Symbolic execution of transition systems with function summaries

Imen Boudhiba, Christophe Gaston, Pascale Le Gall, Virgile Prévosto

Symbolic execution of transition systems with function summaries *

Imen Boudhiba[1], Christophe Gaston[2], Pascale Le Gall[1]
and Virgile Prevosto[2]

[1] Laboratoire MICS, Grande Voie des Vignes, 92195 Châtenay-Malabry, France
email: {imen.boudhiba,pascal.legall}@centralesupelec.fr
[2] CEA LIST, Point Courrier 174, 91191, Gif-sur-Yvette, France
email: {christophe.gaston,virgile.prevosto}@cea.fr

Abstract. Reactive systems can be modeled with various kinds of automata, such as Input Output Symbolic Transition Systems (IOSTS). Symbolic execution (SE) applied to IOSTS allows computing constraints associated to IOSTS path executions (path conditions). In this context, generating test cases amounts to finding numerical input values satisfying such constraints using solvers. This paper explores the case where IOSTS models contain functions which are outside of the scope of such solvers. We propose to use *function summaries* which are logical formulas built from concrete values describing some representative input/output data tuples of the function. We define algorithmic strategies to solve path conditions including such functions based on techniques using and enriching function summaries. Our method has been implemented within the Diversity tool and has been applied to several examples.

Keywords: Input Output Symbolic Transition Systems, Functions summaries, Symbolic Execution, Transition coverage.

1 Introduction

Many testing theories and algorithms use Symbolic Execution (SE) techniques [10]. In the last decade, it has received much attention and has been incorporated in several testing tools such as NASA's Symbolic (Java) PathFinder [29], UIUC's CUTE and jCUTE [21], UC Berkeley's CREST [13], and the CEA's PathCrawler [28] and DIVERSITY tools [15]. In particular, for the latter one, SE has been adapted for models using variants of abstract labeled transition systems, namely Input Output Symbolic Transition Systems (IOSTS) [17]. Symbolic trees representing all possible execution paths of the model (up to some coverage goals) are computed by executing the model with variables instead of concrete values. For a particular path, a constraint on these variables, called path condition, characterizes concrete values ensuring its execution. Sequences of concrete test inputs

* Part of this work has been conducted within the French PIA project SESAM Grids 2012-2016 [12] and the Vessedia project funded from European Union's Horizon 2020 research and innovation programme under grant agreement No 731453.

exercising a given path are computed by solving the corresponding path condition. As is the case for programs, one of the main limits of this approach is that the usage of some particular functions in the model may make the process inapplicable, either because their symbolic analysis would cause combinatorial explosion in the number of paths to be considered, or because they contain operations that go beyond the capacity of the solver used, or even simply because they are black box functions from the model point of view. This latter situation occurs for example when the modeler makes a reference to an executable function without accessible source code in its model (or without source code processable by the SE tool). In this paper, we call such functions *external functions* and they are assumed to be functionally correct (*i.e.* we suppose that we have a reference implementation for each of the external functions used in the model). In the frame of Model-based Testing [33], a classical way to deal with this situation is to fully abstract external functions by considering the results of their calls as random values. This makes the model behaviors highly non deterministic and causes the test case generation process to compute test cases with a low level of controllability: the behavior of the system under test may deviate from the execution path which it is supposed to exercise without revealing an error in the system. Those situations are referred to as *inconclusive*.

In this paper we adapt to models an approach used at the code level, which is based on a representation of external functions as so-called *function summaries*. A function summary is a logical formula that can be computed from a partial knowledge of the function, represented as a table containing some representative tuples of inputs/output data. Those tuples are obtained by (concretely) executing the function on a set of inputs, either produced randomly or given by the programmer. They may also result from pre-existing unit test campaigns. Path conditions are then computed based on a joint analysis of the guards occurring in executed transitions and of the function summaries associated to external functions called in those transitions. A drawback of such an approach is of course that the tables might be too incomplete to provide input/output data to follow a given model path, meaning that their corresponding summaries are too restrictive to follow the path. The main contribution of this paper is to define a heuristic to deal with this situation by completing the function tables. The heuristic is based on the computation of new inputs by solving formulas built to avoid duplications in the tables and also to take benefits of the potential function dependencies. The overall approach can be seen as a reachability analysis based on symbolic execution techniques and an heuristic search algorithm used to solve path conditions. The resulting symbolic paths can then be used to generate test cases, in a classical model-based testing approach, for IOSTS extended with function calls. Concrete test inputs can thus be given to an existing system, in order to see if this system reacts according to its IOSTS model. This contribution has potential applications for industrial software testing practices especially integration testing, where units of code (i.e. external functions) must be taken into account while testing the whole system.

The remainder of the paper is organized as follows. Section 2 gives basic concepts about IOSTS and SE. In Section 3, we define function summaries. The main contribution of the paper is given in Section 4: it concerns the resolution of path conditions involving function calls using function summaries. The implementation and experiments of our approach are discussed in Section 5. Finally we conclude the paper with a discussion of related work (Section 6) and some concluding words (Section 7).

2 IOSTS

2.1 Preliminaries

A *data signature* is a pair (S, F) where S is a set of *types* and F a set of *functions* provided with a profile $s_1 \cdots s_n \to s_{n+1}$ on S. For $V = \coprod_{s \in S} V_s$ a set of variables typed in S, the set $T_F(V) = \coprod_{s \in S} T_F(V)_s$ of terms over V is defined as usual. For two sets A and B, B^A denotes the set of mappings $f : A \to B$ from A to B and id_A is the identity mapping on A. For a mapping $f : A \to B$, $f[a_i \mapsto b_i]_{i \in 1..n}$ is the mapping associating b_i to a_i for all i in $1..n$ and $f(a)$ to a not belonging to $\{a_i \mid i \in 1..n\}$.

The set $Sen_F(V)$ of *formulas* is built over Boolean constants $\top$ and $\bot$, equalities $t = t'$ for t and t' terms in $T_F(V)$ of same type and Boolean connectives ($\wedge$, $\vee$, $\neg$). *Substitutions* over V are applications $\sigma : V \to T_F(V)$ that preserve types. Substitutions can be canonically extended to $T_F(V)$.

A *F-model* is a set of typed variables $M = \coprod_{s \in S} M_s$ provided with a function $f^M : M_{s_1} \times \cdots \times M_{s_n} \to M_{s_{n+1}}$ for each $f : s_1 \cdots s_n \to s_{n+1}$ in F. An *interpretation* is an application ν in M^V that preserves types and is canonically extended to $T_F(V)$. The satisfaction of a formula φ in $Sen_F(V)$ by an interpretation $\nu \in M^V$, denoted $M \models_\nu \varphi$, is defined as usual. A formula φ is said satisfiable or feasible if there exists an interpretation ν such that $M \models_\nu \varphi$.

In the sequel, a data signature (S, F) and an F-model M are supposed given. Moreover, when a formula φ is satisfiable and can be handled by a solver, $Sat(\varphi)$ will denote a solution computed by a solver (whatever are the solution or the solver).

2.2 IOSTS

Input Output Symbolic Transition Systems (IOSTS) represent behaviors of reactive systems as sequences of emissions or receptions of values through communication channels conditioned by guards expressed on some attribute values. An *IOSTS-signature* Γ is a couple (A, Ch), where $A = \coprod_{s \in S} A_s$ is a set of typed variables, called *attribute variables*. In the sequel, a variable v of A will be denoted either as a simple identifier (id) or as an identifier (id) and an integer (i) pair, denoted as $id[i]$. The latter case will be useful for dealing with modeling of arrays. Ch is a set of *communication channel names*.

An IOSTS communicates with its environment through actions. The set of *symbolic actions* over $\Gamma = (A, Ch)$, denoted $Act(\Gamma)$, is $I(\Gamma) \cup O(\Gamma) \cup \{\tau\}$ where:

$I(\Gamma) = \{c?(x_1, \ldots, x_n) | \forall i \in 1..n, x_i \in A, c \in Ch, \forall i, j \ (i \neq j \Rightarrow x_i \neq x_j)\}$ is the set of inputs, $O(\Gamma) = \{c!(t_1, \ldots, t_n) | \forall i \in 1..n, t_i \in T_F(A), c \in Ch\}$ is the set of outputs and τ is an internal action. If $n = 0$, (resp $n = 1$), then $c\Delta()$ (resp $c\Delta(t_1)$) with $\Delta \in \{?, !\}$ is simply denoted $c\Delta$ (resp. $c\Delta t_1$).

Values of attribute variables can be modified either by a reception from the environment or by an assignment of a value issued from some internal process.

Definition 1 (IOSTS). *An IOSTS (Q, q_0, Tr) over $\Gamma = (A, Ch)$ is a triple where Q is a set of states, $q_0 \in Q$ is the initial state and $Tr \subseteq Q \times Sen_F(A) \times Act(\Gamma) \times T_F(A)^A \times Q$ is a set of transitions tr of the form (q, ψ, act, ρ, q') where:*

- *q and q' are resp. the source ($source(tr)$) and target state ($target(tr)$) of tr;*
- *$\psi \in Sen_F(A)$ is a* guard*;*
- *$act \in Act(\Gamma)$ is a communication action;*
- *$\rho \in T_F(A)^A$ is a substitution.*

Example 1. Let us illustrate an example of IOSTS inspired from the Sesam-Grids project [12]: Figure 1 describes a simple model of a Microgrid, that is, a system of interconnected and distributed smart components (smartmeters, controller, ...) in connection with energy resources and energy consumption devices. Our example describes a simplified model of a controller inside a Microgrid. Electricity measurements are requested by a smart controller ($mReq!$ action) and sent by a smart meter ($getmeas?Value[cpt]$). N measurements are stored in the array $Value$ whose data can be accessed through variables of the form $Value[i]$ with i integer. Then, they are used by the controller to compute the total electricity consumption via an integral calculation ($I \mapsto INTGR(Value)$) where $INTGR$ is an external function. If the overall consumption is greater than 200, then the function $RISE$ (which is also an external function) is called to compute, according to the returned consumption I, a rate r that is added to the energy price. The new price $((1 + r) * i)$ is sent to the actuator via the (out) channel, together with a message indicating whether the rise is normal ("ok", if $r <= 1$) or should trigger a warning ("$alarm$", if $r > 1$). Otherwise, if the consumption is lower than 200, an acknowledgment is sent ($out!("ok", I)$). In Figure 1, guards of the form $t_1 \leq t_2$, $t_1 < t_2$, $t_1 > t_2$ or $t_1 \geq t_2$ are concise representations of respectively $t_1 \leq t_2 = \top$, $t_1 < t_2 = \top$, $t_1 > t_2 = \top$ or $t_1 \geq t_2 = \top$.

2.3 Symbolic execution of IOSTS

Symbolic execution (SE) consists in executing the $IOSTS$ with symbolic variables rather than numerical values. SE gathers constraints (*path condition*) over such variables that characterize under which circumstances a particular execution path is taken.

To store information concerning an execution, we use a set Fr of so-called fresh variables with $Fr \cap A = \emptyset$ and structures called symbolic states. Fr is an infinite set in which, whenever necessary, it is possible to take a variable not previously used. A *symbolic state* η is a tuple (q, π, λ) where: q (or $q(\eta)$) denotes the state in Q reached after an execution leading to η, $\pi \in Sen_F(Fr)$ (or $\pi(\eta)$)

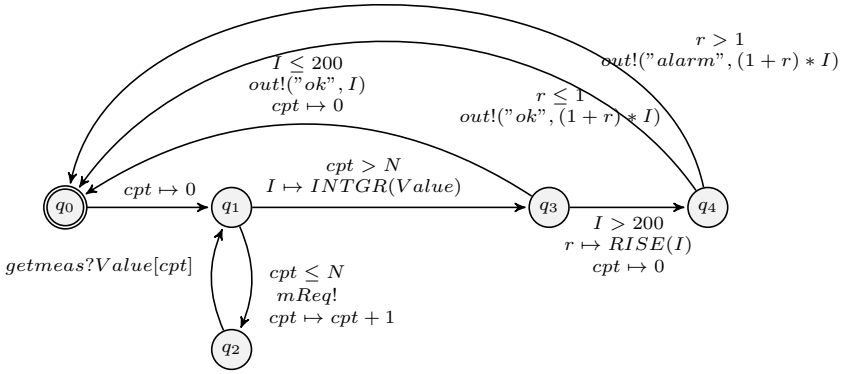

Fig. 1: Micro Grid Controller IOSTS

is the *path condition* that should be satisfied for the execution to reach η and $\lambda : A \to T_F(Fr)$ (or $\lambda(\eta)$) denotes terms over variables in Fr that are assigned to variables of A.

Definition 2 (SE of transitions). *Let $\mathbb{G} = (Q, q_0, Tr)$ be an IOSTS, $tr = (q, \psi, act, \rho, q')$ a transition of Tr and $\eta = (q, \pi, \lambda)$ a symbolic state.*

Let us define λ_i as $\lambda[x_1 \mapsto f_1, \ldots, x_n \mapsto f_n]$ if act is of the form $c?(x_1, \ldots, x_n)$, where each f_i is a fresh variable in Fr and as λ otherwise. The symbolic execution *$SE(tr, \eta)$ of tr from η is the* symbolic transition *$(\eta, \lambda_i(act), \eta')$ where $\eta' = (q', \pi', \lambda')$, with $\pi' = \pi \wedge \lambda(\psi)$, and $\lambda' = \lambda_i \circ \rho$.*

We denote $Fr(SE(tr, \eta))$ the set of all fresh variables of Fr occurring in its definition while not already present in η.

The symbolic execution tree associated with the $IOSTS$ is then defined simply by executing all transitions from all symbolic states.

Definition 3 (SE of $IOSTS$). *Given an IOSTS $\mathbb{G} = (Q, q_0, Tr)$ over $\Gamma = (A, Ch)$, the symbolic execution $SE(\mathbb{G}) = (SS, Init, ST)$ of $\mathbb{G}$ is minimally defined by:*

- *$Init = (q_0, \top, \lambda_0)$ is the initial state. $Init$ is in SS and we have $\forall x \in A$, $\lambda_0(x) \in Fr$, and $\forall x \neq y \in A$, $\lambda_0(x) \neq \lambda_0(y)$. We denote $Fr_0 = \{\lambda_0(x) \mid x \in A\}$ the set of variables used in $Init$.*
- *ST is a set of symbolic transitions such that for any $\eta = (q, \pi, \lambda)$ in SS and for any $tr \in Tr$ of source state q, there exists one symbolic execution $SE(tr, \eta) = (\eta, act_F, \eta')$ of tr from η with $SE(tr, \eta) \in ST$ and $\eta' \in SS$.*

Moreover $Fr(SE(tr, \eta)) \cap Fr_0 = \emptyset$ and for any two distinct symbolic transitions $SE(tr_1, \eta_1)$ and $SE(tr_2, \eta_2)$ in ST, $Fr(SE(tr_1, \eta_1)) \cap Fr(SE(tr_2, \eta_2)) = \emptyset$. For $st = (\eta, act_F, \eta')$ in ST, we denote $source(st) = \eta$ and $target(st) = \eta'$.

Now, we define paths of the symbolic tree resulting from the symbolic execution of an $IOSTS$:

Definition 4 (Symbolic Paths). *The set* $Paths(SE(\mathbb{G}))$ *of* paths *of* $SE(\mathbb{G})$ *is the set of all sequences* $st_1 \cdots st_n$ *with* $\forall i \in 1..n, st_i \in ST$ *such that* $source(st_1) = Init$ *and for any* $j < n$, $q(target(st_j)) = q(source(st_{j+1}))$.

For a path $\delta = st_1 \cdots st_n$ *with* $1 \leq n$, *we note* $End(\delta) = target(st_n)$ *and* $Fr(\delta) = \cup_{i \in 1..n} Fr(target(st_i))$. *By convention,* $End(\varepsilon) = Init$ *and* $Fr(\varepsilon) = \emptyset$.

By construction, path conditions accumulate constraints from all the guards of the IOSTS transitions over a path. Hence, feasibility of path pa is checked by calling a solver over the path condition of the last state of pa. In the sequel, such call will be denoted $Sat(\pi(End(pa))$.

As already mentioned in section 1, external functions such as $INTGR$ and $RISE$ have no built-in interpretations in standard constraint solvers. Thus, when path conditions contain external functions, usual constraint solving techniques cannot be applied directly. We thus propose an improved resolution method to guide the symbolic execution by checking paths feasibility up to some function calls.

3 Function summaries

Instead of either considering external functions as purely abstract by replacing their output with a fresh variable or inserting their code (inlining) at model level quickly leading to combinatorial explosion, our approach allows to control this issue by replacing external functions with *summaries* that offer a partial and extendable view of the function within the model.

In the sequel, P will denote a distinguished subset of F, the set of all external functions.

3.1 Summaries

Values which are used to summarize an external function are taken from its concrete execution on some inputs. This can be represented and saved as a table mapping a finite set of tuples associating input and output data of the function.

Notation 31 *For* $p \in P$ *an external function of profile* $s_1 \ldots s_n \rightarrow s_{n+1}$ *and* $p^M \in M$ *its interpretation, we denote by* tab_p, *a set of tuples* $(v_1, \ldots, v_n, v_{n+1}) \in M_{s_1} \times \cdots \times M_{s_n} \times M_{s_{n+1}}$ *verifying* $p^M(v_1, \ldots, v_n) = v_{n+1}$, *a function table of* p. *If* tab_p *covers the entire domain of* p *then it will fully represent* p^M.

We associate with each external function a summary built on values in the function table. More precisely, a function call will be stored in a given symbolic state under the form

$$(p, (t_1, \cdots, t_n), x)$$

indicating that a call has been performed for the external function p with arguments $(t_1, \cdots, t_n)$ and that its result is stored in the fresh variable x.

We note $Calls(P)$ the set of all function calls.

Definition 5 (Function call summaries). *Let $Cls \subseteq Calls(P)$ be a set of function calls and $Tab = (tab_p)_{p \in P}$, where tab_p is a function table associated to p in P.*
The summary *of Cls up to Tab is:*

$$Sums(Cls, Tab) = \bigwedge_{\substack{cl=(p,(t_1,\cdots,t_n),x) \in Cls, \\ tab_p \in Tab}} Sum(cl, tab_p)$$

where $Sum(cl, tab_p)$ computes the disjunction of different inputs/output tuples of tab_p, i.e.:

$$Sum(cl, tab_p) = \bigvee_{(v_1,\cdots,v_{n+1}) \in tab_p} ((\bigwedge_{i \leq n} t_i = v_i) \wedge x = v_{n+1})$$

with the convention $Sum(cl, \emptyset) = \bot$.

Example 2. Let us go back to the Microgrid example (Figure 1) with the hypothesis $N = 2$. Then, a scenario involves two measurements m_1 and m_2. For illustrative purposes, values that will be used in the summaries associated to the external functions $INTGR$ and $RISE$ are given in Table 1. Based on the values provided by Table 1, we can then compute the corresponding summaries for the set of function calls $Cls = \{cl_1, cl_2\}$ with $cl_1 = (INTGR, ([m_1, m_2]), I)$ and $cl_2 = (RISE, (I), r)$:

- $Sum(cl_1, tab(INTGR))$ is
  $(m_1 = 123 \wedge m_2 = 96 \wedge I = 228) \vee (m_1 = 148 \wedge m_2 = 141 \wedge I = 300)$
- $Sum(cl_2, tab(RISE))$ is
  $(I = 202 \wedge r = 0.7) \vee (I = 300 \wedge r = 2.42)$

Finally, $Sums(Cls, Tab)$ is simply $Sum(cl_1, tab(INTGR)) \wedge Sum(cl_2, tab(RISE))$

$tab(INTGR)$	
[123, 96]	228
[148, 141]	300

$tab(RISE)$	
202	0.7
300	2.42

Table 1: $INTGR$ and $RISE$ tables

3.2 Formula transformation

A path condition φ is built over terms of $T_F(V)$ that may contain occurrences of external functions. Since the satisfiability of φ requires to find an interpretation ν ensuring $M \models_\nu \varphi$, the idea is to proceed in two steps:

1. φ is transformed into a formula with no occurrence of external functions and a set of function calls.

2. the satisfiability of φ is checked with a constraint solver and based on external function summaries (built on concrete executions of called functions).

For a formula φ containing a term of the form $p(t_1, \cdots, t_n)$ with $p \in P$, we will eliminate the occurrence of p in φ by introducing a new fresh variable x in charge of storing the result of the application of p to the terms $t_1, \ldots, t_n$. An interpretation will be acceptable as a solution only if it evaluates terms $t_1, \ldots, t_n$ and x so that they correspond to one of the elements of the function table tab_p. Thus, while eliminating all occurrences of external function in formulas, we also collect all tuples $(t_1, \ldots, t_n, x)$ memorizing all contexts of calls of external functions.

Let us first define a function $\chi : T_F(V) \to T_F(V \cup Fr) \times Calls(P)$ as follows:

- if t is a variable or a constant[3], $\chi(t) = (t, \emptyset)$
- if t is of form $f(t_1, \ldots, t_n)$ with $f \notin P$, then[4]
  $\chi(t) = (f(\chi(t_1)_{|1}, \ldots, \chi(t_n)_{|1}), \cup_{i \in 1..n} \chi(t_i)_{|2})$
- if t is of form $p(t_1, \ldots, t_n)$ with $p \in P$, then
  $\chi(t) = (x, \{(p, (\chi(t_1)_{|1}, \ldots, \chi(t_n)_{|1}), x)\} \cup_{i \in 1..n} \chi(t_i)_{|2})$ with x a fresh variable in Fr.

We assume that all fresh variables introduced by function calls (i.e. the sets $\chi(t)_{|2}$) are pairwise disjoint. Note that the term substitution is performed iteratively by starting with the innermost sub-terms in case of nested function calls. The function χ is canonically extended to formulas by preserving formula structure and accumulating all sets of function calls in a unique set. In the sequel, for a formula φ, $\chi(\varphi)_{|1}$, the formula φ without external functions, will be denoted $\overline{\varphi}$ and $\chi(\varphi)_{|2}$, the set of associated calls, will be denoted $\kappa(\varphi)$.

Example 3. Let us denote φ_0 the formula $RISE(INTGR([x+1, 0])) > 1$ defined over the external functions $INTGR$ and $RISE$ and the variable x.

Then, $\overline{\varphi_0}$ is $r > 1$ while $\kappa(\varphi_0)$ is $\{(RISE, (I), r), (INTGR, ([x + 1, 0]), I)\}$ with I and r new fresh variables.

Now that we are able to remove external functions, we will propose an algorithm to check formula satisfiability.

4 Satisfiability of formulas up to function summaries

Given a formula φ possibly built over external functions, we aim at defining an algorithm that analyzes the satisfiability of φ up to summaries of called external functions. Recall that φ is transformed into a formula $(\overline{\varphi})$ with no occurrence of external functions, while we keep track of external function calls in $\kappa(\varphi)$. We also suppose that we have function tables for these external function calls in Tab.

[3] A constant is a function of profile of the form $\to s$.
[4] For a couple $c = (e, f)$ of elements, $c_{|1}$ is e and $c_{|2}$ is f.

Since by construction $Sums(\kappa(\varphi), Tab)$ is the formula specifying that function calls of κ match with at least a tuple of Tab, if $Sat(\overline{\varphi} \wedge Sums(\kappa(\varphi), Tab))$ returns a solution, then φ is satisfiable (i.e. there exists an interpretation ν such that $M \models_\nu \varphi$). Otherwise, either φ is not satisfiable (i.e. $\forall \nu, M \not\models_\nu \varphi$) or φ is satisfiable but the function tables (used to summarize functions) are not complete enough to exhibit an interpretation ν such that $M \models_\nu \varphi$. Therefore, in the sequel we place ourselves in the hypothesis that function tables can be enriched during the satisfiability search. Furthermore, some heuristics are proposed to help the solver finding a solution when current function summaries are not compatible with the formula, *e.g.* to find additional input data for external functions that could make the formula feasible. We describe these heuristics in more detail below.

4.1 Resolution strategies

NoCorres strategy: $NoCorres$ computes a formula guaranteeing that the inputs of called external functions are different from those already present in the current function tables. This formula will be used later to get new inputs for the concrete executions of external functions, ensuring the enrichment of the function tables, which in turn increases chances to find a solution.

Definition 6 (NoCorres). *Let* $Cls \subseteq Calls(P)$ *be a set of function calls and* $Tab = (tab_p)_{p \in P}$ *a set of tables where* tab_p *is the table associated to* p *in* P. $NoCorres(Cls, Tab)$ *is defined as follows:*

$$NoCorres(Cls, Tab) = \bigvee_{\substack{cl=(p,(t_1,\cdots,t_n),x) \in Cls, \\ tab_p \in Tab}} (NoCorres1(cl, tab_p))$$

with $NoCorres1((p, (t_1, \cdots, t_n), x), tab_p)$ *being defined as follows:*

$$\bigwedge_{(v_1,\cdots,v_{n+1}) \in tab_p} (\bigvee_{i \leq n} t_i \neq v_i)$$

By considering the conjunction of $NoCorres(\kappa(\varphi), Tab)$ and $\overline{\varphi}$, the solver cannot compute interpretations of variables for which all function calls in $\kappa(\varphi)$ are already present in Tab.

Example 4. By applying the above $NoCorres$ definition on the set of function calls and tables given in Example 2, $NoCorres(Cls, Tab)$ is the formula $NoCorres1(cl_1, Tab(INTGR)) \vee NoCorres1(cl_2, Tab(RISE))$ where:

- $NoCorres1(cl_1, tab(INTGR))$ is
  $(m_1 \neq 123 \vee m_2 \neq 96) \wedge (m_1 \neq 148 \vee m_2 \neq 141)$
- $NoCorres1(cl_2, tab(RISE))$ is $(I \neq 202) \wedge (I \neq 300)$

Dependency heuristic: Another way to accelerate and help the solver resolution task to find new inputs, when no solution is found with the current function tables, is to take advantage of direct dependencies between function calls. Indeed when the arguments of a call to an external function f depend on the result of calls to other external functions f_i, we can indicate to the solver to search for solutions in which the inputs of f are compatible with the outputs already present in the tables of the f_i.

Definition 7 (Dep). *Let $Cls \subseteq Calls(P)$ be a set of function calls and $Tab = (tab_p)_{p \in P}$ a set of tables where tab_p is the function table associated to p in P.*

$Dep(Cls, Tab)$ is defined as follows:

$$Dep(Cls, Tab) = \bigwedge_{cl \in Cls} (Dep1(cl, Cls, Tab))$$

with $Dep1(cl, Cls, Tab)$ being defined as follows:

$$\bigwedge_{cl_i = (q, (t'_1, \cdots, t'_n), x') \in Dep_{cl}} (CorresRes(cl_i, tab_q))$$

where:

- *For $cl = (p, (t_1, \cdots, t_n), x)$, $Dep_{cl} = \{cl_i = (q, (t'_1, \cdots, t'_n), x') \in Cls \setminus \{cl\} \mid x' \in \bigcup_{i \leq n} Occ(t_i)\}$ is the set of function calls on which the inputs of cl depend, i.e. the result of these function calls occurs in one or several input terms of cl.*
- *For $cl' = (q, (t'_1, \cdots, t'_n), x')$, $CorresRes(cl', tab_q)$ is the formula*

$$\bigvee_{(v_1, \cdots, v_{n+1}) \in tab_q} (x' = v_{n+1})$$

$CorresRes(cl', tab_q)$ extracts values from the tables of function calls on which cl' depends.

Example 5. Let us consider again the function calls introduced in Example 2. We have:

- $Dep_{cl_2} = \{cl_1\}$ with $cl_1 = (INTGR, ([m_1, m_2]), I)$, because I (the variable storing the result of cl_1) is an input of $cl_2 = (RISE, (I), r)$.
- $CorresRes(cl_1, tab(INTGR))$ is $(I = 228 \vee I = 300)$

Therefore, to find new inputs for $RISE$, we can give to the solver $Dep(cl_2, Cls)$, that is $CorresRes(cl_1, tab(INTGR))$ to exploit the values of I already existing in $Tab(INTGR)$.

By combining both *NoCorres* and *Dep* heuristics, it is possible to generate new sets of inputs, i.e. new rows in the tables of external functions that are relevant for the overall path constraint resolution. Indeed, *NoCorres* ensures that we will add at least one new row to a function table, while *Dep* takes care of avoiding solutions that cannot be satisfied by the current summaries of the callers, and are thus useless for the current solving process.

4.2 SolveTab algorithm

Algorithm 1 describes our dynamic solving procedure $SolveTab(m, \overline{\varphi}, \kappa(\varphi), Tab)$. The goal of $SolveTab$ is to analyze the satisfiability up to some function tables Tab of a formula ($\overline{\varphi}$) with no occurrences of external functions, provided with a set of function calls ($\kappa(\varphi)$). The integer parameter m is used to bound the number of attempts that are made to increase the size of function tables. As explained in the previous section, we resort to this mechanism when no tuple in a function table is compatible with φ. However, since there is no guarantee that extending function tables will eventually lead to find a solution for $\overline{\varphi} \wedge Sums(\kappa(\varphi), Tab)$, we cut the search after at most m steps. If no solution is found beyond m or no possible function inputs are found, then we can deduce that the formula is unsatisfiable up to function tables. The case $m = 0$ corresponds to a purely static version where a solution is searched without modifying function tables (i.e. Tab is then a static variable).

Algorithm 1: $SolveTab(m, \psi, \kappa, Tab)$

Data: m: an integer, ψ: a formula without black-box functions, κ : a set of function calls, Tab: function tables
Result: satisfiability of ψ up to κ; Tab: updated function tables

```
1  begin
2      sol ← Sat(ψ ∧ Sums(κ, Tab));
3      TabModified ← True;
4      while TabModified and sol = NONE and m ≠ 0 do
5          In ← Sat(ψ ∧ NoCorres(κ, Tab) ∧ Dep(κ, Tab))   /* Search for new
           solutions while considering dependencies between function
           calls */ ;
6          if In = NONE then        /* no possible inputs for function */
7              In ← Sat(ψ ∧ NoCorres(κ, Tab))   /* Search for new possible
               solutions without considering function dependencies */ ;
8          if In ≠ NONE then      /* new possible inputs for function */
9              for each call cl = (p, (t1, ··· , tn), x) in κ do
10                 VIns ← Ins(In, cl)  /* extract inputs for p from In */;
11                 r ← Exec_p(VIns)      /* concrete execution of p */;
12                 Tab(p) ← (VIns, r).Tab(p)   /* save new tuple in Tab */;
13             sol ← Sat(ψ ∧ Sums(κ, Tab));
14         else
15             TabModified ← False;
16         m ← m − 1;
17     return (sol, Tab);
```

Let us now give some few comments on Algorithm 1. By hypothesis, arguments $(\psi, \kappa) = \chi(\varphi)$ for some φ, so that ψ and κ share external functions and variables.

If $Sat(\psi \wedge Sums(\kappa, Tab))$ gives back a result sol, then sol directly provides an interpretation of variables that satisfies the formula φ in which external functions are interpreted according to values given in their associated tables Tab. In other words, by construction $M \models_{sol} \varphi$.

In case no solution is found with $Sat(\psi \wedge Sums(\kappa, Tab))$ in Line 2 (ψ is unsatisfiable up to current function summaries), then we try to find additional input data for external functions that could enrich Tab (and function summaries) and make the formula satisfiable. In order to guide the solver to find new inputs, Dep and $NoCorres$ heuristics can be used. $NoCorres$ is necessary to compute new inputs distinct from those already present in current tables (that failed to provide us with a solution). On the other hand, Dep is an alternative option that can facilitate the search for inputs that are compatible with dependencies among function calls. However, this might restrict the search space too much, as we may need to add a new output to the table of function f to find a suitable input for g that depends on f. Therefore, if adding the Dep condition results in an unsatisfiable formula (Line 7), we call the solver again with only the formula and the $NoCorres$ condition. Indeed, the new tuples might give us new outputs that will in turn make Dep satisfiable at next step, or even give directly a solution to the original problem.

If an interpretation (variable In) is found, then lines 10, 11 and 12 allow us to add new tuples in function tables (variable Tab), by successively extracting input values $VIns \leftarrow ins(In, cl)$ for each function call (line 10), concretely executing the function (line 11) with $VIns$ as input data, ($Exec_p$ being the reference implementation of p), to obtain the corresponding output (variable r) and finally storing new tuples in Tab_p (line 12). Then, the solver is called (line 13) to check the satisfiability of ψ, with the new summaries based on the enriched function tables.

Last, let us point out that a bigger number m of allowed attempts to search for new tuples enriching function tables gives more chances to find a solution, but will possibly take longer to compute.

4.3 Discussion: *SolveTab* for symbolic execution

Since path conditions are formulas built over terms that may contain occurrences of external functions, we proceed as detailed in the previous section in order to check symbolic paths feasibility according to function summaries. For that, we use the $SolveTab$ solving Algorithm 1 based on concrete executions of called functions to guide the symbolic execution of $IOSTS$. Therefore, we need to extend symbolic states by accumulating function calls. An extended symbolic state is of the form $\eta = (q, \pi, \lambda, \kappa)$, where κ (or $\kappa(\eta)$) denotes the sequence of function calls of the form $(p, (t_1, \cdots, t_n), x)$. By construction, the path condition π does not contain occurrences of external functions.

A symbolic state $(q, \pi, \lambda, \kappa)$ is satisfiable according to function summaries only if there exists an integer m and function tables Tab such that the result (sol, Tab) provided by $SolveTab(m, \pi, \kappa, Tab)$ defines an interpretation sol of fresh variables ensuring the interpretation of π as true.

The symbolic execution in accordance with function tables is based on the $SolveTab$ resolution algorithm which takes as arguments, in addition to the path condition and the set of accumulated function calls, the parameter m and the set of tables Tab. If we note $SE_{Tab,m}(\mathbb{G})$ the symbolic execution of $\mathbb{G}$ using $SolveTab$ for building satisfiable symbolic states, then $SE_{Tab,m}(\mathbb{G})$ is highly dependent on the initial tables (Tab) specifying called functions, the chosen threshold of resolution attempts m, the choice of the solver and the strategy of traversal (in depth, in width, ...) of the symbolic tree. Thus, the construction of $SE_{Tab,m}(\mathbb{G})$ will more or less approach the ideal symbolic tree $SE(\mathbb{G})$. If elements of Tab are representative enough of the behavior of called functions, with regard to the solicitations of the $IOSTS$ or if m is large enough to find a solution for each possible path of the $IOSTS$ (by enriching function tables), then $SE_{Tab}(\mathbb{G})$ becomes closer to $SE(\mathbb{G})$ and more states become reachable.

5 Implementation and Experiments

5.1 Implementation

DIVERSITY tool: DIVERSITY [3, 14], is a multi-purpose and customizable platform for formal analysis based on symbolic execution (test generation, proof, deadlock search, etc.) that is on its way to becoming an Eclipse open-source project[5]. DIVERSITY generates a symbolic tree (for a fixed maximal height) by simulating the system specification with input symbols rather than concrete values for data. Test inputs are computed by solving the path conditions. For that purpose, DIVERSITY integrates solvers such as $Z3$[6], and $CVC4$[7].

We have implemented the $SolveTab$ algorithm 1 and the heuristics described in Section 4 as additional Formal Analysis Module (FAM) for checking satisfiability of path conditions within DIVERSITY. By default, constraints involving external functions are left uninterpreted (i.e. out of the scope of solvers). In addition, variable assignments within an $IOSTS$ transition contain only basic operations. Now, all such external functions can be handled with the newly implemented technique, using values recorded in function tables. The $SolveTab$ algorithm allows to know whether a new execution context EC (a symbolic state) can be built in the symbolic execution tree in accordance with called functions, after the execution of a transition in the $IOSTS$. In other words, it checks if a transition of the $IOSTS$ could be fired or not according to called functions, with a possible enrichment of the function tables. Therefore, thanks to $SolveTab$ we ensure the symbolic tree's construction with only feasible paths compatible with called functions while ensuring a high transition coverage of the $IOSTS$ transitions.

[5] `https://projects.eclipse.org/proposals/eclipse-formal-modeling-project`
[6] `https://z3.codeplex.com/`
[7] `http://cvc4.cs.nyu.edu/web/`

Microgrids case study: Here, we discuss the application of our technique to the Microgrid example 1 that uses external function calls ($INTGR$ and $RISE$). For that, we present the resulting symbolic trees (to a depth of 6) by applying the $SolveTab$ algorithm with the initial function tables 1 and various values of m, the number of attempts made to increase the function tables. At first, we use the static version (static tables, $m = 0$). Result is shown in the left tree of Figure 2. Then we progressively increase m until 196, function tables are enriched by concretely executing functions with new inputs found thanks to NoCorres and Dependency heuristics. As expected, this permits to cover more transitions of the model, reach more symbolic states and explore more feasible paths. Indeed the size of the generated tree grows with the value of m. The middle tree of figure 2 is obtained with $m = 1$, while the one on the right is the result of $m = 196$, where all transitions of the $IOSTS$ are covered.

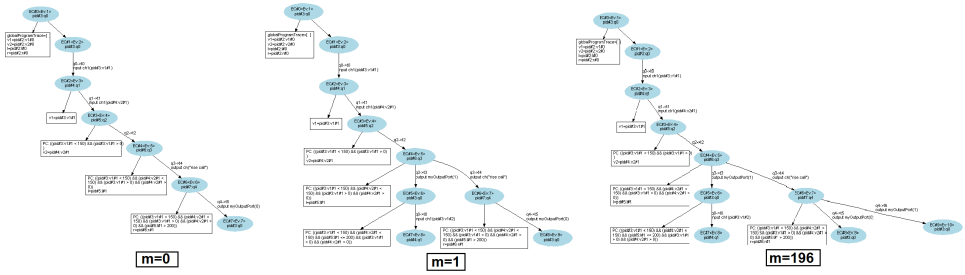

Fig. 2: Symbolic execution tree in DIVERSITY

5.2 Experiments

A summary of the main outcomes of the application of our technique to the Microgrid example is provided in Table 2. It should be noted that our results are based on the initial function tables given in table 1 and that we dispose of the functions' implementation in order to be able to execute them with appropriate inputs in an unitary setting. For each experiment (i.e. each line), we fix "m" (maximum number of attempts for the $SolveTab$ algorithm) and the maximum "$height$" allowed for the symbolic tree. Then we record results concerning the enriched tables (number of tuples) and the symbolic tree computation: the number of reachable states, the achieved $IOSTS$ transitions coverage and the execution time. Again, when $m = 196$ all possible symbolic states are covered, and there is no need to try to further enrich the function tables.

The resulting coverage rate and execution time depend on the initial function tables. In fact, the more the elements of called function tables are representative of the function behaviors and compatible with path conditions, the less attempts are needed to achieve the same level of coverage. For instance, if we now start with the initial function tables of table 3 for the Microgrid example, all transitions of the specification are covered (in $2s759ms$) without doing any concrete

Micro-Grid IOSTS: initial Tab (Tables 1)						
"m"	**height**	**Enriched Tab**		**states**	**transition coverage**	**time**
		INTGR	**RISE**			
0	15	2	2	16	62.5%	638ms
1	15	8	3	27	75%	2s
196	**15**	**198**	**3**	**120**	**100%**	**4m15s**

Table 2: Experiments with initial Tables 1

$tab(INTGR)$	
[0, 0]	0
[12, 18]	30
[123, 96]	228
[148, 141]	300

$tab(RISE)$	
202	0.7
228	0.97
300	2.42

Table 3: Other initial Tables

execution ($m = 0$), as all possible outcomes are captured by one row of the tables.

Micro-Grid IOSTS: other initial Tab (Tables 3)						
"m"	**height**	**Enriched Tab**		**states**	**coverage**	**time**
		INTGR	**RISE**			
0	**15**	**4**	**3**	**120**	**100%**	**2s759ms**

Table 4: Experiments with other initial Tables 3

6 Related Work

IOSTS and symbolic execution have been used in many works [6, 10, 17] for different purposes. Until now standard solvers were not capable of dealing with functions occurring in path conditions. The usage of symbolic execution and path feasibility analysis are studied in [7,35] but this is limited to the analysis of functions themselves and does not take into consideration the impact of function calls on the feasibility of the whole system.

Our work borrows the idea of mixing symbolic execution with information obtained from instrumented concrete executions, from *concolic testing*, a method that has been implemented in various settings, such as [9,18,32]. However, these frameworks are primarily directed at source code level in the objective of unit testing. Another concolic tool, PathCrawler [36] proposed an approach to encompass function calls [25], using pre/post-condition couples as a specification.

Other unit testing frameworks traditionally use stubs [26], which are built manually in an ad-hoc way, to replace external functions for which no implementation is available. Also, automatically-generated software stubs [19] are used for

abstracting (over-approximating) lower-level functions during dynamic test generation [16] based on code.

[1,4,20,22], propose different techniques to conduct symbolic execution with simplified summary representations. But these techniques do not include heuristics similar to the ones (NoCorres, Dependency) we introduce in our paper.

Other techniques have been proposed to deal with constraints generated from symbolic code execution [2,30]. These techniques fall back on concrete values and use randomization to help simplify complex constraints that cannot be solved directly. Our approach based on IOSTS models is orthogonal to these code-based approaches and uses some heuristic search to solve path conditions including external function calls.

In our previous work [8], like in the work [25], we specify external functions by means of contracts (instead of tables): we unfolded the symbolic tree of an IOSTS by replacing each transition including an external function call by as many transitions as there are behaviors in the contract associated to the considered external function.

7 Conclusion and future work

In this work, we use the IOSTS framework extended with external functions. We adapt existing symbolic execution techniques over IOSTS to deal with such function calls, using function tables to summarize them with a formula. The construction of a symbolic execution tree is based on an algorithm $SolveTab$ that permits to check path conditions satisfiability up to function summaries with the possibility of enriching function tables by concretely executing them, in order to achieve a high transition coverage of the $IOSTS$.

The proposed approach has been implemented within the DIVERSITY symbolic execution tool. The results show that symbolic execution coupled with function summaries is a practical and effective approach to take into consideration external functions for IOSTS models analysis mainly for model-based testing which is a significant motivation for our work. Nevertheless, it could probably be applied to other families of transition systems like those in [31,33,34].

References

1. Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 367–381, Berlin, Heidelberg, 2008. Springer-Verlag.
2. Saswat Anand, Corina S. Păsăreanu, and Willem Visser. Jpf-se: A symbolic execution extension to java pathfinder. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 134–138, Berlin, Heidelberg, 2007. Springer-Verlag.
3. Mathilde Arnaud, Boutheina Bannour, and Arnault Lapitre. An illustrative use case of the diversity platform based on uml interaction scenarios. *Electron. Notes Theor. Comput. Sci.*, pages 21–34, 2016.

4. Domagoj Babic and Alan J. Hu. Structural abstraction of software verification conditions. In *Proceedings of the 19th International Conference on Computer Aided Verification*, pages 366–378, Berlin, Heidelberg, 2007. Springer-Verlag.
5. Boutheina Bannour, Jose Pablo Escobedo, Christophe Gaston, and Pascale Le Gall. Off-line test case generation for timed symbolic model-based conformance testing. In *Testing Software and Systems*, pages 119–135. Springer, 2012.
6. Boutheina Bannour, Christophe Gaston, Marc Aiguier, and Arnault Lapitre. Results for compositional timed testing. In *20th Asia-Pacific Software Engineering Conference, APSEC*, pages 559–564. IEEE Computer Society, 2013.
7. N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS*, volume 5505 of *LNCS*. Springer, 2009.
8. Imen Boudhiba, Christophe Gaston, Pascale Le Gall, and Virgile Prevosto. Model-based testing from input output symbolic transition systems enriched by program calls and contracts. In *Testing Software and Systems - 27th IFIP WG 6.1 International Conference, ICTSS*, volume 9447 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2015.
9. Cristian Cadar and Dawson Engler. Execution generated test cases: How to make systems code crash itself. *Software Model Checking (SPIN)*, 2005.
10. James C.King. Symbolic execution and program testing. *Communication ACM*, 19:385–394, 1976.
11. Lori A Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, pages 215–222, 1976.
12. The consortium Sesam-Grids. The Sesam-Grids Project. In http://www.sesam-grids.org/.
13. CREST. `https://code.google.com/archive/p/crest/`. Accessed: 2017-03-04.
14. J. Deltour, A. Faivre, E. Gaudin, and A. Lapitre. Model-based testing: An approach with SDL/RTDS and DIVERSITY. In *SAM*, volume 8769 of *LNCS*. Springer, 2014.
15. DIVERSITY. `https://projects.eclipse.org/proposals/eclipse-formal-modeling-project`. Accessed: 2017-03-04.
16. Dawson Engler and Daniel Dunbar. Under-constrained execution: Making automatic code destruction easy and scalable. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 1–4, New York, NY, USA, 2007. ACM.
17. Christophe Gaston, Pascale Le Gall, Nicolas Rapin, and Assia Touil. Symbolic execution techniques for test purpose definition. In *Testing of Communicating Systems*, pages 1–18. Springer, 2006.
18. Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. *PLDI 05 Programming Language Design and Implementation*, pages 213–223, 2005.
19. Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, pages 213–223, 2005.
20. Denis Gopan and Thomas Reps. Low-level library analysis and summarization. In *Proceedings of the 19th International Conference on Computer Aided Verification*, pages 68–81, Berlin, Heidelberg, 2007. Springer-Verlag.
21. jCUTE. `http://osl.cs.illinois.edu/software/jcute/`. Accessed: 2017-03-04.
22. Sarfraz Khurshid and Yuk Lai Suen. Generalizing symbolic execution to library classes. *SIGSOFT Softw. Eng. Notes*, pages 103–110, 2005.
23. Nicolas Kicillof, Wolfgang Grieskamp, Nikolai Tillmann, and Victor Braberman. Achieving both model and code coverage with automated gray-box testing. In *Advances in Model-Based Testing (A-MOST)*. ACM, 2007.

24. MathWorks. The Simulink documentation. In http://fr.mathworks.com/help/simulink/.
25. Patricia Mouy, Bruno Marre, Nicky Willams, and Pascale Le Gall. Generation of all-paths unit test with function calls. *International Conference On Software Testing (ICST)*, pages 32 –41, 2008.
26. Glenford J. Myers, Corey Sandlers, and Tom Badgett. *The Art of Software Testing*. John Wiley & Sons, 3 edition, 2011.
27. Object Management Group. The UML standard specification. In http://www.omg.org/spec/UML/2.4.1/.
28. PathCrawler. `http://frama-c.com/pathcrawler.html`. Accessed: 2017-03-04.
29. Symbolic PathFinder. `http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc`. Accessed: 2017-03-04.
30. Corina S. Păsăreanu, Neha Rungta, and Willem Visser. Symbolic execution with mixed concrete-symbolic solving. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 34–44, New York, NY, USA, 2011. ACM.
31. J. J.M.M. Rutten. A calculus of transition systems (towards universal coalgebra). Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, The Netherlands, 1995.
32. Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. *SIGSOFT Softw. Eng. Notes*, 30:263–272, 2005.
33. Jan Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer networks and ISDN systems*, 29(1):49–79, 1996.
34. Jan Tretmans. Model based testing with labelled transition systems. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, pages 1–38. Springer-Verlag, Berlin, Heidelberg, 2008.
35. Y. Wang, Y. Xing, and X. Zhang. A method of path feasibility judgment based on symbolic execution and range analysis. *International Journal of Future Generation Communication and Networking*, 2014.
36. Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. PathCrawler: automatic generation of path tests by combining static and dynamic analysis. *European Dependable Computing Conference*, pages 281–292, 2005.