When BDDs Fail: Conformance Testing with Symbolic Execution and SMT Solving

Elisabeth Jöbstl, Martin Weiglhofer, Bernhard K. Aichernig, and Franz Wotawa
Institute for Software Technology
Graz University of Technology, Austria
Email: {joebstl,weiglhofer,aichernig,wotawa}@ist.tugraz.at

Abstract—Model-based testing is a well known technique that allows one to validate the correctness of software with respect to its model. If a lot of data is involved, symbolic techniques usually outperform explicit data enumeration. In this paper, we focus on a new symbolic test case generation technique. Our approach is based on symbolic execution and on satisfiability (modulo theory; SMT) solving. Our work was motivated by the complete failure of a well-known existing symbolic test case generator to produce any test cases for an industrial Session Initiation Protocol (SIP) implementation. Hence, we have replaced the BDD-based analysis of the existing tool with a combination of symbolic execution and SMT solving. Our new tool generates the test cases for SIP in seconds. However, further experiments showed that our approach is not a substitutive but a complementary approach: we present the technique and the results obtained for two protocol specifications, the first supporting our new technique, the second being witness for the classic BDD-technique.

Keywords—model-based testing, symbolic execution, conformance testing, symbolic input-output conformance, symbolic test case generation

I. INTRODUCTION

In recent years model-based testing has gained interest with the growing acceptance of models in software/hardware design and development. Of particular importance are formal models with precise semantics. Testing with such models allows one to measure the conformance of the system under test with the model. Furthermore, such formal models can be analyzed by formal techniques drawn from areas, like formal verification, model checking, and control-flow analysis. The ultimate aim of this analysis is the automated generation of test cases in order to reduce the huge testing efforts.

As with all complex automation tasks, there is no single approach serving all problem domains equally well. Hence, a major task of today’s research in model-based testing is to find the right mix of tools for specific problem domains. This is the line of research this paper is contributing to.

In problem domains, where data plays an essential part, explicit data enumeration techniques are not well-suited, because they quickly suffer from the state-space explosion problem, well-known from model checking. One way to overcome this problem is the use of symbolic techniques. However, as this paper is going to demonstrate, we need a set of symbolic techniques to serve our needs.

In our particular case, we are testing implementations of communication protocols with heavy data dependencies, e.g. the Session Initiation Protocol (SIP) used in Voice-over-IP applications. We applied the non-symbolic test case generator TGV [1] successfully to find bugs in this protocols [2]. However, we were not fully satisfied with the needed abstractions of and restrictions to the models in order to avoid state-space explosion. Therefore, we switched to the symbolic test case generator STG [3].

The STG tool implements the approach of Rusu et al. [4] extended to models with infinite state space [5]. In this tool, the test case generation relies on binary decision diagrams (BDDs, [6]) for calculating the sets of relevant states.

However, we encountered the following problem: STG was not able to generate a single test case for SIP. Our machines ran out of memory, an indication for state-space explosion. Knowing that BDDs may fail for models with many variables, we replaced the BDD-engine of STG with our own test case generator based on symbolic execution. Our new tool generates the test cases for SIP in seconds. In this paper we present our new approach and the gathered data of two case studies, the SIP protocol and the Conference Protocol. The latter demonstrates that we did not find the silver bullet for symbolic test case generation: STG outperforms our tool in case of the Conference Protocol, showing that the presented approach is complementary to STG.

After this introduction, we briefly introduce symbolic transition systems in Section II. Then, we present our approach for test case generation in Section III. In Section IV we discuss experimental results of applying our approach to two protocol specifications. Furthermore, this section compares the results of our approach with the results obtained when using STG. Finally we discuss related research in Section V and conclude in Section VI.

II. PRELIMINARIES

Our work uses symbolic transition systems (STS) for representing specifications and test purposes.

A. Symbolic Transition Systems

Different formalizations of symbolic transition systems can be found in the literature, e.g., [7], [4], [8]. While these transition systems are very similar, there are subtle differences in their semantics. As our approach reuses some techniques of the STG tool, we rely on symbolic transition systems as proposed by Rusu et al. [4].
A symbolic transition system is a tuple \( (D, \Theta, Q, q_0, \Sigma, T) \), where \( D = V \cup P \cup M \) is a set of typed data, whereas \( V \) is the set of variables, \( P \) is the set of system parameters, and \( M \) is the set of action parameters. \( V, P, \) and \( M \) are pairwise disjoint. \( D \) is nonempty and finite. \( \Theta \) is the initial condition on elements in \( V \cup P \). \( Q \) is the nonempty and finite set of locations. \( q_0 \in Q \) is the initial location. \( \Sigma = \Sigma^i \cup \Sigma^o \cup \Sigma^{int} \) is the nonempty, finite alphabet, whereas \( \Sigma^i \) is the set of input actions, \( \Sigma^o \) is the set of output actions, and \( \Sigma^{int} \) is the set of internal actions. Each action \( a \in \Sigma \) is possibly carrying typed action parameters. \( T \) is the set of transitions.

A transition is a tuple \( \langle q, a, \mu, G, A, q' \rangle \) consisting of: the initial location \( q \in Q \), an action \( a \in \Sigma \), the action parameters of \( a \) denoted by \( \mu \), the guard \( G \), a context update \( A \), and \( q' \in Q \) the destination location of the transition.

Note that symbolic transition systems can be used to model non-deterministic systems.

Example 1. Fig. 1 depicts an STS that reads three input values of type integer, representing the three side lengths of a triangle. Then the type of the triangle is determined. This STS has five locations, i.e. \( Q = \{l_0, \ldots, l_4\} \), \( q_0 = l_0 \), \( V = \{a, b, c\} \), \( M = \{p, q, r\} \) and \( P = \emptyset \). Furthermore, \( \Theta = \text{true} \) and the alphabet is given by \( \Sigma^i = \{\text{Read}\} \), \( \Sigma^{int} = \{\tau\} \) and \( \Sigma^o = \{\text{NotPositive}, \text{NotTria}, \text{IsTria}, \text{Scalene}, \text{Isosceles}, \text{Equilateral}\} \).

We call an STS \( T = \langle D, \Theta, Q, q_0, \Sigma, T \rangle \) initialized if each system parameter is substituted by a valid value and the initial condition \( \Theta \) constrains each variable \( v \in V \) to exactly one value. An STS is deterministic if any sequence of actions (including instantiated action parameters) has at most one successor state. Furthermore, an STS is complete if every action is enabled in every state.

B. Product of Symbolic Transition Systems

Test purpose based test case generation usually relies on the calculation of the product between the specification and the test purpose. In case of STSs, the two STS need to be compatible for the product operation [10].

Definition 2 (Compatible for Product): Two STS \( I_1 \) and \( I_2 \) with data sets \( D_1 = V_1 \cup P_1 \cup M_1 \) and \( D_2 = V_2 \cup P_2 \cup M_2 \) and alphabets \( \Sigma_1 = \Sigma^i_1 \cup \Sigma^o_1 \cup \Sigma^{int}_1 \) and \( \Sigma_2 = \Sigma^i_2 \cup \Sigma^o_2 \cup \Sigma^{int}_2 \) are compatible for product if (1) \( V_1 \cup V_2 = \emptyset \) and \( M_1 \cap M_2 = \emptyset \); and (2) \( \Sigma_1^i = \Sigma_2^i \), \( \Sigma_1^o = \Sigma_2^o \), and \( \Sigma_1^{int} = \Sigma_2^{int} \). The parameters of the actions must have the same types.

Given two compatible symbolic transition systems, the product is defined as follows:

Definition 3 (Product): The product of two STS \( I_1 = \langle D_1, \Theta_1, Q_1, q_{01}, \Sigma_1, T_1 \rangle \) and \( I_2 = \langle D_2, \Theta_2, Q_2, q_{02}, \Sigma_2, T_2 \rangle \) is given by \( P = \langle D, \Theta, Q, q_0, \Sigma, T \rangle \), where

\[
V = V_1 \cup V_2, \quad P = (P_1 \cup P_2) \setminus (V_1 \cup V_2), \quad M = M_1 \cup M_2, \quad \Theta = \Theta_1 \land \Theta_2, \quad Q = Q_1 \times Q_2, \quad q_0 = (q_{01}, q_{02})
\]

\[
\Sigma^i = \Sigma_1^i \cup \Sigma_2^i, \quad \Sigma^o = \Sigma_1^o \cup \Sigma_2^o, \quad \Sigma^{int} = \Sigma_1^{int} \cup \Sigma_2^{int}
\]

\( T \) is given by the smallest set of transitions satisfying the following inference rule:

\[
\left\{ (q_1, a, \mu_1, G_1, A_1, q'_1) \in T_1, (q_2, a, \mu_2, G_2, A_2, q'_2) \in T_2 \right\} \rightarrow \left\{ (q_1, a, \mu_1, G_1 \land G_2[a_2/\mu_2], A_1 \cup A_2[a_2/\mu_2], \langle q'_1, q'_2 \rangle) \in T \right\}
\]

\( G_2[a_2/\mu_2] \) denotes the guard of the transition of \( T_2 \) in which each action parameter of \( a_2 \) is substituted by the corresponding action parameter of \( \mu_2 \). \( A_2[a_2/\mu_2] \) denotes the set of assignments with the same replacement of action parameters.

Example 2. The product between the STS of Fig. 1 and of Fig. 3(b) is shown in Fig. 2. As the two source STS have no shared variables and action parameters and their alphabet is the same, they are compatible for the product operation.

C. Test Purposes and Test Cases

A test purpose is a formalization of what a test engineer intends to exercise. As we are using symbolic transition systems, test purposes are given in terms of STS.

Definition 4 (Test Purpose): Given a specification \( S \) in the form of an STS, a test purpose is an initialized, complete and deterministic STS \( TP = \langle D, \Theta, Q, q_0, \Sigma, T \rangle \), equipped with two disjoint sets of trap states: \( \text{Accept}^{TP} \subset Q \) defines pass verdicts, and \( \text{Reject}^{TP} \subset Q \) limits the exploration of the
Definition 5 (Test Case): A test case is an initialized, deterministic STS \( TC = (D, \Theta, Q, q_0, \Sigma, T) \) without internal transitions, where \( Q \) comprises three pairwise disjoint sets of trap states, \( \text{Fail} \subset Q, \text{Pass} \subset Q, \) and \( \text{Inconclusive} \subset Q \) defining fail, pass and inconclusive verdicts. Furthermore, a test case is controllable, i.e., in any state a test case either provides a single stimulus to the implementation or it accepts all responses from an implementation.

Inconclusive verdict states are commonly used in test purpose based testing to denote that the implementation behaved correctly, but the test goal defined by the test purpose has not been reached. As usual, we invert the notion of inputs and outputs for test cases: Inputs of a test case are outputs of the implementation/specification and vice versa.

III. Symbolic Execution Based Test Case Generation

Having a notion of specifications (STS), test purpose and test cases we want to automatically derive test cases. We propose to use a combination of the approaches of Rusu et al. [4] and Gaston et al. [8]. The former use Abstract Interpretation implemented by means of BDDs to approximate the traces accepted by the test purpose. In contrast, the latter first symbolically execute the specification and then apply the test purpose which is given in terms of traces of the symbolic execution.

By combining both approaches, we gain the benefit that the specification is first restricted by the test purpose. The search for \( \text{Accept} \) locations is then implemented by means of symbolic execution and SMT solving. More precisely, we calculate a test case as follows:

1) Completion of the test purpose: A test purpose has to be complete (see Def. 4). As a user may provide an incomplete test purpose, we automatically complete it first.

2) Product of the specification and the test purpose: Then we calculate the product of the specification and the test purpose. The construction of the product stops in \( \text{Accept} \) and \( \text{Reject} \) states.

3) Closure of the product: As the product may comprise internal transitions and may be non-deterministic, we remove the internal transitions and make the product deterministic. Furthermore inputs and outputs are inverted.

4) Symbolic execution: The traces, relevant w.r.t. the test purpose, are calculated by symbolically executing the product.

5) Test tree selection: From the symbolic execution we select a trace leading to an \( \text{Accept} \) state of the test purpose. This trace is then extended to a tree by adding necessary input-labeled transitions (i.e., possible outputs of an implementation).

6) Test tree transformation: As the test tree is given in terms of the symbolic execution, we transform the tree to a symbolic transition system.

A. Completion, Product and Closure

We complete our test purposes in the same way as described in [10]: if a location \( q \in Q \) has missing outgoing transitions, these transitions are added in terms of self-loops or as transitions leading to \( \text{Reject} \) states. For every added transition, the set of assignments is the set of identity assignments, i.e., variables do not change their values.

Example 4. Completing the test purpose of Fig. 3(a) leads to a test purpose as illustrated in Fig. 3(b).

Once the test purpose has been made complete, the product of the specification and the completed test purpose is calculated. Recall that test cases use an inverse notion of input and output, i.e., an output of a test case is an input in terms of the implementation/specification. Thus, we invert the inputs and outputs after the product operation.

As internal (unobservable) transitions are meaningless for test cases they have to be removed. This elimination of internal actions is called closure and works for STS without cycles of internal actions as follows. Let \( \tau_1, \tau_2, ..., \tau_n \) be a sequence of internal actions leading to an action \( a \). The guard belonging to \( \tau_i \) is denoted by \( G_i \) and the corresponding set of assignments is given by \( A_i \) for \( 1 \leq i \leq n \). The action \( a \) is guarded by \( G \) and triggers the assignments \( A, \mu \) are the parameters of the action \( a \). The whole sequence of symbolic transitions can be replaced by one symbolic transition \( t = \langle q_0, a, \mu, G_1 \land (G_2 \circ A_1) \land ... \land (G_n \circ A_n^{-1} \circ ... \circ A_1) \land (G \circ A_n \circ A_{n-1} \circ ... \circ A_1), A \circ A_n \circ A_{n-1} \circ ... \circ A_1, q_{n+1} \rangle \), where \( f \circ g \) denotes composition of the functions \( f \) and \( g \).

Finally, in order to avoid a dependency of the verdicts on the internal choices of the tester, we make test cases deterministic. The goal of the determinization step is to compute an STS which has no non-deterministic choices and is trace-equivalent to the original STS. As determining
symbolic transition systems is difficult in general, Rusu et al. [4] propose a heuristic. We use this heuristic where the basic idea is to delay the effect of nondeterminism until nondeterminism can be resolved.

**Example 5.** Fig. 4 shows the deterministic closure of Fig. 2. Inputs and outputs have been inverted. As Fig. 2 is already deterministic, the path condition cannot be resolved. As Fig. 4 is deterministic, the path condition can be resolved.

**B. Symbolic Execution**

Symbolic execution techniques for STS were introduced by Gaston et al. [8]. The main idea is the same as for symbolically executing programs. Concrete values of action parameters as well as initialization values for STS variables are replaced by symbolic values, which have unique names. During the symbolic execution, the constraints on these variables are computed. The result of this execution is a symbolic execution tree:

**Definition 6 (Symbolic Execution Tree):** A symbolic execution tree is a tuple \( \langle S, L, \eta_0, T \rangle \) where \( S \) is a finite set of symbolic extended states, \( L \) is a finite set of symbolic communication actions, \( \eta_0 \) is the initial symbolic extended state and \( T \) is the transition relation. A transition \( \langle \eta, sa, \eta' \rangle \in T \) comprises a starting state \( \eta \in S \), a final state \( \eta' \in S \) and a symbolic communication action \( sa \in L \).

The states of a symbolic execution tree are so called symbolic extended states:

**Definition 7 (Symbolic Extended State):** A symbolic extended state (SES) in a symbolic execution tree of an STS \( \mathcal{I} = \langle D, \Theta, Q, q_0, \Sigma, T \rangle \) is a triple \( \eta = \langle q, \pi, \sigma \rangle \), where \( q \in Q \), \( \pi \) is the path condition and \( \sigma \) is a mapping from variables and action parameters to their symbolic values.

An SES is satisfiable if its path condition \( \pi \), which is a Boolean expression, is satisfiable. The symbolic extended states are the nodes of the symbolic execution tree. These nodes are connected with symbolic transitions.

**Definition 8 (Symbolic Transition):** A symbolic transition is a triple \( \langle \eta, sa, \eta' \rangle \), where \( \eta \) is the source SES, \( \eta' \) the destination SES and \( sa \) is the symbolic communication action.

Symbolic transitions are labeled by actions.

**Definition 9 (Symbolic Communication Action):** A symbolic communication action is a tuple \( sa = \langle a, \mu_{sa}, \sigma_{sa} \rangle \), where \( a \) is the corresponding action in the STS and \( \mu_{sa} \) is a list of unique identifiers denoting the action parameters of \( sa \). \( \sigma_{sa} \) is a mapping from the original action parameter names to the symbolic communication action's unique parameter names in \( \mu_{sa} \).

**Example 6.** Fig. 5 shows a symbolic execution tree with the five states \( \eta_0, \ldots, \eta_4 \). For the sake of brevity, we omit \( \mu_{sa} \) on the tree's transitions and identity assignments of \( \sigma_{sa} \).

An algorithm for symbolically executing STS has been introduced by Gaston et al. [8]. However, the used STS slightly differ from the one used in this paper. The main difference is the use of action parameters as arguments for actions. The STS definition used in this work does not support the use of variables, system parameters, or terms thereof as arguments of any actions. Hence, temporary variables only visible in a certain transition, so-called messages (action parameters), have to be used. To compensate this difference, the original algorithm of Gaston et al. [8] had to be adapted.

Algorithm 1 illustrates the adapted symbolic execution procedure. It starts by generating a unique symbolic variable for each element of \( D \) (Line 1). This mapping together with the initial state \( q_0 \) and the path condition \( \text{true} \) serves as initial state \( \eta_0 \) of the symbolic execution tree. The state \( \eta_0 \) is added to the set of states to be visited \( S \) (Line 2).

As long as there are unvisited states in \( S \) (Line 3), the algorithm takes (and removes) one state \( \eta \) from \( S \) (Line 4). The symbolic state \( \eta \) refers to a state \( q \) of the symbolic transition system \( \mathcal{I} \); Algorithm 1 executes every outgoing transition of \( q \) (Line 5). Therefore, it generates the corresponding symbolic communication actions \( sa \) and the symbolic transitions \( \langle \eta, sa, \eta' \rangle \). For each action parameter of \( a \) a new symbolic value is generated: the list of new values is stored in \( \mu_{sa} \). Additionally, the mapping between action parameters and symbolic values is generated (Line 7). In contrast to Gaston et al., we do not only create new symbolic variables for input actions, but also for output actions.

Our algorithm then continues by calculating the path condition \( \pi' \) for \( \eta' \). The path condition is the conjunction of the path condition \( \pi \) with the guard \( G \) of the STS's transition. The variables within \( G \) are replaced by their symbolic values, i.e., we apply the mappings \( \sigma \) and \( \sigma_{sa} \) to \( G \). Here is the second difference to the algorithm of Gaston et al.: as we do not use variables directly as action parameters we have to calculate the path condition by \( \pi' = \pi \land \sigma(\sigma_{sa}(G)) \) instead of using \( \pi' = \pi \land \sigma(G) \).

Finally, we add \( \eta' \) to the list of states to be visited (Line 13), if the path condition of \( \eta' \) is satisfiable (Line 12).

The problem of deciding about the satisfiability of Boolean formulas is NP-complete [11]. Nevertheless, recent advances in SAT (Boolean satisfiability) solving have yielded powerful SAT solvers. In most cases, they can efficiently handle problems with up to millions of clauses and variables [12].

We currently support only Boolean and integer data type. Thus, the path condition \( \pi \) is a Boolean expression over
Algorithm 1 Symb.Exec. of $\mathcal{I} = (D, \Theta, Q, q_0, \Sigma, T)$.
1: $\sigma_0 \leftarrow$ map of variables of $D$ to symbolic values
2: $\eta_0 \leftarrow \{ q_0, \text{true}, \sigma_0 \} \quad S \leftarrow \{ \eta_0 \}$
3: while $S \neq \emptyset$ do
4: \hspace{0.5em} pick and remove some $\eta = \langle q, \pi, \sigma \rangle$ from $S$
5: \hspace{1.0em} for all $\langle q, a, \mu, G, A, q' \rangle \in T$ do
6: \hspace{1.5em} $\mu_{sa} \leftarrow$ list of unique symb. values for every param. of $a$
7: \hspace{1.5em} $\sigma_{sa} \leftarrow$ map of action parameters to symbolic values
8: \hspace{1.5em} $sa \leftarrow (a, \mu_{sa}, \sigma_{sa})$
9: \hspace{1.5em} $\sigma' \leftarrow \sigma \circ \sigma_{sa} \circ A$
10: \hspace{1.5em} $\eta' \leftarrow \langle q', \pi', \sigma' \rangle$
11: \hspace{1.5em} if $\langle \eta', \pi', \sigma' \rangle$ is the calculated symbolic transition
12: \hspace{2.0em} $\eta'$ is not included in a previously generated state then
13: \hspace{2.5em} $S \leftarrow S \cup \{ \eta' \}$
14: \hspace{1.5em} end if
15: \hspace{1.0em} end for
16: end while

identifiers and constants of type Boolean and/or integer, which may include arithmetic operators. SMT (Satisfiability Modulo Theories) solvers are an extension to SAT solvers, which support arithmetics and other first-order theories like uninterpreted functions, arrays, or recursive datatypes [12]. Hence, for our application, SMT solvers are better suited than pure SAT solvers, since they allow the direct use of arithmetics in Boolean formulas.

Besides the satisfiability of path conditions, we use the state inclusion (Line 12 of Algorithm 1) of Gaston et al. [8] for reducing the number of symbolic states to be explored.

Definition 10 (State inclusion): Given two symbolic extended states $\eta = \langle q, \pi, \sigma \rangle$ and $\eta' = \langle q', \pi', \sigma' \rangle$, then $\eta'$ is included in $\eta$, i.e., $\eta' \subseteq \eta$, iff $q = q'$ and $(\pi' \land \bigwedge_{x \in A} (x = \sigma'(x)))$ is a tautology.

Again, checking for state inclusion is implemented by means of an SMT solver. Obviously, a formula is a tautology if its negation is unsatisfiable. Hence, the SMT solver is used to decide whether a state is included in another. Therefore we negate the formula, extend it by the necessary variable declarations and then give it to the SMT solver.

Despite using state inclusion and satisfiability of path conditions, there is still the possibility of non-termination of the symbolic execution. If there are loops whose number of iterations depend on symbolic inputs, we have an infinite symbolic execution tree. In order to guarantee program termination, we introduce an upper bound for loop unfolding.

The predicate upperBoundReached($q$) returns true if the number of SES in the current path that correspond to the STS location $q$ exceeds a certain bound. For example, an upper bound of two means that each path of the symbolic execution tree has at most two SES corresponding to the same STS location. If the upper bound is too small, the symbolic execution may terminate too soon and the test case selection may fail because there is no satisfiable Accept state.

Example 7. Fig. 5 shows the symbolic execution tree (SET) obtained when applying Algorithm 1 to the STS of Fig. 4.

The algorithm starts by generating the initial state $\eta_0$ (Line 2) of the SET. This state is given by the initial state $p_0$ of the STS, the initial path condition true and a mapping $\sigma_0$ which has one symbolic value (e.g., $a_0, b_0$) for every element in $D$ of the STS.

The algorithm then picks $\eta_0$ from $S$ (Line 4) and processes every outgoing transition of the state $p_0$ (Line 5), which is only one. First $\mu_{sa} = \{ p_1, q_1, r_1 \}$ and $\sigma_{sa} = \{ p \mapsto p_1, q \mapsto q_1, r \mapsto r_1 \}$ are generated. This leads to the symbolic communication action $sa = \langle \text{Read}(p,q,r), \mu_{sa}, \sigma_{sa} \rangle$. Then, $\pi' = \pi \land \sigma \circ \sigma_{sa} \circ A$ is a tautology.

$\eta'$ is satisfiable and $\eta = \langle p_1, \text{true}, \sigma_1 \rangle$ is not included in $\eta_0$, $\eta'$ is added to $S$.

C. Test Tree Selection

Once the symbolic execution tree has been generated, we need to select a test case from the set of test cases described by the execution tree. Therefore, we first select a test tree, i.e., some parts of the execution tree. This tree is later transformed into an STS. While a test tree consists of symbolic extended states and symbolic transitions, it satisfies already the controllability property of test cases.

Our algorithm for the selection of a test tree from a symbolic execution tree is based on the idea of selecting accepted behavior similarly to the TGV tool [1]. Therefore, we randomly select one satisfiable Accept state $\eta_A = \langle q_A, \pi_A, A \rangle$. Similar to TGV we say reaching one Accept state is sufficient to have a test case satisfying the test purpose.

Starting from $\eta_A$ the path leading to the root SES is selected by backtracking. In a next step, the selected path is extended to a tree by adding missing inputs (possible outputs of the system under test) that are allowed by the specification. These transitions are needed in order to prevent fail verdicts on allowed outputs from the system under test. Instead of giving inconclusive verdicts when observing such
outputs, we extend these additional transitions by adding parts of the symbolic execution tree. Note that during this forward traversal we ensure controllability of the generated test tree. We stop the test tree selection in terminal states, i.e., in states with no outgoing transitions.

In other words, one state in a test tree either has only inputs or only one output as outgoing symbolic transitions. **Example 8.** Selecting a test tree from the symbolic execution tree of Fig. 5 results into the whole SET. This is because there is only one reachable \textit{Accept} state ($\eta_4$). Extending this tree by adding missing inputs leads to the whole SET. □

D. Test Tree Transformation

As specifications and test purposes, test cases are symbolic transition systems. Thus, finally, we need to transform the selected test tree $TT = \langle S, L, \eta, T \rangle$ into a test case $TC = \langle D, \Theta, Q, q_0, \Sigma, \tau, T \rangle$. Note that the resulting STS is still tree-structured.

The data, i.e., $D$, of $TC$ is given by the symbolic values of $TT$. That is, we interpret the symbolic values as variables of the test case. Let $\eta_0 = \langle q, \pi_0, \sigma_0 \rangle$, then $\Theta = \pi_0$, $q_0 = \eta_0$, $Q = S$, and $S = \bigcup_{(a, \mu, A, \sigma) \in L} a$.

Let $\langle \eta, sa, \eta' \rangle \in T$ be a transition of the symbolic execution tree, then the corresponding test case transition $\langle q, a, \mu, G, A, q' \rangle \in T$ is obtained as follows. $q = \eta$ and $q' = \eta'$. Furthermore, $a$ is the action of $sa = \langle a, \mu_{sa}, \sigma_{sa} \rangle$. The action parameters of $a$ are given by $\mu_{sa}$, i.e., $\mu = \mu_{sa}$. While $A$ is the identity assignment, the guard $G$ is given by the path condition of $\eta' = \langle q, \pi, \sigma \rangle$, i.e., $G = \pi$.

**Example 9.** The test case obtained from the symbolic execution tree of Fig. 5 is shown in Fig. 6. The guards of the transitions are given by the path conditions of the final states of the corresponding end state in the SET. The pass state is defined by the \textit{Accept} state of the SET. Inconclusive verdicts (inconc.) are assigned to non-\textit{Accept} terminal states. □

IV. EXPERIMENTAL RESULTS

We have implemented the presented approach in JAVA™. Our tool uses the same input language as the STG tool and reuses some parts of STG (product calculation, and closure). Thus, we use STG to conduct the steps one, two and three of our approach (see Section III). For a more detailed discussion of our tool we refer the reader to [13].

In this section, we present the results obtained when applying our tool and the STG tool to two different protocols:

(1) the Session Initiation Protocol (SIP) [14] and (2) the Conference Protocol (CP) [15].

Table I summarizes the characteristics of the two specifications in terms of lines of code, the number of variables, the number of locations, the number of transitions, and the number of input, output, and internal actions. Furthermore, this table shows the number of action parameters (act.param.), the number of arithmetic parameters used in guards and the number of logical operators used in guards. For these characteristics, we list the minimum, the maximum and the average number. We conducted our experiments on a PC with Intel© Dual Core Processor 2 GHz and 2 GB RAM.

**Session Initiation Protocol:** SIP is an RFC standard [14] and handles communication sessions between two end points in the context of Voice-over-IP networks. SIP defines various entities that are used within a SIP network. One of these entities is the Registrar, which is responsible for maintaining location information of users. We developed an STS representing the full functionality of a SIP Registrar.

**Conference Protocol:** This Protocol has been used previously to analyze the failure-detection ability of different formal testing approaches, e.g., [16]. The specification and 27 erroneous implementations are available to the public. The protocol itself is a simple communication protocol for a chat application. Users can join conferences, exchange messages and leave conferences. Each user has a nickname and can join one conference at a time.

A. Test Case Generation

1) Test Purposes: For each of the two applications under test we identified the relevant scenarios from the informal specifications. We formalized these scenarios in terms of test purposes. The two tables II and III give an overview of the size of the (intermediate) results produced throughout test case generation. We use three metrics to quantify their size (top rows of the tables): (1) the lines of code needed to specify them in STG’s input format, (2) the number of locations, and (3) the number of transitions. Remarkably, the test purposes for the Conference Protocol are much larger than the test purposes for the Session Initiation Protocol. This difference is mainly because the Conference Protocol has four different observable actions while the SIP specification only uses two different observable actions. Thus, refusing all actions except one for the Conference Protocol takes thricre the amount of code of the SIP specification.

2) Closed Products: The second parts of these tables describe the closed products derived from the specifications by means of our test purposes. Note that for calculating the closed product, we reused parts of the STG tool. According to Zinovieva-Leroux [10], the current version of STG does not support determinization yet. Consequently, our closed products are not deterministic. We measure the size of each closed product in terms of the number of locations, transitions, variables, and different action parameter names.
Although the specification for the SIP Registrar is larger than the one of the Conference Protocol, the average size of the closed products is similar. Thus, the selected parts of the specifications are almost of equal size.

3) Symbolic Execution Trees: The symbolic execution trees for the closed products are described in the middle of the tables II and III. They state (1) the number of symbolic extended states, (2) the number of satisfiable Accept states, and (3) the upper bound, which is used as an upper limit for loop unfolding during symbolic execution.

On average, a symbolic execution tree for the Conference Protocol has 26572 SES, whereas a symbolic execution tree for the SIP Registrar has only 210 SES. Hence, the Conference Protocol causes huge symbolic execution trees compared to the symbolic execution trees of the SIP Registrar, while their closed products are almost of the same size. An explanation for this circumstance may be the internal structure of the closed products. By trend, STS with many loops result in larger symbolic execution trees.

4) Generated Test Cases: The two tables II and III also describe the test cases that were extracted from the symbolic execution trees. Five metrics illustrate their size. They count (1) the total number of SES in each test case, (2) the number of SES in the longest path, (3) the number of SES in the shortest path, (4) the number of Pass verdict states, and (5) the number of Inconclusive verdict states in each test case.

For the SIP Registrar specification and its six different test purposes, our approach generated test cases for all of the used test purposes within seconds. Nevertheless, our approach did not succeed in generating a test case for two test purposes for the Conference Protocol.

5) Test Case Generation Time: Finally, the amount of time that was needed for test case generation by the STG tool and the amount of time that was taken by our implementation are listed at the bottom of the tables II and III. The elapsed time for test case generation with our approach is broken down into two parts: (1) the amount of time that is needed by STG to complete the test purpose, to calculate the product, and to close the product, and (2) the amount of time that is consumed by the symbolic execution of the closed product and the test case selection from the resulting symbolic execution tree. The rows labeled by termination state whether STG and our approach terminated successfully, i.e., whether they generated test cases.

For the SIP Registrar specification and its six different test purposes, STG was not able to generate any test cases. In contrast, our approach successfully generated test cases for all of the used test purposes within seconds. Fig. 7 illustrates the elapsed time until STG reported an error and the time needed by our approach. For none of the used test
purposes, our approach took longer than 15 seconds. Test case generation needed in average 7 seconds.

In contrast, for the Conference Protocol, STG did significantly perform better than our approach. First of all, STG was able to generate test cases for all test purposes. Our approach did not manage to generate test cases for two test purposes, in reasonable time. Furthermore, STG did not manage to generate test cases for two test purposes. Our approach took up to about ten hours to successfully create a test case. Fig. 8 illustrates this difference.

Figure 9 and Figure 10 partition the test case generation time needed by our new implementation for each test purpose into four tasks: (1) the parsing of the closed product, (2) the calculation of the symbolic actions and SES, (3) the satisfiability checking of the path conditions, and (4) the state inclusion checking.

**B. Discussion**

Compared to STG, our approach works well for the SIP Registrar for which STG cannot generate any test cases.

However, it suffers from performance problems when being applied to the Conference Protocol. While STG uses BDDs, our approach is based on a combination of symbolic execution and SAT (SMT) solving. In the field of model checking, it has been observed that BDD-based model checkers and SAT-based model checkers are often able to solve different classes of problems. Hence, BDD-based techniques and SAT-based techniques are assumed to be complementary [17], [18]. The two test case generation
techniques presented in this work seem to behave similarly, although our approach is not solely based on SAT solving.

STG uses BDDs, which are limited in their applicability due to their memory requirements. In the worst case, BDDs grow exponentially with the size of the Boolean formula they are representing. Hence, the most common failure related to BDDs is “running out of memory” [19]. This conforms to our experience made by applying STG to the SIP Registrar specification, which uses very large guards and a considerable number of variables and action parameters.

By contrast, our approach employs a combination of symbolic execution and SAT solving and does not perform well for the Conference Protocol, which comprises smaller guards and less variables. Nevertheless, its symbolic execution trees are huge and the longest paths in the resulting test cases contain significantly more SES than the longest paths in the SIP Registrar test cases. Again, an analogy to model checkers exists: It has been observed that BDD-based techniques show a good performance for designs with deep counterexamples, whereas SAT-based model checking techniques are more effective for designs with shallow counterexamples [20].

V. RELATED RESEARCH

Test purpose based testing is an active area of research and many tools make use of test purposes for steering the test case selection. For example, the TGV tool [1] uses test purposes given in terms of labeled transition systems. Test purposes for testing real-time systems have been considered in [21]. The SpecEXPLORER tool does not rely on a single representation of a test purpose but combines various techniques [22], e.g., the tool offers state filtering or the selection of a representative set of data values for variables. Symbolic transition systems for the modelling of specifications and test purposes have been considered in [4], [5], [8].

Rusu et al. [4] calculate the product between a given symbolic test purpose and a symbolic specification. It contains the symbolic traces that were selected from the specification by the test purpose. Internal actions and non-deterministic choices are eliminated from the product. The final test case is obtained by selecting parts that lead to the Accept locations of the test purpose. During this selection, simplifications are performed, e.g., unreachable states are pruned. Finally, the test case is made input-complete, i.e., missing transitions are added such that they lead to a new location Fail. [5] extend this approach to infinite-state symbolic models.

The application of symbolic execution in the area of testing was already proposed in the 1970s [23]. More recently, the use of this technique has been revitalized.

The AGATHA tool set [24] applies symbolic execution on EIO LTS, which are basically the same as STS, in order to validate system specifications. In contrast to this work, AGATHA does not use the concept of test purposes and symbolically executes the whole system specification. Gaston et al. [8] extend AGATHA in order to assist test purpose definition. Test purposes are subtrees of the symbolic execution tree and may either be chosen manually or derived by two coverage criteria.

SYMSTRA [25] generates unit tests that reach a high branch and intra-method path coverage for object-oriented systems. A framework for correctness checking and test input generation based on symbolic execution and model checking is presented in [26].

Many testing tools (e.g. SELECT [27], DART [28], EXE [29], PEX [30]) combine concrete and symbolic execution, which is often referred to as dynamic symbolic execution or concolic execution.

VI. CONCLUSIONS

We have developed a new tool for test case generation from symbolic transition systems. Our test case generator takes STG input files, uses the product building of STG and then applies our own symbolic execution technique to generate test cases. An external SMT solver is called to check for satisfiability of paths and symbolic state inclusion.

The contributions of this work can be summarized as follows: (1) a new approach for generating test cases that works in case the BDDs fail, (2) a tool implementing this approach and interfacing with STG, (3) two case studies with experimental data showing both the advantages and limitations of our approach.

Immediate future work is the implementation of the test execution driver and the actual testing of the protocol implementations. Then, data about the error-detection rate can be given. Furthermore, a mapping from UML statecharts to symbolic transition systems is under development. Another direction of further research is to find indicators on the level of the model that allows one to choose between the BDD-based approach and the approach presented in this paper. Finally, while we only select one test case satisfying a test purpose it would be interesting to extend our technique for generating multiple test cases per test purpose.

ACKNOWLEDGMENT

The research herein is partially conducted within Sofnet Austria and funded by the Austrian Federal Ministry of Economics, the
province of Styria, the Steirische Wirtschaftsförderungsgesellschaft mbH, and the city of Vienna in terms of the center for innovation and technology. This research is also partly funded by the EU FP7 project MOGENTES ICT-216679.

REFERENCES