map do |constraint|
    normalize_parameters_and_variables(constraint,
                                       updated_variables,
                                       var_num,
                                       true)
  end

  update_variable_names = transition.parsed_update.variable_names.deep_dup
  parameters_to_normalize = update_variable_names &
                            @sts.label_parameter_names.to_a
  transition_constraints = constraint_flattener(transition.parsed_constraint)

  # Renames the label parameters that occur in the update and constraint
  # of the current transition.
  renamed_constraints = (renamed_constraints +
                         renamed_updates +
```

```
                         transition_constraints).map do |constraint|
        normalize_parameters_and_variables(constraint,
                                           parameters_to_normalize,
                                           var_num)
      end

      # Writes the constraints to the relevant object attributes. Writes the
      # new names of the parameters to the objects as well, so that you know
      # what variable has to be given a value at what step in a trace.
      update_parameter_names = parameters_to_normalize.map{|param|
                                                   param+"_#{var_num}" }
      step.update_parameter_names = update_parameter_names
      trace.update_parameter_names.concat(update_parameter_names)
      trace.constraints.concat(renamed_constraints)
    end

    constraints = constraint_collector(constraints, transition)
  end
  trace
end

# Collects constraints from the given transition and adds them to the Hash
# constraints_of_variables.
# Helper function for assign_path_constraints_to_updates
def constraint_collector(constraints_of_variables, transition)
  constraints = constraint_flattener(transition.parsed_constraint)+
                constraint_flattener(transition.parsed_update)
  constraints.each do |constraint|
    constraint.variable_names.each do |variable|
      if !constraints_of_variables[variable]
        constraints_of_variables[variable] = [constraint]
      else
        constraints_of_variables[variable].append(constraint)
      end
    end
  end
  constraints_of_variables
end

# Together with structnode_finder, replaces struct nodes with reference
# nodes, in order to handle them properly during solving.
def flatten_ref_nodes(node)
  case node
  when Aml::Dumont::StructReferenceNode
    child_names = node.children.map do |child|
      flatten_ref_nodes(child)
    end
    Aml::Dumont::ReferenceNode.new(:ref, [child_names.join('.')], :integer )
  when Aml::Dumont::ReferenceNode
    node.reference
  when Aml::Dumont::LiteralNode
```

```
    node.value
  end
end

# Together with flatten_ref_nodes, replaces struct nodes with reference
# nodes, in order to handle them properly during solving.
def structnode_finder(node)
  if node.is_a?(Aml::Dumont::Node) && node.children
    node.children = node.children.map do |child|
      if child.is_a? Aml::Dumont::StructReferenceNode
        flatten_ref_nodes(child)
      else
        structnode_finder(child)
      end
    end
  end
  node
end

# Source of idea: Ardavan Ghaffari:
# /sts-engine/lib/strategies/trace_coverage.rb, branch: tracecoverage
# Renames variables in order to discern variable domains before and
# after they are updated.
def normalize_parameters_and_variables(node,
                                       names_to_normalize,
                                       var_num,
                                       lhs_only=false)
  newnode = case node
  when Aml::Dumont::StructReferenceNode
    if names_to_normalize.include?(node.children[0].variable_name) &&
       names_to_normalize.include?(node.children[1].value)
      node.children[1].value << "_" + var_num.to_s
    end
    node
  when Aml::Dumont::ReferenceNode
    if names_to_normalize.include?(node.variable_name)
      node.variable_name << "_" + var_num.to_s
    end
    node
  when Aml::Dumont::AssignNode && lhs_only
    if names_to_normalize.include?(node.children[0].variable_name)
      other_names_to_normalize = names_to_normalize-
                                 [node.children[0].variable_name]
      normalize_parameters_and_variables(node.children[0],
                                         names_to_normalize,
                                         var_num,
                                         lhs_only)
    else
      node
    end
  else
```

```
      normalize_child_parameters_and_variables(node,
                                               names_to_normalize,
                                               var_num,
                                               lhs_only)
    end
    newnode
  end

  # Helper function for normalize_parameters_and_variables
  def normalize_child_parameters_and_variables(node,
                                               names_to_normalize,
                                               var_num,
                                               lhs_only=false)
    if node.children.present?
      node.children.each do |child|
        if child.is_a? Aml::Dumont::Node
          normalize_parameters_and_variables(child,
                                             names_to_normalize,
                                             var_num,
                                             lhs_only)
        end
      end
    end
    node
  end

  # Flattens the tree structure of a given and- or update node into a list of nodes
  def constraint_flattener(constraint)
    constraints = case constraint
    when nil then []
    when Aml::Dumont::AndNode
      constraint.children.map do |child|
        constraint_flattener(child)
      end.flatten
    when Aml::Dumont::UpdateNode
      constraint.children.map do |child|
        constraint_flattener(child)
      end.flatten
    else [constraint]
    end
  end

end
```

A.4 Constraint solving

```
class ConstraintSolver
  def initialize(solver=nil)
    if solver
      @solver = solver
    else
```

```
    solver = Treesolver::Api.v(2)
    solver.seed = rand(2**60)
  end
end

# Solves constraints and assigns the resulting parameter domains to
# the steps where those parameters are updated.
# Does so by constructing treesolver queries from aml constraints,
# then having treesolver solve those.
# Returns only label parameter domains.
def main(trace, sts)
  @trace = trace
  label_parameters = sts.stimulus_labels.to_a[0].parameters.map{|parameter|
                                                              parameter.name}
  trace.query = combine_queries(constraints_to_queries(trace.constraints) +
                                initial_valuations_to_queries(sts.vars))
  trace.domains = solve_for_domains(trace.query,label_parameters) if trace.query
end

# Translates a Aml::Dumont::Node type constraint into a GNUProlog query.
def constraints_to_queries(constraints)
  constraints.flat_map do |constraint|
    case constraint
    when Aml::Dumont::AssignNode
      @solver.new_query(:==, *constraints_to_queries(constraint.children))
    when Aml::Dumont::BinaryNode , Aml::Dumont::UnaryNode
      @solver.new_query(constraint.label,
                        *constraints_to_queries(constraint.children))
    when Aml::Dumont::UpdateNode
      constraint.children.flat_map {|update|
        @solver.new_query(:==,
                          *constraints_to_queries(update.children))
      }
    when Aml::Dumont::StructReferenceNode
      var_name = constraint.children[0].variable_name() +
                 '.' +
                 constraint.children[1].value
      new_query = @solver.new_variable_query(var_name, :integer)
    when Aml::Dumont::ReferenceNode
      new_query = @solver.new_variable_query(constraint.children[0], :integer)
    when Aml::Dumont::LiteralNode
      @solver.new_query(constraint.value, constraint.type)
    else
      nil
    end
  end
end

# Combines a list of queries into a list containing a single, long AND query.
# Returns the single AND query.
def combine_queries(queries)
```

```
    while queries.length > 1
      queries[0] = @solver.new_query(:'&&', queries[0], queries.pop)
    end
    queries[0]
  end

  # Takes a GNUProlog query and feeds it to GNUProlog to be solved.
  # Returns a hash {label_parameter => [parameter domain]}
  def solve_for_domains(query, label_parameters)
    codec = Treesolver::Codec.new
    connection = Treesolver::Connection.new
    connection.start
    connection.receive_response.join("\n")  # Receives start message, empties buffer.

    query_message = query.to_treesolver_format(codec)
    connection.write_string(query_message+".\n")
    response = connection.receive_response

    variable_domains = {}
    (0..response.length-2).each do |i|
      if response[i].include?(" = _#")  # Filters single value domains.
        domain = response[i][0..-2].split("(")[1]
        domain = domain.split(":").map do |subdomain|
          splitdomain = subdomain.split("..").map{|value| value.to_i}
          if splitdomain.length == 1
            splitdomain[0]..splitdomain[0]
          else
            Range.new(*splitdomain)
          end
        end
      else
        domain = [response[i].split(" ")[-1].to_i]
      end
      variable = query.variables[i].name
      variable_domains[variable] = domain
    end
    variable_domains.select{|key, value| @trace.update_parameter_names.include?(key)}
  end

  def initial_valuations_to_queries(vars)
    initialized_vars = vars.keys.select{|key| vars[key].value}-["current_time"]
    initial_queries = initialized_vars.map do |var|
      variable_query = @solver.new_variable_query(var, :integer)
      literal_query = @solver.new_query(vars[var].value, vars[var].type_specification)
      @solver.new_query(:==, variable_query, literal_query)
    end
  end

end
```