Coverage-Based Testing with Symbolic Transition Systems

Petra van den Bos [1] * and Jan Tretmans [1,2] *

[1] Institute for Computing and Information Sciences,
Radboud University, Nijmegen, the Netherlands;
[2] ESI (TNO), Eindhoven, the Netherlands
{petra, tretmans}@cs.ru.nl

Abstract. We provide a model-based testing approach for systems comprising both state-transition based control flow, and data elements such as variables and data-dependent transitions. We propose test generation and execution, based on model-coverage: we generate test cases that aim to reach all transitions of the model. To obtain a test case reaching a certain transition, we need to combine reachability in the control flow, and satisfiability of the data elements of the model. Concrete values for data parameters are generated on-the-fly, i.e., during test execution, such that received outputs from the system can be taken into account for the inputs later provided in test execution. Due to undecidability of the satisfiability problem, SMT solvers may return result 'unknown'. Our algorithm deals with this explicitly. We implemented our method in Maude combined with Z3, and use this to demonstrate the applicability of our method on the Bounded Retransmission Protocol benchmark. We measure performance by counting the number of inputs and outputs needed to discover bugs in mutants, i.e., in non-conforming variants of the specification. As a result, we find that we perform 3 times better, according to the geometric mean, than when using random testing as implemented by the tool TorXakis.

1 Introduction

Software testing involves experimentally checking desired properties of a software product by systematically executing that software. The software is stimulated with test inputs, and the actual outputs are compared with expected outputs. Model-Based Testing (MBT) is a form of black-box testing where the software being tested, called System Under Test (SUT), is tested for correctness with respect to a model. The model serves as a formal specification for the SUT, prescribing the behaviour that the SUT shall, and shall not, exhibit. Moreover, the model is the basis for the algorithmic generation of test cases and for the evaluation of actual test outputs.

Many of the modelling formalisms used for MBT are based on some kind of state-transition model: states in the model represent an abstraction of the states

* Funded by the Netherlands Organisation of Scientific Research (NWO): 13859.

of the system, and transitions between states represent the actions that the system may perform. Depending on the kind of state-transition model, an action can be the acceptance of an input, the production of an output, an internal step of the system, or the combination of a trigger and the corresponding response.

Plain state-transition formalisms, though a powerful semantic model, are not powerful enough to specify present-day systems. Such systems, in addition to state-transition-modelled control flow, involve complicated data objects, operations on data, inputs and outputs parameterized with data, and conditions on data guarding the enabling of transitions. Consequently, many state-transition formalisms have been extended with the ability to deal with data, variables, parameters, and conditions, often referred to as extended, or symbolic state-transition models.

For a plain state-transition model, test generation corresponds to graph operations on the model, such as selecting a finite path (in case of deterministic models), a tree (in case of nondeterministic models), or a tour through the model. The extension with data, however, complicates the test-generation process. A naive approach of unfolding data, i.e., encoding all possible data values in transitions, and thus mapping the data to a plain state-transition model, leads to the infamous state-space explosion problem. A second disadvantage of unfolding is that all structure and information available in the data definitions and constraints is lost. This information can be very useful in the test selection process. The latter disadvantage also applies to the converse, when mapping all state-transition information to data, i.e., to state variables. Consequently, a more sophisticated way of combining state-transition information with data is necessary where the differences and subtle interplay between the two are taken into account. A common approach is to combine graph-based state-transition system manipulation with the symbolic treatment of data and variables.

In this paper, we present theory, an implementation, and an application of such a model-based testing approach, that combines state-transition-based control flow and symbolic treatment of data and variables. Our models are expressed as *Symbolic Transition Systems* (STS) [8,9], which combine Labelled Transition Systems (LTS) with an explicit notion of data, variables, and data-dependent conditions, founded on first order logic. As the basis for test generation we use the **ioco**-testing theory for LTS [23,24]. The *implementation* or *conformance relation* **ioco** is a formal relation between SUTs and models, that defines precisely when an SUT is correct with respect to its model. The **ioco**-testing theory provides a test generation algorithm that is *sound* and *exhaustive*, i.e., the (possibly infinitely many) test cases generated from an LTS model detect all and only **ioco**-incorrect implementations.

We lift **ioco**-test generation to the symbolic level, analogous to [8]. In addition to [8], we generate test cases that satisfy *switch coverage*, i.e., all symbolic transitions of the STS model are covered in the test cases (as far as nondeterminism allows). Switch coverage is thus a way of test selection, which is sound, but usually not exhaustive.

As an intermediate structure we define a *symbolic execution graph*, which incorporates classical symbolic concepts like path conditions and reachability, and which, in addition, is adapted to nondeterministic STS. We select finite paths, i.e. *test purposes*, from the symbolic execution graph, that guarantee switch coverage. After that, data values are selected in an *on-the-fly* test generation and execution algorithm. This algorithm takes into account previous input and output values, making it more flexible than selecting all input values beforehand.

We define measures for achieved switch coverage, both *a priori*, i.e., during test-purpose generation, and *a posteriori*, i.e., after test execution. Due to unsatisfiable constraints and SUT nondeterminism, full coverage may not always be achieved and a posteriori coverage may be lower than a priori coverage.

To implement the method, we tie together *Maude*, a language and tool set for rewriting systems [5], and *Z3*, an SMT-solver [6]. We encode models in the Maude language, from which test purposes satisfying switch coverage are generated. In this step Maude internally uses Z3 to check constraints. A Python program takes a test purpose and implements the on-the-fly test generation and execution algorithm, where Z3 is used again to generate witnesses serving as input data values satisfying the constraints of the test purpose. Since satisfiability is undecidable, Z3 may produce an 'unknown' result that we explicitly take into account in our algorithm.

The Bounded Retransmission Protocol (BRP), a benchmark in protocol verification and testing [13], is used as a case study. We compare our switch-coverage-driven test generation method with random path, on-the-fly test generation by the MBT tool TorXakis [22,25]. We show that TorXakis in total needs 8 times more inputs and outputs to discover bugs in mutants, i.e., in non-conforming variants of the specification. According to the geometric mean, we find that we perform 3 times better than TorXakis.

Overview In Section 2, we give preliminaries on LTS, **ioco**, data specifications, STS, and the semantics of STS. Symbolic execution graphs are defined in Section 3. Section 4 introduces switch coverage, provides the main on-the-fly test-generation and execution algorithm, and proves soundness. The implementation in Maude and Z3 is presented in Section 5, and the BRP case study is discussed in Section 6. Section 7 concludes, and mentions open issues and future work.

Related work The technique of symbolic execution was originally applied on programs [18] and applied, among others, for white-box testing [12]. Later on, it has found its way into other fields, such as model-based testing.

Gaston et al. [11] study model-based testing based on Symbolic Transition Systems, as we do. An important difference is that their work restricts output-parameter values to functions over constants and input parameters, i.e., expected output values can always be predicted. This implies that nondeterminism or uncertainty in output parameters cannot be modelled. In the area of Extended Finite State Machines similar restrictions are made [10,14,20,21]. The test generation in [11] is guided by test purposes, which are finite parts of the symbolic execution of the STS. Originally assumed to be given [17], test purposes in [11]

are generated from the model according to two criteria: (*i*) a maximum length on the sequences of executed switches, which is coarser than our switch coverage, or (*ii*) exclusion of 'redundant' parts of symbolic execution, e.g., a loop of switches in an STS is only executed once, which is what our switch coverage achieves too, but in general our approach could benefit from this exclusion of redundant behaviours.

The Guarded Labeled Assignment Systems (GLAS) models of [26] are very similar to our STS: the syntactical definition differs, their semantics in terms of symbolic executions are closely related. The paper shows this by analyzing the relation between **ioco** for STS and *i/o-refinement* for GLAS. No test generation method, however, is proposed.

Our work mainly builds on [8,9], except that we do not include internal τ switches. We extend the on-the-fly, random-path test generation of those papers with switch-coverage-driven test selection. In addition, [9] compares a couple of coverage measures: state coverage, location coverage, and symbolic-state (see Section 3) coverage. For full, semantic state coverage, all possible combinations of location and variable values have to be covered, which is usually not feasible. For location coverage only all locations have to be covered, independent from variable values; location coverage is implied by our switch coverage. For symbolic-state coverage each symbolic state must be covered, which can usually only be achieved up to some length n of test cases. Full switch coverage can be achieved with symbolic-state coverage if n is chosen high enough, i.e., n should be at least as long as the longest test purpose, which causes it to be more costly than our switch coverage.

2 Preliminaries

2.1 Labeled Transition Systems

In this section, we give a summary of theory on Labeled Transition Systems and the conformance relation **ioco**. Definitions are a bit simpler than in [24], as we restrict to systems without the internal, unobservable τ transitions.

Definition 1. *A* Labeled Transition System *(LTS) with inputs and outputs is a tuple* $(Q, q_0, \Sigma_I, \Sigma_O, T)$ *where:*

- Q *is a set of states, and* $q_0 \in Q$ *is the initial state,*
- Σ_I *and* Σ_O *are sets of input and output labels, respectively, with* $\Sigma_I \cap \Sigma_O = \emptyset$*,*
- $T \subseteq Q \times \Sigma \times Q$ *is the transition relation, where we write* $\Sigma = \Sigma_I \cup \Sigma_O$*.*

If a state $q \in Q$ has no outgoing transitions with an output label, then we say that q is *quiescent*, denoted $\delta(q)$. This is handled in an explicit way by **ioco**, by adding a self-loop transition with special output label δ.

For an LTS $(Q, q_0, \Sigma_I, \Sigma_O, T)$ with $q \in Q$, $Q' \subseteq Q$, $\mu \in \Sigma \cup \{\delta\}$, $\sigma \in (\Sigma \cup \{\delta\})^*$, and ϵ the empty sequence, we define:

$$\begin{aligned}
\mathsf{init}(Q') &= \bigcup_{q \in Q'} \{x \in \Sigma \mid \exists q' \in Q : (q, x, q') \in T\} \\
\mathsf{out}(Q') &= \{x \in \mathsf{init}(Q') \mid x \in \Sigma_O\} \cup \{\delta \mid \exists q \in Q' : \delta(q)\} \\
Q' \ \mathsf{after}\ \epsilon &= Q' \\
Q' \ \mathsf{after}\ \mu\sigma &= \{q' \in Q \mid \exists q'' \in Q' : (q'', \mu, q') \in T \cup \{(q'', \delta, q'') \mid \delta(q'')\}\} \ \mathsf{after}\ \sigma \\
\mathsf{traces}(q) &= \{\sigma \in (\Sigma \cup \{\delta\})^* \mid \{q\} \ \mathsf{after}\ \sigma \neq \emptyset\}
\end{aligned}$$

In our notation, we sometimes replace the initial state q_0 of an LTS $\mathcal{L}$ by the LTS itself, e.g. $\mathsf{traces}(\mathcal{L}) = \mathsf{traces}(q_0)$, and $\mathcal{L} \ \mathsf{after}\ \sigma = \{q_0\} \ \mathsf{after}\ \sigma$. For technical reasons we have to restrict to systems that have no unbounded nondeterminism, i.e., $\mid q \ \mathsf{after}\ \sigma \mid < \infty$ for all q and σ. This way, the set of reached symbolic states is finite for any trace σ.

The conformance relation **ioco** relates an LTS with an input-enabled LTS. LTS are *input-enabled* if every state has an outgoing transition for every input.

Definition 2. *Let $\mathcal{L}$ be an LTS, and $\mathcal{L}'$ an input-enabled LTS, such that $\mathcal{L}$ and $\mathcal{L}'$ have the same label sets. Then $\mathcal{L}'$* **ioco** *$\mathcal{L}$ if for all $\sigma \in \mathsf{traces}(\mathcal{L})$, we have $\mathsf{out}(\mathcal{L}' \ \mathsf{after}\ \sigma) \subseteq \mathsf{out}(\mathcal{L} \ \mathsf{after}\ \sigma)$.*

2.2 Data, Terms, and Constraints

We use basic concepts from the theory of data-type specifications; see e.g., [7]. We use the following notation: B^A is the set of all functions from A to B; $\circ$ is function composition; and $\uplus$ denotes disjoint union. Furthermore, $f : A \to B$ denotes a function $f \in B^A$. We sometimes use set builder notation as a convenient notation for functions with small domains and codomains, e.g. $\{a \mapsto b\}$ is a function with $\{a \mapsto b\} \in \{b\}^{\{a\}}$, and $\{a \mapsto b\}(a) = b$, for some elements a and b.

Syntax We assume a data signature $sig = (S,\ F)$ as given, where S is a non-empty set of *sort names* and F is a non-empty set of *function symbols*. Each function symbol consists of a name f, a list of argument sort names $\langle s_1, \ldots, s_n \rangle \in S^n$, and a result sort name $s \in S$, together written as $f :: s_1, \ldots, s_n \to s$. If $n = 0$ then f is called a *constant*.

Given a signature, we can construct terms, which may contain variables. Let $\mathfrak{X}_s$ be a set of *variables* of sort $s \in S$, and let $\mathfrak{X} = \uplus_{s \in S} \mathfrak{X}_s$ be the set of all variables. *Terms* of sort s over variables $X \subseteq \mathfrak{X}$, denoted $\mathcal{T}_s(X)$, are built from variables $x \in X$ and function symbols $f \in F$, in a sort-safe manner:

- If $x \in X$ is a variable of sort s, then x is a term of sort s;
- if $(\, f :: s_1, \ldots, s_n \to s\,) \in F$ is function symbol, and $t_1, \ldots, t_n$ are terms of sorts $s_1, \ldots, s_n$, respectively, then $f(t_1, \ldots, t_n)$ is a term of sort s.

The set of all terms over $X \subseteq \mathfrak{X}$ is $\mathcal{T}(X) = \uplus_{s \in S} \mathcal{T}_s(X)$. The set of variables actually occurring in a term $t \in \mathcal{T}_s(X)$ are called the *free variables* of t, denoted

vars(t), with *vars*$(t) \subseteq X$. A *ground term* is a term in $\mathcal{T}_s(\emptyset)$, i.e., a term without free variables. The function $\mathtt{sort}_t : \mathcal{T}(\mathfrak{X}) \to S$ gives the sort of a term; it is extended to sequences of terms as usual.

We assume that there exists a specific sort $\mathtt{Bool} \in S$, which corresponds to the usual Booleans, with the usual Boolean function symbols in F, such as *True*, *False* $:: \to \mathtt{Bool}$, $\neg :: \mathtt{Bool} \to \mathtt{Bool}$, and $\wedge, \vee :: \mathtt{Bool}, \mathtt{Bool} \to \mathtt{Bool}$. Terms of sort Bool over variables $X \subseteq \mathfrak{X}$ are denoted by $\mathcal{T}_{\mathtt{Bool}}(X)$.

A variable in a term can be substituted by another term. A *term mapping* specifies this substitution; it is a function $m : X \to \mathcal{T}(Y)$, for $X, Y \subseteq \mathfrak{X}$, which is sort-safe: $\mathtt{sort}_t(x) = \mathtt{sort}_t(m(x))$ for any $x \in X$. The set of all term mappings $m : X \to \mathcal{T}(Y)$ is denoted by $\mathcal{T}(Y)^X$. For any $X \subseteq \mathfrak{X}$, $id \in \mathcal{T}(X)^X$ is the *identity term mapping* defined as: $id(x) = x$ for all $x \in X$. Given $m \in \mathcal{T}(Y)^X$ and $t \in \mathcal{T}(Z)$, the simultaneous *substitution* of all $x \in vars(t) \cap X$ by $m(x)$ is denoted $t[m]$. So, substitution is a postfix function on terms: $[m] : \mathcal{T}(Z) \to \mathcal{T}(Z \cup Y)$.

Semantics The semantics of a data signature $sig = (S, F)$, i.e., the values in its sorts, is constituted by equivalence classes of ground terms. The value of a ground term t denoted $[\![t]\!]$, is defined by $[\![t]\!] = \{t' \mid t' \equiv t\}$. Here, we assume an equivalence on ground terms, $\equiv \subseteq \mathcal{T}(\emptyset) \times \mathcal{T}(\emptyset)$, which is sort-safe: if $t_1 \equiv t_2$ then $\mathtt{sort}_t(t_1) = \mathtt{sort}_t(t_2)$. Such an equivalence $\equiv$ could be specified as a set of equations (equational specification [7]) or as a set of rewrite rules.

The semantics of a data signature $sig = (S, F)$ is then the multi-sorted initial algebra $\mathcal{A} = (\{\mathcal{U}_s \mid s \in S\}, \{\mathrm{f}_f \mid f \in F\})$, where $\mathcal{U}_s = \{[\![t]\!] \mid t \in \mathcal{T}_s(\emptyset)\}$ is the set of values of sort s; and for each function symbol $(f :: s_1, \ldots, s_n \to s) \in F$ there is a function $\mathrm{f}_f : \mathcal{U}_{s_1} \times \ldots \times \mathcal{U}_{s_n} \to \mathcal{U}_s$ defined by $\mathrm{f}_f([\![t_1]\!], \ldots, [\![t_n]\!]) = [\![f(t_1, \ldots, t_n)]\!]$, where $t_1, \ldots, t_n$ are ground terms of sorts $s_1, \ldots, s_n$, respectively. The set of all possible values is $\mathcal{U} = \biguplus\{\mathcal{U}_s \mid s \in S\}$. Function $\mathtt{sort}_v : \mathcal{U} \to S$ gives the sort of a value; it is extended to sequences of values as usual.

A *valuation* for $X \subseteq \mathfrak{X}$ is a function assigning values to variables: $\vartheta : X \to \mathcal{U}$, which is sort-safe: $\mathtt{sort}_t(x) = \mathtt{sort}_v(\vartheta(x))$. The set of all valuations for X is denoted $\mathcal{U}^X$. The extension to evaluate terms based on a valuation ϑ is called a *term evaluation* and denoted by $\vartheta_{\mathcal{T}} : \mathcal{T}(X) \to \mathcal{U}$. It is defined as $\vartheta_{\mathcal{T}}(x) = \vartheta(x)$ and $\vartheta_{\mathcal{T}}(f(t_1, \ldots, t_n)) = \mathrm{f}_f(\vartheta_{\mathcal{T}}(t_1), \ldots, \vartheta_{\mathcal{T}}(t_n))$. For a sequence of distinct variables $\bar{x} = x_0 \ldots x_n \in X^*$ and a sequence of values $\bar{w} = w_0 \ldots w_n \in \mathcal{U}^*$, we denote with $\bar{x} \mapsto \bar{w}$ the valuation in $\mathcal{U}^{\{x_0, \ldots, x_n\}}$ defined by $(\bar{x} \mapsto \bar{w})(x_i) = w_i$ for all $0 \leq i \leq n$. The semantics of a ground term mapping $m \in \mathcal{T}(\emptyset)^X$ is the valuation $[\![m]\!]$ defined as $[\![m]\!](x) = [\![m(x)]\!]$ for all $x \in X$.

In our test algorithm, we need to represent the values in a valuation $\vartheta \in \mathcal{U}^X$ as terms again. We therefore use any term mapping $tmap(\vartheta) \in \mathcal{T}(\emptyset)^X$ satisfying $(tmap(\vartheta))(x) = t \Rightarrow \vartheta(x) = [\![t]\!]$, for all $x \in X$.

For sort Bool we assume that $[\![\ ]\!]$ interprets ground terms in $\mathcal{T}_{\mathtt{Bool}}(\emptyset)$ as usual, e.g., $[\![True]\!] = \mathtt{true}$. Boolean terms can be seen as formulas, for which we can consider their satisfiability. A Boolean term $t \in \mathcal{T}_{\mathtt{Bool}}(X)$ is *satisfiable* if there exists a valuation $\vartheta \in \mathcal{U}^{vars(t)}$ such that $\vartheta_{\mathcal{T}}(t) = \mathtt{true}$. Satisfiability, however, is undecidable in general. Hence, a tool solving satisfiability problems in our algorithms may return 'unknown'. Therefore we will distinguish explicitly

between semantic satisfiability and a tool `solver`, with `solver(`t`)` returning either `sat`, `unsat`, or `unknown`. Moreover, we assume that `solver` allows to retrieve a valuation that witnesses satisfiability, if `solver(`t`) = sat`, so that we can use these values as input values for the SUT in our testing algorithm. That is, we assume a function `getValues` that, given a term $t \in \mathcal{T}_{\texttt{Bool}}(X)$ and a sequence $\bar{p} \in vars(t)^*$, returns values $\bar{w} \in \mathcal{U}^*$, with $\texttt{sort}_v(\bar{w}) = \texttt{sort}_t(\bar{p})$, such that the valuation $\bar{p} \mapsto \bar{w}$ together with a valuation for the remaining variables in $vars(t)$, makes the Boolean term t evaluate to true, i.e., `getValues(`t`,`$\bar{p}$`)` $= \bar{w}$ implies that $\exists \vartheta \in \mathcal{U}^{vars(t)\setminus\{\bar{p}\}} : (\bar{p} \mapsto \bar{w} \uplus \vartheta)_{\mathcal{T}}(t) = \texttt{true}$.

2.3 Syntax of Symbolic Transition Systems

Definition 3. *A Symbolic Transition System (STS) with inputs and outputs is a tuple* $(\mathcal{L}, l_0, \mathcal{V}_l, m_{ini}, \mathcal{V}_p, \Gamma_I, \Gamma_O, \mathcal{R})$ *where:*

- $\mathcal{L}$ *is a finite set of* locations*,*
- $l_0 \in \mathcal{L}$ *is the* initial location*,*
- $\mathcal{V}_l$ *is a finite set of* location variables*,*
- $m_{ini} \in \mathcal{T}(\emptyset)^{\mathcal{V}_l}$ *is the* initialization*,*
- $\mathcal{V}_p$ *is a finite set of* gate parameters *such that* $\mathcal{V}_p \cap \mathcal{V}_l = \emptyset$*,*
- Γ_I *is a finite set of input* gates*,*
- Γ_O *is a finite set of output* gates*,*
- $\mathcal{R} \subseteq \mathcal{L} \times (\Gamma_I \cup \Gamma_O) \times \mathcal{V}_p^* \times \mathcal{T}_{\texttt{Bool}}(\mathcal{V}_l \cup \mathcal{V}_p) \times \mathcal{T}(\mathcal{V}_l \cup \mathcal{V}_p)^{\mathcal{V}_l} \times \mathcal{L}$ *is the* switch relation *with a finite number of elements.*

We require that $\Gamma_I \cap \Gamma_O = \emptyset$*, and denote* $\Gamma = \Gamma_I \cup \Gamma_O$*. The function* $\texttt{sort}_g : \Gamma \to S^*$*, associates a sequence of sorts to a gate. We refer to the elements of a switch* $(l_1, \lambda, p_0 \dots p_k, \phi, \psi, l_2) \in \mathcal{R}$*, with source location, gate, parameters, guard, assignment, and destination location, respectively, and we require that:*

- $p_0 \dots p_k$ *is a sequence of distinct variables*
- $\texttt{sort}_g(\lambda) = \texttt{sort}_t(p_0 \dots p_k)$
- $\phi \in \mathcal{T}_{\texttt{Bool}}(\mathcal{V}_l \cup \{p_0, \dots, p_k\})$
- $\psi \in \mathcal{T}(\mathcal{V}_l \cup \{p_0, \dots, p_k\})^{\mathcal{V}_l}$

Example 4. Figure 1 shows an example STS in graphical representation. The formal definition of this STS is: $(\{l_0, l_1, l_2\}, l_0, \{x\}, \{x \mapsto 0\}, \{p\}, \{inX?\}, \{outX!, done!\}, \{r_0, r_1, r_2\})$*. We have* $\texttt{sort}_g(inX?) = \texttt{sort}_g(outX!) = \texttt{Int}$*, and* $\texttt{sort}_g(done!) = \epsilon$*. We have switches:*
$r_0 = (l_0, inX?, p, 1 \le p \le 10, \{x \mapsto x + p\}, l_1)$*,*
$r_1 = (l_1, outX!, p, p = x, id, l_0)$*, and*
$r_2 = (l_0, done!, \epsilon, x > 15, id, l_2)$*.* □

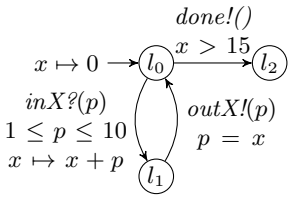

Fig. 1. *Example STS*

2.4 Semantics of Symbolic Transition Systems

We define the semantics of an STS in terms of an LTS. We call this LTS the *interpretation*. We first define the states and labels of this LTS.

Definition 5. *The domain of* semantic states *of* $\mathcal{S}$ *is* $SemState = \mathcal{L} \times \mathcal{U}^{\mathcal{V}_l}$.

Definition 6. *A* gate value *is an element* $(\lambda, \bar{w}) \in \Gamma \times \mathcal{U}^*$ *such that* $\mathtt{sort}_g(\lambda) = \mathtt{sort}_v(\bar{w})$. *We denote the set of all gates values with* Γ_{val}. *We define* $\Gamma^O_{val} = \{(\lambda, \bar{w}) \in \Gamma_{val} \mid \lambda \in \Gamma_O\}$, *and* $\Gamma^I_{val} = \{(\lambda, \bar{w}) \in \Gamma_{val} \mid \lambda \in \Gamma_I\}$.

For a given semantic state and gate value, there will be a transition in the interpretation LTS, depending on the guard of a switch. If so, we use the assignment from the switch to compute the destination state of this transition.

Definition 7. *Let* $q = (l, \vartheta) \in SemState$ *be a semantic state,* $u = (\lambda_1, \bar{w}) \in \Gamma_{val}$ *a gate value, and* $r = (l_1, \lambda_2, \bar{p}, \phi, \psi, l_2) \in \mathcal{R}$ *a switch. Then* r *is* enabled *in* q *for* u, *denoted* $enab(q, u, r)$, *if:*

$$l = l_1 \wedge \lambda_1 = \lambda_2 \wedge ((\bar{p} \mapsto \bar{w}) \uplus \vartheta)_{\mathcal{T}}(\phi) = \mathtt{true}$$

If $enab(q, u, r)$, *the* successor *of* q *and* u *for* r, *denoted* $succ(q, u, r)$, *is the semantic state:*

$$(l_2, ((\bar{p} \mapsto \bar{w}) \uplus \vartheta)_{\mathcal{T}} \circ \psi)$$

Definition 8. *The* interpretation *of* $\mathcal{S}$ *is the LTS* $[\![\mathcal{S}]\!] = (SemState, (l_0, [\![m_{ini}]\!]), \Gamma^I_{val}, \Gamma^O_{val}, T_c)$ *with* $T_c \subseteq SemState \times \Gamma_{val} \times SemState$, *such that:*

$$T_c = \{(q, u, succ(q, u, r)) \mid q \in SemState \wedge u \in \Gamma_{val} \wedge r \in \mathcal{R} \wedge enab(q, u, r)\}$$

Example 9. Consider the STS of Example 4. Switch r_0 is enabled in initial semantic state $q_0 = (l_0, \{x \mapsto 0\})$ for gate value (inX?,3): $(\{x \mapsto 0\} \uplus \{p \mapsto 3\})_{\mathcal{T}}(1 \leq p \leq 10) = 1 \ \mathrm{f}_{\leq}\ 3 \ \mathrm{f}_{\leq}\ 10 = \mathtt{true}$. Then $succ(q_0, (\text{inX?}, 3), r_0) = (l_1, \vartheta)$ with $\vartheta = (\{x \mapsto 0\} \uplus \{p \mapsto 3\})_{\mathcal{T}} \circ \{x \mapsto x + p\} = \{x \mapsto 0 \ \mathrm{f}_{+}\ 3\} = \{x \mapsto 3\}$. Consequently, $((l_0, \{x \mapsto 0\}), (\text{inX?}, 3), (l_1, \{x \mapsto 3\}))$ is a transition in the interpretation. Furthermore, switch r_2 is not enabled in q_0 for gate value (done!,ϵ): $(\{x \mapsto 0\} \uplus \emptyset)_{\mathcal{T}}(x > 15) = 0 \ \mathrm{f}_{>}\ 15 = \mathtt{false}$. □

We note that the interpretation of an STS may have an infinite number of states and transitions. Additionally, it hides all information about the structure of an STS, which actually could be used in test generation. We improve on this in the next section, by applying symbolic execution on STS.

3 Symbolic Execution Graphs

This section covers the symbolic execution elements of an STS, resulting in a symbolic execution graph, which is an LTS, having symbolic states, and switches as transitions. Analogous to semantic states, *symbolic states* keep track of the location, and a mapping of location variables to *symbolic values*. The symbolic values of location variables are (syntactic) terms over the parameters, encountered on the switches, instead of (semantical) values from $\mathcal{U}$. The third element of a symbolic state is the *path condition*: a term of sort `Bool`, constraining the

possible values of the encountered parameters from the traversed switches, and constructed as a conjunction over the encountered guards of switches.

Since switches with the same gate will use the same gate parameters, and since a single switch can be traversed multiple times, the same parameter may be encountered several times, but it can be bound to different values at each of these points. For our symbolic states to be well-defined, we need to distinguish them. Therefore, we will identify distinct occurrences of parameters by a syntactical mechanism: adding prime ($'$) symbols to parameter names, each time a switch is taken. This is achieved by simultaneously substituting all variables v by v', so also when v already has some primes. We use the notation $\mathcal{V}'$ to denote the set consisting of variables $\mathcal{V}$ and any of their (multiply) primed variants.

Definition 10. *Let $\mathcal{V}_1, \mathcal{V}_2$ be sets of variables, let $t \in \mathcal{T}(\mathcal{V}_1')$ be a term, and $m \in \mathcal{T}(\mathcal{V}_1')^{\mathcal{V}_2}$ a term mapping. Then we define:*

$$t' = t[\{v \mapsto v' \mid v \in \mathcal{V}_1'\}]$$
$$m' = \{v \mapsto (m(v))' \mid v \in \mathcal{V}_2\}$$

Definition 11. *The domain of* symbolic states *of $\mathcal{S}$ is denoted SymState, where $SymState = \mathcal{L} \times \mathcal{T}(\mathcal{V}_p')^{\mathcal{V}_l} \times \mathcal{T}_{\texttt{Bool}}(\mathcal{V}_p')$.*

A symbolic state has a transition for a switch, if `solver` returns `sat` or `unknown` for the conjunction of the guard of the switch and the path condition of the symbolic state. We use the term mapping of the state to substitute the location variables of the guard with terms over parameters, and use priming to prevent variables of the guard to become indistinguishable from the parameters of the previously taken switches. To obtain a successor state for an enabled switch, we use the assignment of the switch to update the term mapping, and conjunct the checked guard to the path condition of the current state.

Definition 12. *Let $s = (l, m, \eta) \in SymState$, and $r = (l_1, \lambda, \bar{p}, \phi, \psi, l_2) \in \mathcal{R}$. Then r is* enabled *in s, denoted $enab(s, r)$, if:*

$$l = l_1 \wedge \texttt{solver}(\phi[m'] \wedge \eta') \in \{\texttt{sat}, \texttt{unknown}\}$$

If $enab(s, r)$, the successor *of s for r, denoted $succ(s, r)$, is the symbolic state:*

$$(l_2, [m'] \circ \psi, \phi[m'] \wedge \eta')$$

Example 13. Consider the STS of Example 4. Suppose we check which switches are enabled in symbolic state $q_0 = (l_0, x \mapsto 0, \mathit{True})$. As $l_1 \neq l_0$, r_1 is not enabled in q_0. Switches r_0 and r_2 do have source location l_0, so we check satisfiablity. For r_0 we obtain: $(1 \leq p \leq 10[(x \mapsto 0)']) \wedge (\mathit{True})' = 1 \leq p \leq 10 \wedge \mathit{True}$, which is clearly satisfiable. Then we obtain the successor: $(l_1, [\{x \mapsto 0\}'] \circ \{x \mapsto x+p\}, 1 \leq p \leq 10[\{x \mapsto 0\}'] \wedge (\mathit{True})') = (l_1, \{x \mapsto 0 + p\}, 1 \leq p \leq 10 \wedge \mathit{True})$. Switch r_2 is not enabled in q_0: $x > 15[(x \mapsto 0)'] \wedge (\mathit{True})' = 0 > 15 \wedge \mathit{True}$. □

We now define a *symbolic execution graph* as an LTS with symbolic states, and transitions labeled by switches. A trace in the graph means that values could be chosen for the parameters, for the switches to be enabled subsequently, or that the trace received the benifit of the doubt, because `solver` returned `unknown`.

Definition 14. *The* symbolic execution graph *(SEG) of $\mathcal{S}$ is an LTS $seg(\mathcal{S}) = (SymState, (l_0, m_{ini}, True), \Sigma_I, \Sigma_O, T_s)$ with $\Sigma_I = \{(l_1, \lambda, \bar{p}, \phi, \psi, l_2) \in \mathcal{R} \mid \lambda \in \Gamma_I\}$, $\Sigma_O = \{(l_1, \lambda, \bar{p}, \phi, \psi, l_2) \in \mathcal{R} \mid \lambda \in \Gamma_O\}$, $T_s \subseteq SymState \times \mathcal{R} \times SymState$, where:*

$$T_s = \{(s, r, succ(s, r)) \mid s \in SymState \wedge r \in \mathcal{R} \wedge enab(s, r)\}$$

Example 15. In Figure 2 we compute a part of the SEG for the STS of Example 4. We note that states with an unsatisfiable path condition may actually appear in the SEG, if the chosen `solver` cannot detect its unsatisfiability. □

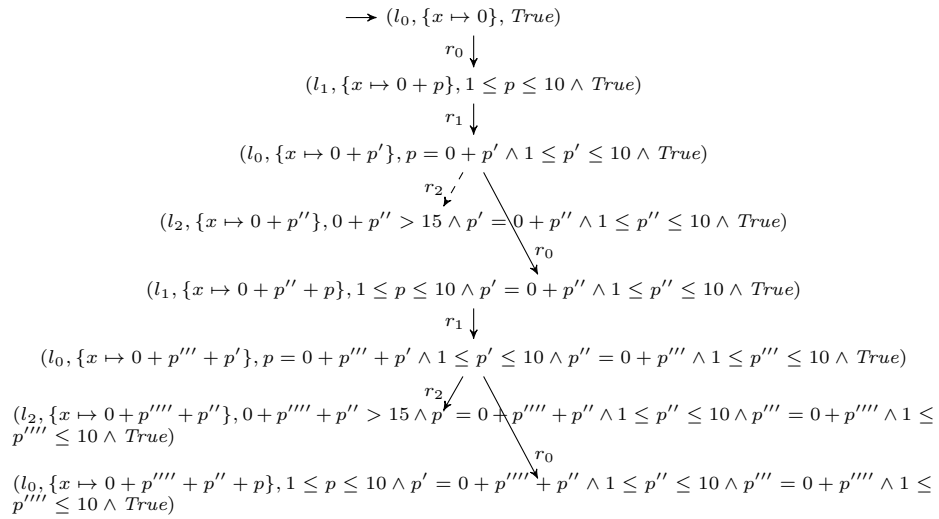

Fig. 2. A part of the symbolic execution graph for the STS of Example 4. For the path condition of the state reached by the dashed edge, `solver` returned `unknown`.

4 Test Purposes with Switch Coverage

We use traces of the SEG as test purposes for test execution. The path condition of the state reached by the trace describes possible values of parameters, while the trace itself contains all the gates to be encountered.

Definition 16. *An element $(r_0 \dots r_n, \eta) \in \mathsf{traces}(seg(\mathcal{S})) \times \mathcal{T}_{\mathtt{Bool}}(\mathcal{V}'_p)$ is a* test purpose *for $\mathcal{S}$ if:*

$$\exists l \in \mathcal{L}, \exists m \in \mathcal{T}(\mathcal{V}'_p)^{\mathcal{V}_l} : (l_0, m_{ini}, True) \,\mathsf{after}\, r_0 \dots r_n = \{(l, m, \eta)\}$$

For a set of test purposes TP we define:

$$cov(TP) = |\bigcup \{\{r_0, \dots, r_n\} \mid (r_0 \dots r_n, \eta) \in TP\}| / |\mathcal{R}|$$

Given a set TP for $\mathcal{S}$, we call $cov(TP)$ the a priori switch coverage *of TP.*

Example 17. From the SEG of Example 15 we obtain the following test purpose with 100% a priori switch coverage: $(r_0 r_1 r_0 r_1 r_2, 0 + p'''' + p'' > 15 \wedge p' = 0 + p'''' + p'' \wedge 1 \leq p'' \leq 10 \wedge p''' = 0 + p'''' \wedge 1 \leq p'''' \leq 10 \wedge \mathit{True})$. The test purpose for path $r_0 r_1 r_2$ also gives 100% a priori switch coverage, but in Example 18 we show it is not executable, and therefore has 0% a posteriori switch coverage. □

For a test purpose $(r_0 \dots r_n, \eta)$ we call $r_0 \dots r_n$ the *path*, and η the *path condition*. If all switches occur in some finite trace of the SEG, a set of purposes with 100% a priori coverage can be found through breadth first search. In Algorithm 1, we give pseudo code for the test execution of a test purpose, including on-the-fly data generation for parameters. We explain the intuition of it below.

Execution of a test purpose may result in either of three verdicts *Pass*, *Inconclusive*, or *Fail*. Verdict *Fail* means that a non-conformance (with respect to **ioco**) to the given STS specification was found. Verdict *Pass* means that execution was completed, without encountering any non-conformance. Verdict *Inconclusive* means that execution of the test purpose could not be completed, but no non-conformance was detected either.

A test purpose is executed by executing its switches of the path in the given order. A gate value is provided to the SUT, or received from the SUT, if the switch being executed has an input gate, or output gate, respectively. Every time a new gate value is obtained, we update the path condition, by substituting the values from the gate value for the parameters of the switch. If the path condition becomes unsatisfiable by this substitution, we then can immediately return *Inconclusive*, as there is no way the test purpose can be executed completely. Also, we use this path condition to obtain suitable values for parameters of a switch with an input gate. The substitution ensures that all previously observed gate values are taken into account. If `solver` returns `unknown`, we resort to using only the guard of the current switch to find values.

The path condition of a test purpose contains (multiply) primed variables, while the parameters of switches are not primed, so for obtaining the parameters of the path condition (e.g., line 8 of Algorithm 1), we use a notation for adding primes to parameters. We denote adding k primes to a parameter p with $(p)^{\prime \cdot k}$, so, e.g., $(p)^{\prime \cdot 0} = p$, and $(p)^{\prime \cdot 3} = p'''$. We extend this notation to sequences.

During execution of the test purpose, we keep track of state (l, ϑ), which is the current semantic state for the observed gate values. We use valuation ϑ to obtain parameter values for the guard of the current switch, if `solver` returns `unknown` for the path condition in line 12 of Algorithm 1.

Additionally, we use (l, ϑ) to check whether it enables the output gate value, to check whether we are still on track for reaching the test purpose (line 28). If this is not the case, we need to check if the specification allows a different sequence of switches for the observed gate values (line 36) to see whether the SUT only deviated from the path of the test purpose, or that it is really non-conforming. We therefore keep track of all semantic states that can be obtained for the observed gate values in C. To do this as efficiently as possible, we use the information available in the SEG. As the graph can be infinite, we assume that, if Algorithm 1 is given a test purpose $(r_0, \dots, r_n, \eta)$ to be executed, it is

given a partially computed SEG which contains all traces of length $n+1$. This is no strong requirement, as to find the test purpose via breadth first search, we already computed (almost) all of these traces. Moreover, computing all traces can be done *before* test execution of the test purposes, and will compensate for computation time needed *during* test execution.

Specifically, the algorithm keeps track of a set C containing pairs $(q, s) \in \mathit{SemState} \times \mathit{SymState}$, representing the current semantic state with the corresponding symbolic state, for a sequence of switches, consistent with the gate values observed so far. The set of switches enabled in q is at most $\mathsf{init}(s)$, as s describes a set of semantic states, of which q is one. We then use q to select the switches from $\mathsf{init}(s)$ actually enabled in q, and compute its successor with the last obtained gate value. The successor of s for these switches can be obtained from the SEG without any computation. If an output gate value, not enabled in (l, ϑ), is received from the SUT, we can check, as decribed above, whether this gate value is enabled (line 36). If so, the SUT is conforming but deviated from the test purpose, and otherwise it is non-conforming.

For communication with the SUT the algorithm uses two procedures: `sendInput` provides an input to the SUT, and `receiveOutput` obtains an enabled output from the SUT.

Example 18. We execute the first test purpose of Example 17 according to Algorithm 1. Hence, we set $(l, \vartheta) := (l_0, \{x \mapsto 0\})$, and $C := \{((l_0, \{x \mapsto 0\}), (l_0, \{x \mapsto 0\}, \mathit{True}))\}$. We discuss the first 2 of 5 iterations of the for-loop:

Iteration 0: As r_0 has input gate inX?, we execute lines 6-22 of the algorithm. Suppose that `solver(`η`) = sat`, and that we obtain value 6 for parameter $p'^{\cdot(4-0)} = p''''$ on line 7. Then we send (inX?,6) to the SUT. We substitute 6 for p'''' in the path condition: $\eta := 0 + 6 + p'' > 15 \wedge p' = 0 + 6 + p'' \wedge 1 \leq p'' \leq 10 \wedge p''' = 0 + 6 \wedge 1 \leq 6 \leq 10$. We obtain $(l, \vartheta) := (l_1, \{x \mapsto 6\})$, and $C := \{(l, \vartheta), (l_1, \{x \mapsto 0 + p\}, 1 \leq p \leq 10 \wedge \mathit{True})\}$.

Iteration 1: Switch r_1 has output gate outX!, so we execute lines 24-39 of the algorithm. We then receive some output from the SUT. We discuss two cases:

1. Suppose that we receive (outX!,6). We then observe that $(\text{outX!}, 6) \in \Gamma^O_{val}$. We see that $\mathit{enab}((l_1, \{x \mapsto 6\}), (\text{outX!}, 6), r_1)$ holds. We substitute 6 for $p'^{\cdot(4-1)} = p''' : \eta := 0 + 6 + p'' > 15 \wedge p' = 0 + 6 + p'' \wedge 1 \leq p'' \leq 10 \wedge 6 = 0+6 \wedge 1 \leq 6 \leq 10$. Now η is satisfiable (choose valuation $\{p'' \mapsto 10, p' \mapsto 10\}$). We obtain $(l, \vartheta) := (l_0, \{x \mapsto 6\})$, and $C = \{((l, \vartheta), (l_0, \{x \mapsto 0 + p'\}, p = 0 + p' \wedge 1 \leq p' \leq 10 \wedge \mathit{True}))\}$, and go to the next iteration.
2. Now suppose that we receive (outX!,7). We observe that $(\text{outX!}, 7) \in \Gamma^O_{val}$. As $6 \neq 7$, $\mathit{enab}((l_1, \{x \mapsto 6\}), (\text{outX!}, 7), r_1)$ does not hold. Since no other switch than r_1 is enabled in $(l_1, \{x \mapsto 0+p\}, 1 \leq p \leq 10 \wedge \mathit{True})$, according to the SEG, we find that the condition of line 36 is false, so we return *Fail*. □

For the second test purpose of Example 17 we assumed that `solver` could not detect that the path condition is not satisfiable, so `solver`(η) = `unknown`. This leads us to line 12 in Algorithm 1. The second condition in line 12 can be solved, e.g., with $p = 8$, so then according to line 14, $\eta := 0 + 8 > 15 \wedge p' = 0 + 8 \wedge 1 \leq 8 \leq 10 \wedge \mathit{True}$. Most likely, now any `solver` will detect that this is not satisfiable,

Input: A specification STS $\mathcal{S}$
Input: A test purpose $((r_0 \dots r_n), \eta)$
Input: A symbolic execution graph $seg(\mathcal{S})$
Input: An SUT with procedures `sendInput` and `receiveOutput`
Output: One of the verdicts: *Pass*, *Fail*, *Inconclusive*

$(l, \vartheta) := (l_0, [\![m_{ini}]\!])$;
$C := \{((l_0, [\![m_{ini}]\!]), (l_0, m_{ini}, \mathit{True}))\}$;
for $0 \leq i \leq n$ **do**
  Let $(l_1, \lambda, \bar{p}, \phi, \psi, l_2) = r_i$;
  if $\lambda \in \Gamma_I$ **then**
    if `solver(`η`) = sat` **then**
      $\bar{w}$:= `getValues(`$\eta, (\bar{p})^{\prime \cdot (n-i)}$`)`;
      $\eta := \eta[tmap((\bar{p})^{\prime \cdot (n-i)} \mapsto \bar{w})]$;
      `sendInput(`$\lambda, \bar{w}$`)`;
      $(l, \vartheta) := succ((l, \vartheta), (\lambda, \bar{w}), r_i)$;
      $C := \{(succ(q, (\lambda, \bar{w}), r), succ(s, r)) \mid (q, s) \in C, r \in \mathsf{init}(s), enab(q, (\lambda, \bar{w}), r)\}$;
    else if `solver(`η`) = unknown` $\wedge$ `solver(`$\phi[tmap(\vartheta)]$`) = sat` **then**
      $\bar{w}$:= `getValues(`$\phi[tmap(\vartheta)], \bar{p}$`)`;
      $\eta := \eta[tmap((\bar{p})^{\prime \cdot (n-i)} \mapsto \bar{w})]$;
      if `solver(`η`) = unsat` **then**
        return *Inconclusive*;
      else
        `sendInput(`$\lambda, \bar{w}$`)`;
        $(l, \vartheta) := succ((l, \vartheta), (\lambda, \bar{w}), r_i)$;
        $C := \{(succ(q, (\lambda, \bar{w}), r), succ(s, r)) \mid (q, s) \in C, r \in \mathsf{init}(s), enab(q, (\lambda, \bar{w}), r)\}$;
    else
      return *Inconclusive*;
  else
    $(\lambda_{sut}, \bar{w})$ = `receiveOutput()`;
    if $(\lambda_{sut}, \bar{w}) \notin \Gamma^O_{val}$ **then**
      return *Fail*;
    else
      if $enab((l, \vartheta), (\lambda_{sut}, \bar{w}), r_i)$ **then**
        $\eta := \eta[tmap((\bar{p})^{\prime \cdot (n-i)} \mapsto \bar{w})]$;
        if `solver(`η`) = unsat` **then**
          return *Inconclusive*;
        else
          $(l, \vartheta) := succ((l, \vartheta), (\lambda_{sut}, \bar{w}), r_i)$;
          $C := \{(succ(q, (\lambda, \bar{w}), r), succ(s, r)) \mid (q, s) \in C, r \in \mathsf{init}(s), enab(q, (\lambda, \bar{w}), r)\}$;
      else
        if $\exists (q, s) \in C, \exists r \in \mathsf{init}(s) : enab(q, (\lambda_{sut}, \bar{w}), r)$ **then**
          return *Inconclusive*;
        else
          return *Fail*;
return *Pass*;

Algorithm 1: Test generation and execution algorithm for test purposes

$\texttt{solver}(\eta) = \texttt{unsat}$, so the algorithm ends in line 16 with verdict *Inconclusive*. The a posteriori coverage is 0%, decreasing from 100% a priori coverage.

Theorem 19 proves soundness of Algorithm 1. For the proofs we refer to [2].

Theorem 19. *Let $t = ((r_0 \dots r_n), \eta)$ be a test purpose for $\mathcal{S}$, and let $\mathcal{I}$ be an input-enabled LTS, such that $\mathcal{I}$* **ioco** *$[\![\mathcal{S}]\!]$. Then Algorithm 1 does not return Fail for $\mathcal{S}$, t, and seg($\mathcal{S}$), when using $\mathcal{I}$ as SUT, and some tool* `solver`.

The *a posteriori switch coverage* can be determined after all test purposes have been executed. We define it as *cov(TP')*, where *TP'* is the set of test purposes for which Algorithm 1 returned verdict *Pass*, with $|C| = 1$ at that point (line 40). This is a conservative approach: a test purpose $(\bar{r}, \eta)$ only counts if we are sure that its execution ends in $(l_0, m_{ini}, \mathit{True})$ $\mathsf{after}\ \bar{r}$. A more liberal approach would count any test purpose with verdict *Pass*, without requiring $|C| = 1$, meaning that nondeterministically another final state than $(l_0, m_{ini}, \mathit{True})$ $\mathsf{after}\ \bar{r}$ might have been reached.

5 Implementation of the test approach in Maude and Z3

We implemented our testing method with Maude: a language and tool set for rewriting systems. We encode each switch of an STS as a conditional rewriting rule. Such a rule rewrites a symbolic state to its successor state. A rewrite rule is conditional: it can only be applied if the guard, of the switch it encodes, holds. Maude queries SMT-solver Z3 to check $\texttt{solver}(\phi[m'] \wedge \eta') \in \{\texttt{sat}, \texttt{unknown}\}$, as in Definition 12. To do this, we encoded the used Maude data types in the SMT-LIB language, which is an input language for Z3. The 'meta-level' feature of Maude supports this translation, by enabling syntactic inspection of terms. As our case study only involved integers and booleans, we only constructed translation bindings for these data types. The Maude language, however, can be used to define any data type, so one could make these bindings for any data type that Z3 (or any other SMT solver) supports.

We used Maude's search query to search in the state space of the states, which can be obtained by applying rewriting rules. We use this to find sequences of switches ending on a certain switch. This way, Maude searches in the symbolic execution graph for test purposes contributing to a priori switch coverage.

We wrote a Python program to execute the test purposes, following Algorithm 1. The program queries Z3 again, to find suitable values for input parameters, as outlined in the algorithm.

6 Case study: the Bounded Retransmission Protocol

To evaluate our approach, we used the Bounded Retransmission Protocol [13] benchmark from the Automata Wiki [1]. This is a repository containing automata in various automata formalisms, which are suitable for benchmarking.

For the Bounded Retransmission Protocol the Wiki provides a specification automaton, and 6 mutants. See Figure 3 for the specification, and the Wiki for all the mutants. Each mutant differs on one aspect from the specification, e.g. a different guard, an extra switch, or an extra assignment.

The Bounded Retransmission Protocol is a variation on the alternating bit protocol. After sending a message, the sender waits for an acknowledgment from the receiving party. If an acknowledgment fails to appear, the sender times out, and sends the message again. This retransmission is executed at most `max_rn` times, for some number `max_rn`.

	Switch coverage	TorXakis
Mutant 1	44	12
Mutant 2	16	234
Mutant 3	8	12
Mutant 4	6	18
Mutant 5	18	1620
Mutant 6	164	76
Sum	256	1972
Geom. mean	21.5	64.9

Table 1. Number of inputs and outputs needed for detecting the BRP mutants

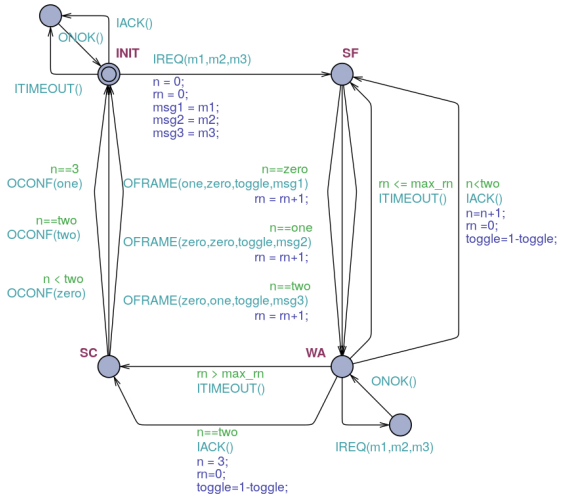

Fig. 3. Specification of BRP

We generated a set of test purposes with 100% switch coverage from the specification, and applied it on the mutants. We then measured the number of inputs and outputs needed until obtaining a *Fail* verdict. To obtain *Fail* for mutant 6, just executing all the switch coverage test purposes was not sufficient. This was caused by the SMT-solver we used (Z3): it always returns the same solution for the same query, and the particular solution did not result in obtaining a *Fail*. Retrieving random solutions from an SMT-solver is out of scope of this paper, but it has been investigated in e.g. [15,16,19], and some techniques are implemented in the tool TorXakis [25]. We chose to solve the problem by randomly generating values for input switch parameters, and then checking the satisfiability of the constraint for these values. As none of the parameters of input switches of the BRP specification were constrained by any of the test purposes, this worked well enough for this case study. We averaged the number of inputs and outputs over 100 executions.

The test purposes were executed in the order of their number of inputs and outputs (longest test purpose first). We applied this heuristic with the idea that the longest test purpose is likely to be better at discovering more difficult faults,

which are hidden further away, and at the same time covers a lot of the switches of the specification.

We compared our results with random testing (on gate actions and data values) of STS as implemented in the tool TorXakis [22,25]. Again, we measured the number of inputs and outputs until encountering a *Fail*. We report on the average number over 100 executions for each mutant.

Our test generation produced 6 test purposes to obtain 100% a priori switch coverage. They in total consist of 47 inputs and outputs. As the guards of the nondeterministic switches, i.e. switches having the same gate and source location, are disjunct, we could determine the exact a posteriori switch coverage, which was 100% as well. Table 1 shows the results for testing with our test purposes, and random testing with TorXakis.

The faults in mutants 2,3,4, and 5 are detected by the test purpose with the longest path of 18 inputs and outputs. Note that the difference in number of inputs and outputs for mutant 5 is especially big. The fault of mutant 1 is detected by one shorter test purpose, in just 4 inputs and outputs. In total we needed more inputs and outputs because of our heuristic to do the longest test purpose first. Mutant 6 was detected by the longest and the third longest test purpose, if the right values are chosen for the gate parameters. All test purposes are executed before these two test purposes are tried again with (possibly) different values. This causes TorXakis to be faster in detecting the fault than us.

Overall, our approach is better than testing with TorXakis: the total number of inputs and outputs needed to detect the faults of all mutants is 8 times more for TorXakis than for our approach. One can see however, that random testing can be quite effective for faults detectable with few inputs and outputs (mutant 1), and that the thoroughness of covering all switches can sometimes be a bit less effective than random testing (mutant 6). Our approach is significantly better in reaching a switch which is not so likely to be reached by using random test generation (mutant 5), and the number of inputs and outputs to obtain a *Fail* is much more stable than for random testing. Despite the varying results for each mutant, the geometric mean shows that our approach is overall 3 times better than TorXakis.

7 Conclusions and Future Work

We proposed a test generation and execution method for Symbolic Transition Systems. We extend on the work in [8,9], and provide sound test selection based on switch coverage. We select data values as late as possible during test execution, to provide optimal flexibility with respect to received outputs from the SUT. Furthermore, our test generation and execution explicitly deals with solvers returning 'unknown'. The BRP case study shows applicability of our approach.

There are ample opportunities to extend upon this paper in future work:

- The most important extension is adding quiescence, which is a key concept in the **ioco** theory, but which we did not take into account yet. Actually, our test algorithm tests for a weaker form of **ioco** where quiescence is not

considered, i.e., removing quiescence from **ioco** in Definition 2 and only requiring inclusion of outputs. Since this weaker form of **ioco** is implied by the original **ioco** of Definition 2, soundness in Theorem 19 is not affected. But it may occur that our algorithm deadlocks when the SUT is quiescent and the test purpose expects an output (line 24). The reason that we did not add quiescence yet, is that it leads to quantified guards in switches: a state is quiescent if there does not exist an output parameter value that makes the guard of an output switch true. It is not yet clear how and where to add these quantified guards. Quiescence can be added as just another output label in the STS. Another option is to add quiescence when constructing the symbolic execution graph, so that quiescence can also be part of test purposes. Yet another option is to take quiescence only into account in the testing algorithm. Whatever option, it will introduce quantified guards, which complicates the formalism and the satisfiability checking.
- By executing test purposes simultaneously, instead of one by one, test execution can proceed until all test purposes received a verdict. This could improve test efficiency. Also, test purposes returning *Inconclusive* could be re-executed, to obtain better a posteriori switch coverage.
- The test algorithm is written down as if we assume *input-eager* interaction with the SUT: the SUT always accepts an input, even though it could have produced an output. This could be relaxed to *input-fair* interaction [3,4].
- The symbolic states of the symbolic execution graph are defined as syntactic objects. When adding semantics, an equivalence could be defined e.g. as in Maude [5], so that the size of the symbolic execution graph could be reduced.
- We have existential quanfication for both inputs and outputs in path conditions, whereas universal quantification might be more natural for outputs. This may, however, lead to more unsatisfiable path conditions and moreover, current solvers usually perform worse on formulas with alternating quantification.
- Our switch coverage approach could be combined with traditional data coverage techniques e.g. equivalence partitioning, boundary value analysis, or the techniques for random value selection used in TorXakis. Witnesses of solvers usually do not produce much data coverage, as mutant 6 of the BRP benchmark showed.
- To check scalability, our switch coverage testing approach should be evaluated more extensively, by performing experiments on additional case studies.

Acknowledgments This work was largely performed while the first author was visiting INRIA/the University of Lille. We would like to thank Vlad Rusu for making this visit possible, and for his feedback and support with using Maude.

References

1. Automata Wiki. `automata.cs.ru.nl`.
2. Homepage Petra van den Bos. `petravdbos.nl`.

3. P. van den Bos, R. Janssen, and J. Moerman. n-Complete test suites for IOCO. *Software Quality Journal*, 27(2):563–588, Jun 2019.
4. P. van den Bos and M. Stoelinga. Tester versus bug: A generic framework for model-based testing via games. In Andrea Orlandini and Martin Zimmermann, editors, Proceedings Ninth International Symposium on *Games, Automata, Logics, and Formal Verification,* Saarbrücken, Germany, 26-28th September 2018, volume 277 of *Electronic Proceedings in Theoretical Computer Science*, pages 118–132. Open Publishing Association, 2018.
5. M. Clavel, F. Durn, S. Eker, P. Lincoln, N. Mart-Oliet, J. Meseguer, and J.F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187 – 243, 2002. Rewriting Logic and its Applications.
6. L. de Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
7. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I – Equations and Initial Semantics.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1985.
8. L. Frantzen, J. Tretmans, and T. Willemse. Test Generation Based on Symbolic Specifications. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Software Testing – FATES 2004*, volume 3395 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2005.
9. L. Frantzen, J. Tretmans, and T. Willemse. A Symbolic Framework for Model-Based Testing. In K. Havelund, M. Núñez, G. Roşu, and B. Wolff, editors, *Formal Approaches to Software Testing and Runtime Verification – FATES/RV'06*, volume 4262 of *Lecture Notes in Computer Science*, pages 40–54. Springer-Verlag, 2006.
10. G. Friedman, A. Hartman, K. Nagin, and T. Shiran. Projected state machine coverage for software testing. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 134–143, New York, NY, USA, 2002. ACM.
11. C. Gaston, P. Le Gall, N. Rapin, and A. Touil. Symbolic execution techniques for test purpose definition. In M. Ümit Uyar, Ali Y. Duale, and Mariusz A. Fecko, editors, *Testing of Communicating Systems*, pages 1–18, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
12. P. Godefroid, M.Y. Levin, and D.A. Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Communications of the ACM*, 55(3):40–44, 2012.
13. L. Helmink, M.A.P. Sellink, and F.W. Vaandrager. Proof-checking a data link protocol. In *International Workshop on Types for Proofs and Programs*, pages 127–165. Springer, 1993.
14. W. Huang and Jan Peleska. Complete model-based equivalence class testing for nondeterministic systems. *Formal Aspects of Computing*, 29(2):335–364, Mar 2017.
15. M. Huijben. Efficient Constrained Random Sampling for use in a Model Based Testing Tool. Master's thesis, Institute for Computing and Information Sciences, Radboud University, Nijmegen, The Netherlands, 2019.
16. M. A. Iyer. Race: A word-level atpg-based constraints solver system for smart random simulation. page 299. Citeseer, 2003.
17. B. Jeannet, T. Jéron, V. Rusu, and E. Zinovieva. Symbolic test selection based on approximate analysis. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 349–364, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
18. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

19. N. Kitchen. *Markov Chain Monte Carlo Stimulus Generation for Constrained Random Simulation*. PhD thesis, UC Berkeley, 2010.
20. J. J. Li and W. E. Wong. Automatic test generation from communicating extended finite state machine (cefsm)-based models. In *Proceedings Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. ISIRC 2002*, pages 181–185, April 2002.
21. A. Petrenko. Checking experiments for symbolic input/output finite state machines. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 229–237, April 2016.
22. TorXakis. `https://github.com/torxakis`.
23. J. Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.
24. J. Tretmans. Model Based Testing with Labelled Transition Systems. In R.M. Hierons, J.P. Bowen, and M. Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer-Verlag, 2008.
25. J. Tretmans. On the Existence of Practical Testers. In J.-P. Katoen, R. Langerak, and A. Rensink, editors, *ModelEd, TestEd, TrustEd – Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, volume 10500 of *Lecture Notes in Computer Science*, pages 87–106. Springer Int. Publishing, 2017.
26. M. Veanes and N. Bjørner. Alternating simulation and ioco. In Alexandre Petrenko, Adenilso Simão, and José Carlos Maldonado, editors, *Testing Software and Systems*, pages 47–62, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.