Model-Based Testing from Input Output Symbolic Transition Systems Enriched by Program Calls and Contracts

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

[1] Laboratoire MAS, CentraleSupelec, 92195 Châtenay-Malabry, France
{imen.boudhiba,pascale.legall}@centralesupelec.fr
[2] CEA LIST, Point Courrier 174, 91191 Gif-sur-Yvette, France
{virgile.prevosto,christophe.gaston}@cea.fr

Abstract. An Input Output Symbolic Transition System (IOSTS) specifies all expected sequences of input and output messages of a reactive system. Symbolic execution over this IOSTS then allows to generate a set of test cases that can exercise the various possible behaviors of the system it represents. In this paper, we extend the IOSTS framework with explicit program calls, possibly equipped with contracts specifying what the program is supposed to do. This approach bridges the gap between a model-based approach in which user-defined programs are abstracted away and a code-based approach in which small pieces of code are separately considered regardless of the way they are combined. First, we extend symbolic execution techniques for IOSTS with programs, in order to re-use classical test case generation algorithms. Second, we explore how constraints coming from IOSTS symbolic execution can be used to infer contracts for programs used in the IOSTS.

Keywords: Input output symbolic transition systems · Program contracts · Model-based testing · Symbolic execution · Feasibility

1 Introduction

Symbolic transition systems, such as Input Output Symbolic Transition Systems (IOSTS) [11] are a classical reference modeling framework for model-based testing of reactive systems. They provide a convenient abstraction of the behaviors of such systems by modeling system state evolution using variable assignments. The symbolic execution tree of an IOSTS characterizes the different classes of numeric executions. Each path defines a sequence of symbolic inputs and outputs, and a path condition which is a formula constraining the values exchanged (inputs or outputs) with the environment of the system. It is possible to use

Work described in this paper has been partially funded by the ITEA project OpenETCS and the BGLE project Sesam-Grids.

K. El-Fakih et al. (Eds.): ICTSS 2015, LNCS 9447, pp. 35–51, 2015.
DOI: 10.1007/978-3-319-25945-1_3

such paths as reference symbolic behaviors to be tested (i.e. as *test purposes*). In [11], we have proposed a framework to analyze IOSTS both to extract such test purposes and to solve the oracle problem thanks to a fully on-line algorithm. However, this kind of framework is limited by the symbolic treatment of functions. Indeed, IOSTS variables are assigned by terms built on functions. In order to be able to reason on the symbolic values assigned to variables, the symbolic execution engine is equipped with constraint solving techniques able to analyze those functions. As long as one deals with basic arithmetic or boolean functions, it is generally tractable, but as soon as one deals with user-defined or ad-hoc functions, solving techniques may fail to scale, or even, due to undecidability results, such techniques may not exist. Analyzing such functions (later referred as "programs") may require both to deal with sophisticated data structures and to explore their (arbitrarily complex) control graph.

In this paper we propose an approach to overcome this limitation by abstracting program behaviors by means of *contracts* [18]. A contract for a program consists in a collection of couples, also called *behaviors*, formed of a pre-condition that specifies constraints that the caller must enforce at the call site, and a post-condition which is a property guaranteed at the program return. We enrich the basic IOSTS framework to deal with program calls equipped with contracts. We show how to extend symbolic execution mechanisms to reason about IOSTS equipped with program calls by analyzing those calls through their contracts. Thus, we avoid analyzing the actual behavior of the program and replace it by abstract constraints on its formal parameters. Our framework allows computing symbolic paths that can be used as test purposes. It may happen that guards and contracts are incompatible so that some symbolic paths are infeasible (*i.e.* they have no associated trace). In practice it means that there exists no program that can both satisfy its associated contracts and compute values allowing to follow the whole symbolic path. We show how to use symbolic techniques to check that a given set of symbolic paths is consistent with respect to program calls.

Moreover, since guards occurring on transitions of an IOSTS interact with contracts associated to programs, we present an approach to extract new contracts for each of the program exercised. Such contracts reflect constraints on the program that make the path feasible. As such, they represent new contracts that can be used at the unitary level, to evaluate the correctness of actual program used to implement the system under test.

The remaining of the paper is organized as follows. In Sect. 2, we give basic definitions about many-typed first order logic. Section 3 presents programs and their contracts. In Sect. 4, we introduce IOSTS with programs. Section 5 defines symbolic execution of an IOSTS with programs and the associated feasibility condition. Finally, usage of symbolic execution for testing purposes, including contract inference for unitary testing is introduced in Sect. 6.

2 Preliminaries

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 $\{a_i \mid i \in 1..n\}$. By convention, $[a_i \mapsto b_i]_{i\in 1..n}$ stands for $id_A[a_i \mapsto b_i]_{i\in 1..n}$. For two mappings $f : A_1 \to B$ and $g : A_2 \to B$ with $A_1 \cap A_2 = \emptyset$, $f \cup g : A_1 \cup A_2 \to B$ is the mapping defined by: $\forall a \in A_1, (f \cup g)(a) = f(a)$ and $\forall a \in A_2, (f \cup g)(a) = g(a)$. A^* (resp. A^+) denotes the set of words on A provided with the concatenation operator '.' and the empty word ε (resp. deprived of the empty word ε). For an ordered list $l = (a_1, \ldots, a_n)$ of n elements of A, $\{\!\{l\}\!\}$ denotes the set $\{a_1, \ldots, a_n\}$ of elements occurring in l.

We use classical multi-typed first order logic to handle data. A *data signature* is a pair (S, F) where S is a set of so-called *types* and F is a set of *functions* provided with a profile $s_1 ... s_{n-1} \to s_n$ with each $s_i \in S$. 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 so-called functional terms over V is defined as usual over (S, F). Moreover, each set V_s contains an identified subset, denoted V_s^{fro}, whose elements are called *frozen variables* and we denote $V^{fro} = \coprod_{s\in S} V_s^{fro}$ the subset of V of all frozen variables. 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 usual Boolean connectives ($\wedge$, $\vee$, $\neg$, ...). *Substitutions* over V are applications $\sigma : V \to T_F(V)$ that preserve types and are such that all elements of V^{fro} are frozen for σ (i.e. $\forall v \in V^{fro}, \sigma(v) = v$). Thus, as frozen variables cannot be substituted, they may be considered as new special constants. Substitutions can be canonically extended to $T_F(V)$. For a term t in $T_F(V)$, for a formula φ in $Sen_F(V)$, $Occ(t)$ and $Occ(\varphi)$ will denote the set of variables occurring in respectively t and φ.

A *F-model* is a set of typed variables $M = \coprod_{s\in S} M_s$ provided with a function $\overline{f} : M_{s_1} \times \cdots \times M_{s_{n-1}} \to M_{s_n}$ for each $f : s_1 \cdots s_{n-1} \to s_n$ in F. An *interpretation* is an application $\nu : M^V$ that preserves types and can be 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 by considering the meaning of the equality predicate, Boolean constants and connectives. A formula φ in $Sen_F(V)$ is valid if and only if for all interpretations $\nu : V \to M$, $M \models_\nu \varphi$. In the sequel, data signature (S, F) and F-model M are supposed given.

3 Programs and Contracts

Programs. User-defined functions, called *programs*, are identifiers provided with an interface specifying their formal parameters used to store input and output data. We only consider here programs with no side effect and one output variable.

Definition 1 (Program). *Let $X = \coprod_{s\in S} X_s$ be a set of typed variables. A program over X is an identifier p provided with:*

- *a list $InOut(p) = (x_1, \cdots, x_{n+1}) \in X^{n+1}$, called the* interface *of p, with $n \geq 1$ and $\forall i \neq j, x_i \neq x_j$. $In(p)$ (resp. $Out(p)$) denotes the list $(x_1 \cdots x_n)$ (resp. (x_{n+1})) of* input (resp. output) formal parameters *of p.*
- *and a mapping $Sem : M^{\{\!\{In(p)\}\!\}} \to M^{\{\!\{InOut(p)\}\!\}}$, called the* semantics *of p, verifying the so-called* semantic condition*:*
  $\forall \nu \in M^{\{\!\{In(p)\}\!\}}, \forall x_j \in \{\!\{In(p)\}\!\}, Sem(\nu)(x_j) = \nu(x_j)$.

Depending on the values associated to $In(p)$ through the interpretation ν, Sem associates a value to the formal parameter x_{n+1} in $Out(p)$. The semantic condition ensures that a program call has no effect on its input formal parameters. By extrapolation, given a list $l = (x_1, \cdots, x_{n+1})$, $In(l)$ and $Out(l)$ will resp. denote $(x_1, \cdots, x_n)$ and (x_{n+1}).

A *signature* Σ is a tuple (S, F, X, P) where (S, F) is a data signature and P is a set of programs defined over the set of typed variables X.

Let $V = \coprod_{s \in S} V$ be a set of typed variables. The set $T_\Sigma(V) = \coprod_{s \in S} T_\Sigma(V)_s$ of *typed terms* over V contains:

- all functional terms of $T_F(V)$
- all elements $p(t_1, \cdots, t_n)$ with $p \in P$ of interface $(x_1, \cdots, x_{n+1})$, $\forall 1 \leq i \leq n, x_i \in X_{s_i}$, and $t_i \in T_F(V)_{s_i}$. If $x_{n+1} \in V_s$, $p(t_1, \cdots, t_n) \in T_\Sigma(V)_s$.

Any interpretation $\nu : V \to M$ can be canonically extended on $T_\Sigma(V)$ as follows: for any program p in P defined by its interface $(x_1 \cdots x_{n+1})$ and its semantics Sem_p, let us consider $\mu^p_\nu : \{\!|In(p)|\!\} \to M$ an interpretation such that $\forall 1 \leq i \leq n,\ \mu^p_\nu(x_i) = \nu(t_i)$, we have $\nu(p(t_1, \cdots, t_n)) = Sem_p(\mu^p_\nu)(x_{n+1})$.

Contracts. Contracts specify what programs are expected to compute, as opposed to how they compute their result. They have been introduced in the pioneering work of Floyd [10] and Hoare [12], and form a key ingredient of the Eiffel programming language [18]. In short, a contract describes what a program requires from its caller (the pre-condition) and what it guarantees when it returns (the post-condition). We use here a slightly refined notion where a contract can be split in a set of behaviors [2,5]. In this setting, pre-condition of a behavior indicates a possible case in which the program may be executed. As before, when a behavior is active, its post-condition must hold at the end of the execution.

Most of the times, pre and post conditions of a program are simply formulas in resp. $Sen_F(\{\!|In(p)|\!\})$ and $Sen_F(\{\!|InOut(p)|\!\})$. However, contracts can involve other variables representing the global state of the system. The latter will be frozen variables whose associated values are conditioned by axioms and cannot be modified. These variables will be useful for inferring contracts from symbolic execution tree, as shown in Sect. 6.2.

Definition 2 (Program Contract). *Let $l = (x_1, \ldots, x_{n+1})$ be a list of variables with $\forall i \leq n+1, x_i \in X$. Let W be a subset of frozen variables verifying $X \cap W = \emptyset$. A program contract for l and W is a set:*

$$\{(Pre_1, Post_1), \ldots, (Pre_k, Post_k)\}$$

such that $\forall\ i \leq k$, $Pre_i \in Sen_F(\{\!|In(l)|\!\} \cup W)$ and $Post_i \in Sen_F(\{\!|l|\!\} \cup W)$.

A program contract is said to be:

- disjoint *if for all $i, j \leq k$ with $i \neq j$, the formula $\neg(Pre_i \wedge Pre_j)$ is valid.*
- complete *if the formula $\bigvee_{i \leq k} Pre_i$ is valid.*

Disjointness requires that at most one behavior of the contract is applicable for any considered input data, *i.e.* the pre-conditions are mutually exclusive. For simplicity purpose, we only consider disjoint contracts in this paper. Completeness indicates that for any input at least one behavior is applicable. In practice, programs are often partially defined over their input domain. We thus allow incomplete contracts, rejecting input data outside the scope of preconditions.

Example 1. Let us consider a program *Price* of interface (x_1, x_2) where x_1 is of type *Drink*, an enumerated type with two values $\{0, 1\}$ and x_2 is of type *Integer*. x_1 is the input parameter indicating the selected beverage and x_2 is the output parameter corresponding to its price. An example of contract for *Price* is $C_r = \{(Pre_1, Post_1), (Pre_2, Post_2)\}$ (both disjoint and complete), with:

- $Pre_1 : \ x_1 = 0$, $Post_1 : \ x_2 \geq 100 \wedge x_2 \leq 200$
- $Pre_2 : \ x_1 = 1$, $Post_2 : \ x_2 \geq 200 \wedge x_2 \leq 300$

Definition 3 (Contract Satisfaction). *Let $l = (x_1, \cdots, x_{n+1})$ be an interface, W a set of frozen variables provided with $Ax \subseteq Sen_F(W)$ and C a contract for l and W. Let us consider an interpretation $\nu \in M^W$ such that $M \models_\nu Ax$ and a mapping $Sem : M^{[\![In(l)]\!]} \to M^{[\![l]\!]}$ satisfying the semantic condition.*

Sem satisfies C up to ν, denoted $Sem \models_\nu C$, if and only if:

$$\forall (Pre, Post) \in C, \forall \mu \in M^{[\![In(l)]\!]}, M \models_{\nu \cup \mu} Pre \Rightarrow M \models_{\nu \cup Sem(\mu)} Post$$

$Sem_\nu(C) = \{Sem : M^{[\![In(l)]\!]} \to M^{[\![l]\!]} \mid Sem \models_\nu C\}$ denotes the set of semantics satisfying C up to ν.

For each interface l, we consider the trivial contract $C_{\emptyset,l} = \{\}$, simply denoted $C_\emptyset$, defined on l that does not restrict behaviors of programs, that is $p \in Sem(C_\emptyset)$ for all programs p of interface l. Similarly, we consider the contract $C_{\top,l} = \{(\top, \top)\}$, simply denoted $C_\top$, defined on l that requires that the program is defined for every well-typed input data tuple.

Given a signature $\Sigma = (S, F, X, P)$, a set of frozen variables W with its set of axioms $Ax \subseteq Sen_F(W)$, and an interpretation $\nu \in M^W$ verifying $M \models_\nu Ax$, we consider families $\mathbb{C} = (C_p)_{p \in P}$ of contracts indexed by P, in particular $\mathbb{C}_\emptyset = (C_\emptyset)_{p \in P}$ and $\mathbb{C}_\top = (C_\top)_{p \in P}$. $Mod_\nu(\mathbb{C})$ is the set of all families $\mathcal{S}em = (Sem_p)_{p \in P}$ such that $\forall p \in P, Sem_p \models_\nu C_p$. $\mathcal{S}em$ is then called a *P-model*.

4 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 types variables, called *attribute variables*, such that for all s in S, $A_s \cap X_s = \emptyset$ and where Ch is a set of *communication channel names*.

An IOSTS communicates with its environment through communication actions. The set of *symbolic actions* over Γ, denoted $Act(\Gamma)$, is $I(\Gamma) \cup O(\Gamma) \cup \{\tau\}$ where: $I(\Gamma) = \{c?x | x \in A, c \in Ch\}$ is the set of inputs, $O(\Gamma) = \{c!t | t \in T_\Sigma(A), c \in Ch\}$ is the set of outputs and τ is an internal action.

Values of attribute variables can be modified in two ways: by receiving a value from the environment or by assigning a value from some internal process.

Definition 4 (IOSTS). *An IOSTS (Q, q_0, Tr) over Σ and $\Gamma = (A, Ch)$ is a triple where Q is a set of states, $q_0 \in Q$ is the initial state and $Tr \subseteq Q \times Sen_F(A) \times Act(\Gamma) \times T_\Sigma(A)^A \times Q$ 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;*
- *$\rho \in T_\Sigma(A)^A$ is a substitution associating a term to attribute variables;*

Remark 1. We can always consider an IOSTS in which guards only contain conjunctions. If not, for a transition tr of guard ψ, it suffices to use a disjunctive normal form $\bigvee_{i=1}^{n} \psi_i$ equivalent to ψ and to split the transition into n transitions having the same source, target and communication action as tr and ψ_i as guard.

Example 2 (Drink vending machine). We consider a very simple drink vending machine. Its behavior is specified by the IOSTS in Fig. 1. An initialization step ($q \rightarrow q_0$) sets the amount to zero. Then, in q_0, the machine waits for an amount (x) of *coins* introduced by the user, and updates the amount m. The user then chooses his/her beverage (0 or 1 for "Tea" or "Coffee"). The choice is stored in variable B. In the transition $q_2 \rightarrow q_3$, the program *Price* computes the price of the chosen drink. Two cases are possible here. If the introduced amount is lower than the price ($m < p$), then a message "Add" appears on the screen and the machine returns to q_0. Otherwise ($m \geq p$), the drink is delivered, the amount is reinitialized to zero and the machine goes back to q_0. Note that transitions outgoing from q_3 constrain the value (p) computed by *Price* ($p \geq 150 \wedge p \leq 200$).

For a transition $tr = (q, \psi, act, \rho, q') \in Tr$ and a P-model $\mathcal{S}em$, the semantics of tr, denoted as $Run(tr, \mathcal{S}em) \subseteq M^A \times Act^M(\Gamma) \times M^A$, is defined as the set of triple (ν_i, act_M, ν_f) verifying:

- if act is of the form $c!t$ (*resp.* τ), then $M \models_{\nu_i} \psi$, $\nu_f = \nu_i \circ \rho$ and $act_M = c!\nu_i(t)$ (*resp.* $act_M = \tau$)
- if act is of the form $c?x$, then $M \models_{\nu_i} \psi$, there exists ν_a such that $\nu_a(z) = \nu_i(z)$ for every $z \neq x$, $\nu_f = \nu_a \circ \rho$ and $act_M = c?\nu_a(x)$,

Note that the definition of semantics of transitions is very classical and does not explicitly refers to $\mathcal{S}em$. In fact, semantics of programs are taken into account when defining ν_f from the extensions of ν_i or ν_a to $T_\Sigma(A)$ as defined in Sect. 3.

For a run $r = (\nu_i, act_M, \nu_f)$, we note $source(r)$, $act(r)$ and $target(r)$ resp. for ν_i, act_M and ν_f. ν_i and ν_f are the interpretation of attribute variables resp.

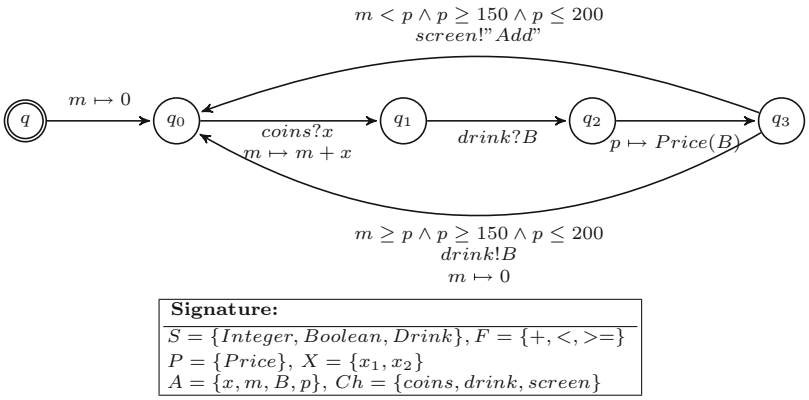

Fig. 1. IOSTS of the drink vending machine.

before and after executing the transition. Let us observe that, given a transition tr and an interpretation ν_i, the set $Run(tr, \mathcal{S}em)$ does not necessarily contain a run of the form (ν_i, act_M, ν_f) due to the fact that ν_i may not satisfy ψ.

The set of paths of an IOSTS $\mathbb{G} = (Q, q_0, Tr)$, denoted $Path(\mathbb{G})$, are all finite sequences $tr_1.\cdots.tr_n$ of transitions with $source(tr_1) = q_0$ and $\forall i, 1 \leq i < n$, $target(tr_i) = source(tr_{i+1})$. The set of runs of a path $pa = tr_1.\cdots.tr_n$ in $Path(\mathbb{G})$, denoted as $Run(pa, \mathcal{S}em)$, are sequences $r_1.\cdots.r_n$ such that $\forall i \leq n$, $r_i \in Run(tr_i, \mathcal{S}em)$ and $\forall i < n$, $target(r_i) = source(r_{i+1})$. Similarly, the set of traces $Traces(pa, \mathcal{S}em)$ of pa is the set of sequences $act(r_1).\cdots.act(r_n)$ for all $r_1.\cdots.r_n \in Run(pa, \mathcal{S}em)$, $act(r)$ being equal to ε if $act(r) = \tau$.

In general, it is not guaranteed that there exists at least a run for a given path pa, as it depends on the semantics associated to programs involved in pa.

Definition 5 (Path Feasibility Condition). *Let $\mathbb{G} = (Q, q_0, Tr)$ be an IOSTS over $\Gamma = (A, Ch)$ and pa a path of $\mathbb{G}$. pa is a* feasible path *if and only if:*

$$\exists\ \mathcal{S}em \in Mod(\mathbb{C}_\emptyset), Traces(pa, \mathcal{S}em) \neq \emptyset$$

Let W be a set of frozen variables provided with $Ax \subseteq Sen_F(W)$ and $\nu \in M^W$ an interpretation satisfying $M \models_\nu Ax$. Let us consider $\mathbb{C} = (C_p)_{p \in P}$ a family of contracts indexed by P. pa is a feasible path *up to $(\nu, \mathbb{C})$ if and only if:*

$$\exists \mathcal{S}em \in Mod_\nu(\mathbb{C}), Traces(pa, \mathcal{S}em) \neq \emptyset$$

5 Symbolic Execution and Path Feasibility Condition

Symbolic execution consists in executing an IOSTS for symbolic values (taken from a dedicated set of frozen variables $Fr = \coprod_{s \in S} Fr_s$) rather than numerical ones, and computing constraints on those values for all possible IOSTS executions. The main novelties with respect to [11] are twofold: substitutions occurring

in transitions may include program calls and a renaming mechanism ensures that a given frozen variable can not appear in two distinct paths.

To store information concerning an execution, we use structures called symbolic states. A *symbolic state* is a tuple of the form $(q, \pi, \lambda, \kappa)$ where $q \in Q$, $\pi \in Sen_F(Fr)$, $\lambda : A \to T_F(Fr)$ is an application preserving types and $\kappa \subset P \times T_F(Fr)^* \times Fr$. For a symbolic state $\eta = (q, \pi, \lambda, \kappa)$, q (or $q(\eta)$) denotes the state reached after an execution leading to η, π (or $\pi(\eta)$) is a constraint on variables in Fr called *path condition* that should be satisfied for the execution to reach η, λ (or $\lambda(\eta)$) denotes terms over variables in Fr that are assigned to variables of A and κ (or $\kappa(\eta)$) denotes the set of tuples of the form $(p, (t_1, \cdots, t_n), x)$ indicating that a program call has been performed for the program p with the arguments $(t_1, \cdots, t_n)$ and that its result is stored in the variable x in Fr.

In our approach we do not have the code of programs. Instead, we reason on their contracts. Since the input formal parameters associated to a call are represented symbolically by functional terms $t_1, \cdots, t_n$, different pre-conditions may hold depending on the way those terms will be interpreted. At the symbolic execution level, we thus consider a sub-case for each of those pre-conditions. More precisely, the symbolic execution of a transition tr from a given symbolic state η will consist in a set of symbolic transitions, one for each possible combination of pre-conditions for all program calls occurring in tr. We now introduce some notations aiming at tracing program calls: for a substitution $\rho : A \to T_\Sigma(A)$ and for $p \in P$, $Res(p, \rho)$ is the set of variables $y \in A$ such that $\rho(y)$ is of the form $p(t_1, \cdots, t_n)$ and for such an y, $Arg(y, \rho)$ is then $(t_1, \cdots, t_n)$ and $Prog(y, \rho) = p$. We also denote $Res(\rho)$ for $\bigcup_{p \in P} Res(p, \rho)$.

Definition 6 (Symbolic Execution of Transitions). *Let $\mathbb{G} = (Q, q_0, Tr)$ be an IOSTS over Σ and $\Gamma = (A, Ch)$, $tr = (q, \psi, act, \rho, q') \in Tr$ be a transition and $\eta = (q, \pi, \lambda, \kappa)$ be a symbolic state over $\mathbb{G}$.*

If act is of the form $c?x$, $\lambda_i = \lambda[x \mapsto f]$, f fresh in Fr. Otherwise, $\lambda_i = \lambda$.

λ' is the substitution such that for all $y \in Res(\rho)$, $\lambda'(y)$ is a fresh variable of Fr and for all $y \in A \setminus Res(\rho)$, $\lambda'(y) = \lambda_i \circ \rho(y)$.

The symbolic execution *$SE(tr, \eta)$ of tr from η is the set defined as follows:*

- *if $Res(\rho) = \emptyset$ then $SE(tr, \eta) = \{(\eta, \lambda_i(act), \eta')\}$ with $\eta' = (q', \pi \wedge \lambda(\psi), \lambda', \kappa)$.*
- *if $Res(\rho) \neq \emptyset$, we consider all mappings $Beh : Res(\rho) \to \bigcup_{p \in P} C_p$ such that for $y \in Res(p, \rho)$, $Beh(y) = (Pre_y, Post_y) \in C_p$. For $y \in Res(p, \rho)$ with $InOut(p) = (x_1, \cdots, x_n, x_{n+1})$ and $Arg(y, \rho) = (t_1, \cdots, t_n)$, we have $(\eta, \lambda_i(act), \eta') \in SE(tr, \eta)$ with*
  - *η' the symbolic state $(q', \pi \wedge \lambda(\psi) \wedge \bigwedge_{y \in Res(\rho)} \Delta(y), \lambda', \kappa')$*
  - *$\Delta(y) = (Pre_y \wedge Post_y)[x_1 \mapsto \lambda_i(t_1) \cdots x_n \mapsto \lambda_i(t_n), x_{n+1} \mapsto \lambda'(x)]$*
  - *κ' the set $\kappa \cup \bigcup_{y \in Res(\rho)} \{(Prog(y, \rho), (\lambda_i(t_1), \cdots, \lambda_i(t_n)), \lambda'(y))\}$*

Elements of $SE(tr, \eta)$ are called symbolic transitions. *We denote $Fr(\eta')$ the set of all fresh variables of Fr occurring in its definition.*

Example 3. In order to illustrate Definition 6, let us consider a transition tr of the form $(q, \psi, c?x, \rho, q')$ with $\rho = [y \mapsto p_1(t_1, t_2), z \mapsto t'_1 + t'_2]$ with p_1 a

program and t_1, t_2, t'_1, t'_2 functional terms. Let us observe that $Res(p_1, \rho) = \{y\}$, $Arg(y, p_1) = (t_1, t_2)$, $Prog(y, \rho) = p_1$ and $Res(\rho) = \{y\}$.

Let $\eta = (q, \pi, \lambda, \kappa)$ be a symbolic state. Let us suppose that the program p_1 is provided with an interface (x_1, x_2, x_3) and with a behavior $(Pre_1, Post_1)$. Then $SE(tr, \eta)$ contains the symbolic transition $(\eta, c?f_1, \eta')$ with f_1 a fresh variable of Fr and η' the symbolic state defined as:

$$\begin{array}{l}(q', \pi \wedge \lambda(\psi) \wedge (Pre_1 \wedge Post_1)[x_1 \mapsto \lambda[x \mapsto f_1](t_1), x_2 \mapsto \lambda[x \mapsto f_1](t_2), x_3 \mapsto f_2] \\ \quad [x \mapsto f_1, y \mapsto f_2, z \mapsto \lambda[x \mapsto f_1](t'_1 + t'_2)], \\ \quad \kappa \cup \{(p_1, (\lambda[x \mapsto f_1](t_1), \lambda[x \mapsto f_1](t_2)), f_2\})\end{array}$$

$Fr(\eta')$ is then $\{f_1, f_2\}$.

Definition 7 (IOSTS Symbolic Execution). *Given an IOSTS $\mathbb{G}$, the symbolic execution $SE(\mathbb{G}) = (Init, ST)$ of $\mathbb{G}$ is minimally defined by:*

- *$Init = (q_0, Ax, \lambda_0)$ with $\forall x \in A, \lambda_0(x) \in Fr$ and $\forall x \neq y \in A, \lambda_0(x) \neq \lambda_0(y)$,*
- *for $tr \in Tr$ and η symbolic state with $source(tr) = q(\eta)$, $SE(tr, \eta) \subseteq ST$.*
- *for any distinct $SE(tr_1, \eta_1)$ $SE(tr_2, \eta_2)$ that are defined, $Fr(SE(tr_1, \eta_1)) \cap Fr(SE(tr_2, \eta_2)) = \emptyset$.*

Definition 8 (Paths and Distinguished Paths). *The set $Paths(SE(\mathbb{G}))$ of* paths *of $SE(\mathbb{G})$ is the set of all sequences $tr_1 \cdots tr_n$ with $\forall i \in 1..n, tr_i \in ST$ such that $source(tr_1) = Init$ and for any $j < n$, $q(target(tr_j)) = q(source(tr_{j+1}))$.*

For a non-empty sequence $\delta = tr_1 \cdots tr_n$, we note $End(\delta) = target(tr_n)$ and $Fr(\delta) = \cup_{i \in 1..n} Fr(target(tr_i))$. By convention, $End(\varepsilon) = Init$ and $Fr(\varepsilon) = \emptyset$.

Given a finite subset Δ of $Paths(SE(\mathbb{G}))$, $DPaths(\Delta)$ is a set of paths δ^* such that there exists an unique path δ in Δ such that δ and δ^* are isomorphic up to a renaming of variables of Fr and such that for two distinct paths δ_1^* and δ_2^* in $DPaths(\Delta)$, $Fr(\delta_1^*) \cap Fr(\delta_2^*) = \emptyset$.

We say that $DPaths(\Delta)$ is a set of distinguished *paths issued from $SE(\mathbb{G})$.*

Generally speaking, a set Δ of $Paths(SE(\mathbb{G}))$ represents a tree whose transitions issued from the root $Init$ can be shared by several paths of Δ while $DPaths(\Delta)$ consists in applying a variable renaming mechanism in order to duplicate shared transitions to completely separate paths. Distinguished paths can still share common variables, namely those in W.

Example 4. The drink vending machine of Fig. 1 has two possible paths from q to q_0 with exactly one cycle on q_0. They share a transition with a call to program $Price$ defined by its contract C_r as seen in Example 1. We thus get 4 distinguished paths shown in Fig. 2. Associated path conditions are the following:

$$\begin{array}{l}pc_1 : B_1 = 0 \wedge p_1 \geq 100 \wedge p_1 \leq 200 \wedge v_1 < p_1 \wedge p_1 \geq 150 \wedge p_1 \leq 200 \\ pc_2 : B_2 = 0 \wedge p_2 \geq 100 \wedge p_2 \leq 200 \wedge v_2 \geq p_2 \wedge p_2 \geq 150 \wedge p_2 \leq 200 \\ pc_3 : B_3 = 1 \wedge p_3 \geq 200 \wedge p_3 \leq 300 \wedge v_3 < p_3 \wedge p_3 \geq 150 \wedge p_3 \leq 200 \\ pc_4 : B_4 = 1 \wedge p_4 \geq 200 \wedge p_4 \leq 300 \wedge v_4 \geq p_4 \wedge p_4 \geq 150 \wedge p_4 \leq 200\end{array}$$

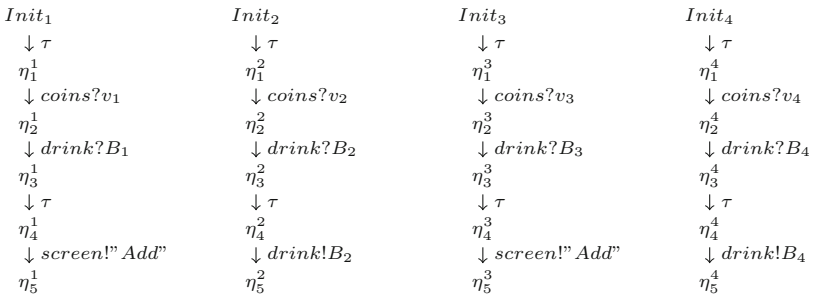

Fig. 2. Symbolic paths.

A path condition is a formula over the frozen variables built by accumulating constraints from the guards of the IOSTS transitions and from constraints of called programs contracts. The path is infeasible if its path condition is not satisfiable. In addition, this feasibility depends on the fact that if a program is called twice with the same arguments, it returns the same value (semantic condition of Definition 1). Since this is not enforced by the path condition alone, we consider another set of constraints accounting for this condition:

Definition 9 (Feasibility of a Set of Paths). *Let $\mathbb{G}$ be an IOSTS over $\Gamma = (A, Ch)$ and let Δ^* be a set of distinguished paths issued from $SE(\mathbb{G})$.*

For any program p of interface $(x_1, \cdots, x_n, x_{n+1})$, for $(p, (t_1, \cdots, t_n), f)$ and $(p, (t'_1, \cdots, t'_n), f')$ two distinct elements of $\cup_{\delta^* \in \Delta^*} \kappa(End(\delta^*))$ we introduce the deterministic program condition *relating to these two program calls as the formula $\phi_{\{f,f'\}}$ defined by $\bigwedge_{i=1}^{n} t_i = t'_i \Rightarrow f = f'$.*

The deterministic program condition *related to Δ^* is then $\Phi_p = \bigwedge \phi_{\{f,f'\}}$, for all f and f' appearing as return variable of a call of p in Δ^*.*

Finally, the feasibility condition of Δ^* *is*

$$\bigwedge_{\delta^* \in \Delta^*} \pi(End(\delta^*)) \wedge \bigwedge_{p \in P} \Phi_p$$

If this feasibility condition holds, it is possible to implement the programs occurring in the IOSTS so that all paths of Δ^* will complete successfully. Note that the contracts of the programs are taken into account in the path condition, and have thus an impact on the paths that are feasible or not.

Example 5. In the context of the drink vending machine, we now want to check the feasibility condition of the distinguished paths associated to the paths described in Example 4 according to two distinct contracts for *Price*, denoted resp. C_w and C_r (in Example 1). Both C_w and C_r include two behaviors resulting in 4 distinguished paths. Path conditions are given in Table 1.

– With the contract C_w, no distinguished path is feasible because of contradictions between guards of the IOSTS transitions and post-conditions of C_w.

– With the contract C_r, all distinguished paths are feasible. *Price* can return anything between 150 and 200 for an argument equal to 0 and must return 200 for an argument equal to 1.

Table 1. Feasibility according to different contracts

$C_w : \{(x_1 = 0, x_2 \geq 0 \wedge x_2 \leq 100), (x_1 = 1, x_2 \geq 250)\}$
$pc_1 : B_1 = 0 \wedge p_1 \geq 0 \wedge p_1 \leq 100 \wedge v_1 < p_1 \wedge p_1 \geq 150 \wedge p_1 \leq 200$
$pc_2 : B_2 = 0 \wedge p_2 \geq 0 \wedge p_2 \leq 100 \wedge v_2 \geq p_2 \wedge p_2 \geq 150 \wedge p_2 \leq 200$
$pc_3 : B_3 = 1 \wedge p_3 \geq 250 \wedge v_3 < p_3 \wedge p_3 \geq 150 \wedge p_3 \leq 200$
$pc_4 : B_4 = 1 \wedge p_4 \geq 250 \wedge v_4 \geq p_4 \wedge p_4 \geq 150 \wedge p_4 \leq 200$
$\phi_{\{p_1,p_2\}} : B_1 = B_2 \Rightarrow p_1 = p_2$
$\phi_{\{p_1,p_3\}} : B_1 = B_3 \Rightarrow p_1 = p_3$
$\phi_{\{p_1,p_4\}} : B_1 = B_4 \Rightarrow p_1 = p_4$
$\phi_{\{p_2,p_3\}} : B_2 = B_3 \Rightarrow p_2 = p_3$
$\phi_{\{p_2,p_4\}} : B_2 = B_4 \Rightarrow p_2 = p_4$
$\phi_{\{p_3,p_4\}} : B_3 = B_4 \Rightarrow p_3 = p_4$
Feasibility: No

$C_r : \{(x_1 = 0, x_2 \geq 100 \wedge x_2 \leq 200), (x_1 = 1, x_2 \geq 200 \wedge x_2 \leq 300)\}$
$pc_1 : B_1 = 0 \wedge p_1 \geq 100 \wedge p_1 \leq 200 \wedge v_1 < p_1 \wedge p_1 \geq 150 \wedge p_1 \leq 200$
$pc_2 : B_2 = 0 \wedge p_2 \geq 100 \wedge p_2 \leq 200 \wedge v_2 \geq p_2 \wedge p_2 \geq 150 \wedge p_2 \leq 200$
$pc_3 : B_3 = 1 \wedge p_3 \geq 200 \wedge p_3 \leq 300 \wedge v_3 < p_3 \wedge p_3 \geq 150 \wedge p_3 \leq 200$
$pc_4 : B_4 = 1 \wedge p_4 \geq 200 \wedge p_4 \leq 300 \wedge v_4 \geq p_4 \wedge p_4 \geq 150 \wedge p_4 \leq 200$
$\phi_{\{p_1,p_2\}} : B_1 = B_2 \Rightarrow p_1 = p_2$
$\phi_{\{p_1,p_3\}} : B_1 = B_3 \Rightarrow p_1 = p_3$
$\phi_{\{p_1,p_4\}} : B_1 = B_4 \Rightarrow p_1 = p_4$
$\phi_{\{p_2,p_3\}} : B_2 = B_3 \Rightarrow p_2 = p_3$
$\phi_{\{p_2,p_4\}} : B_2 = B_4 \Rightarrow p_2 = p_4$
$\phi_{\{p_3,p_4\}} : B_3 = B_4 \Rightarrow p_3 = p_4$
Feasibility: Yes

6 Testing

6.1 Model-Based Testing of IOSTS with Program Calls and Contracts

In a previous work [11], we have proposed an online testing algorithm to test Systems Under Test (SUT) with respect to a basic IOSTS (without program calls). The algorithm is based on the *ioco* conformance relation [21] and on the use of test purposes (TP) to select some behaviors to be tested. A TP is a finite sub-tree of the symbolic execution structure (SES) derived from the IOSTS of reference so that any execution trace constructed by interacting with SUT and leading to a leaf of TP will be considered as covering TP. The testing process is implemented as a simultaneous traversal of both SES and TP. Verdicts depend on whether the observed execution trace does or does not belong to TP and SES: $WeakPASS$ when the execution trace covers TP and belongs to at least one path of SES which does not end at a leaf of TP, $PASS$ when the execution trace covers TP and does not belong to another path of SES, $INCONC$ (for inconclusive) when the execution trace belongs to SES but does not cover TP, $FAIL$ when the execution trace does not cover TP and goes outside SES.

In Sect. 5, we have associated to any IOSTS with contracts a symbolic tree structure in order to be able to use it both as the SES input of the algorithm given in [11] and as a carrier to extract a finite sub-tree to play the role of TP. We can use the work described in [11] with the following slight modifications:

- Unlike [11], we allow unobservable τ transitions. Under the assumption that there does not exist a cycle of τ transitions, we can replace any sequence of consecutive τ transitions by a transition carrying the input/output action just located at the end of the τ sequence. Furthermore, in [11], quiescence conditions are expressed by enriching the reference IOSTS with transitions carrying the special label δ denoting the intended absence of reaction. Because the presence of τ transitions makes such a direct enrichment tricky, it becomes more appropriate to perform this enrichment at the level of the τ-reduced symbolic execution itself. Once the operations of τ-reduction and δ-enrichment are applied to the symbolic execution of the IOSTS with contracts, we can then apply the algorithm of [11] for free.
- In [11], path conditions for paths that are part of test purposes are satisfiable by construction. In our setting, we have to take into account the notion of feasibility, i.e., the existence or not of programs that meet their associated contracts and that are compatible with considered paths. Indeed, if the considered set of distinguished paths constituting the test purpose is unfeasible, then the application of algorithm is meaningless. In other words, the feasibility of the targeted set of paths plays the role of a testing hypothesis.

6.2 Contracts Inference

As we have seen in Sect. 5, the feasibility condition checks whether a given program contract preserves the feasibility of a symbolic path or not. In this section, we focus on the inference of contracts based on path conditions. Such contracts can then be used to define unit tests for the programs. More precisely, we start with an IOSTS $\mathbb{G}$ calling programs without associated contract. We then show that we can infer contracts such that feasible paths of $\mathbb{G}$ are guaranteed to verify the feasibility condition of the IOSTS augmented with contracts. The generated contract for a program p contains one behavior per call to p in $SE(\mathbb{G})$. For that, we use the parts of the final condition of the path on which the call occurs that are related to the return variable and to the arguments.

Given a formula F, we define inductively the set $Rel_F(X)$ of variables related to a set X of variables, as the smallest set satisfying the following conditions

- $X \subset Rel_F(X)$
- $Occ(t_1 = t_2) \cap Rel_{t_1=t_2}(X) \neq \emptyset \Rightarrow Occ(t_1 = t_2) \subset Rel_{t_1=t_2}(X)$
- $Rel_{F_1}(X) \cup Rel_{F_2}(X) = Rel_{F_1 \wedge F_2}(X) = Rel_{F_1 \vee F_2}(X)$
- $Rel_F(X) = Rel_{\neg F}(X)$

Similarly, for a formula F and a set of variables X, $Clean_X(F)$ is defined as follows. As noted in Remark 1, we can assume that the path condition only has conjunctions, and is in negation-normal form.

- $Clean_X(\top) = \top$ and $Clean_X(\bot) = \bot$
- $Clean_X(t_1 = t_2) = t_1 = t_2$ if $Occ(t_1 = t_2) \cap X \neq \emptyset$
- $Clean_X(t_1 = t_2) = \top$ if $Occ(t_1 = t_2) \cap X = \emptyset$
- $Clean_X(\neg t_1 = t_2) = \neg t_1 = t_2$ if $Occ(t_1 = t_2) \cap X \neq \emptyset$
- $Clean_X(\neg t_1 = t_2) = \top$ if $Occ(t_1 = t_2) \cap X = \emptyset$
- $Clean_X(F_1 \wedge F_2) = Clean_X(F_1) \wedge Clean_X(F_2)$

Remark 2. If F is satisfiable, then $Clean_X(F)$ is also satisfiable, as we only remove atomic propositions from the conjunction.

Definition 10 (Contract Inference). *Let $\mathbb{G}$ be an IOSTS , Δ^* a set of distinguished paths from $SE(\mathbb{G})$. We note $\kappa(\Delta^*) = \cup_{\delta^* \in \Delta^*} \kappa(End(\delta^*))$.*

For any f such that $(p, (t_1, \cdots, t_n), f) \in \kappa(\Delta^*)$, with $In(p) = (x_1, ..., x_n)$ and $Out(p) = x_{n+1}$, we define a behavior $(Pre_f, Post_f)$ for p, as well as a set of frozen variables G_f and axioms Ax_f.

We pose $\phi = \pi(End(\delta^*))$ the final condition for the path containing the call and $Y = Occ(t_1, \cdots, t_n) \cup \{f\}$ the variables occurring in the call. Then

- *G_f is $Rel_\phi(Y)$*
- *Ax_f is $Clean_{Rel_\phi(Y)}(\phi)$*
- *Pre_f is $\bigwedge_{i=1}^{n} x_i = t_i$*
- *$Post_f$ is $x_{n+1} = f$*

Finally, the inferred contracts for Δ^* are defined as follows.

- *G is $\bigcup_{(p,(t_1,\cdots,t_n),f) \in \kappa(\Delta^*)} G_f$*
- *Ax is $\bigwedge_{(p,(t_1,\cdots,t_n),f) \in \kappa(\Delta^*)} Ax_f$*
- *$\forall p \in P, C_p = ((Pre_f, Post_f))_{(p,(t_1,\cdots,t_n),f) \in \kappa(\Delta^*)}$*

Example 6. Let us consider here a symbolic path δ^* of our drink vending machine's specification (Fig. 3) that calls twice the program *Price* of interface (x_1, x_2). The first call leads to the appearance of a message "Add" on the screen and the second call permits the drink delivery, such that:

$$\pi(End(\delta^*)) : v_1 < p_1 \wedge p_1 \geq 150 \wedge p_1 \leq 200 \wedge (v_1 + v_2) \geq p_2 \wedge p_2 \geq 150 \wedge p_2 \leq 200$$

From the path condition $\phi = \pi(End(\delta^*))$, two behaviors will be generated according to Definition 10. For p_1 the result of the first call $(Price, (B_1), p_1)$ we have:

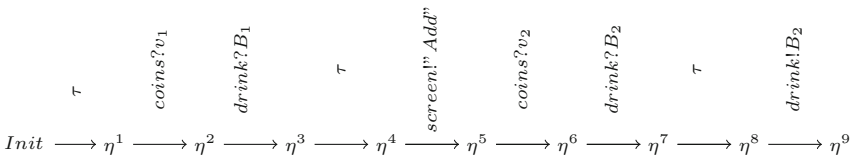

Fig. 3. Symbolic path.

$$
\begin{aligned}
Y &: \{B_1, p_1\} \\
G_{p_1} &: \{B_1, p_1, v_1, p_2, v_2\} \\
Ax_{p_1} &: v_1 < p_1 \wedge p_1 \geq 150 \wedge p_1 \leq 200 \wedge (v_1 + v_2) \geq p_2 \wedge p_2 \geq 150 \wedge p_2 \leq 200 \\
Pre_{p_1} &: x_1 = B_1 \\
Post_{p_1} &: x_2 = p_1
\end{aligned}
$$

For p_2 the result of the second call $(Price, (B_2), p_2)$ we have:

$$
\begin{aligned}
Y &: \{B_2, p_2\} \\
G_{p_2} &: \{B_2, p_2, v_1, p_1, v_2\} \\
Ax_{p_2} &: v_1 < p_1 \wedge p_1 \geq 150 \wedge p_1 \leq 200 \wedge (v_1 + v_2) \geq p_2 \wedge p_2 \geq 150 \wedge p_2 \leq 200 \\
Pre_{p_2} &: x_1 = B_2 \\
Post_{p_2} &: x_2 = p_2
\end{aligned}
$$

Finally, the inferred contract for our program $Price$ in δ^* is defined by: $G = G_{p_1} \cup G_{p_2}$, $Ax = Ax_{p_1} \wedge Ax_{p_2}$ and $C = ((Pre_{p_1}, Post_{p_1}), (Pre_{p_2}, Post_{p_2}))$

We can now define the IOSTS $\mathbb{G}'$ with the same signature and transitions than $\mathbb{G}$ and equipped with the inferred contracts for the programs in P. Then, for every path δ^* in Δ^* that is feasible, there exist paths $\underline{\delta^*}$ in $\mathbb{G}'$ similar to δ^* except that the path conditions π are augmented with axioms and behaviors. For each $(p, (t_1, ..., t_n), f) \in \kappa(End(\delta^*))$, Ax_f is satisfiable by Remark 2 and the behavior $(Pre_f, Post_f)$ becomes trivially true: one of the behaviors of p makes the corresponding transition feasible. Since this is true for any call in δ^*, there exists thus a path in $\underline{\delta^*}$ that is feasible. This leads to the following theorem.

Theorem 1 (Feasibility Preservation). *Let $\mathbb{G}$ be an IOSTS, Δ^* a set of feasible distinguished symbolic paths of $\mathbb{G}$. $\mathbb{G}'$ is the IOSTS obtained by adding to G the inferred contracts of Definition 10. For any path δ^* in Δ^*, there exists a symbolic path $\delta^{*\prime}$ for $\mathbb{G}'$ having the same transitions as δ^* and which is feasible.*

7 Related Work

In the context of reactive systems verification, IOSTS and symbolic execution have been used in many works [1,11,14] for different purposes. They use IOSTS with atomic actions and substitutions whereas, in our case, we enrich IOSTS with programs specified by contracts. Our purpose is to define an integration framework and analyze in one hand the impact of programs contracts on a whole system and in the other hand elicit accurate contracts for our programs.

Our work is quite close to [13], that augments a SOA's BPEL business model with pre- and post-condition contracts defining essential component traits, and derive a suite of feasible test cases, taking into account contracts that are provided for some of the opaque components of their system. On the other hand, they do not infer contracts from the constraints expressed directly in the BPEL model as is done in Sect. 6.2.

The use of symbolic execution and path feasibility analysis are studied in [3,22] but this is limited to the analysis of programs themselves and does not

take in consideration as we do the impact of the program calls on the feasibility of the system as a whole. Similarly, symbolic execution techniques over the code have been used to infer program annotations. More specifically, such approaches concentrate on generating invariants. This is for instance the case in the KeY verification framework [20], for the DySy tool [7], or for the iDiscovery tool [24]. Those invariants are meant to help the formal verification of the code against its specification, while we are aiming at generating a specification that the programs must meet in order to be usable in the context of the system under test.

The problem of inferring contracts for programs has been studied differently in other works that do not rely on symbolic execution. In particular, [6] derives pre-conditions from assertions already present in the code using abstract interpretation. [23] uses dynamic analysis to augment simple programmer-written contracts with candidate post-conditions that describes precisely what the code is doing, building upon techniques developed initially in the Daikon tool [9] for proposing likely invariants. This kind of inference is dual to ours, in the sense that we infer contracts in a top-down approach, in order to express what conditions individual components should fulfill inside a broader system, while the works mentioned above are bottom-up, encapsulating the behavior of actual code in contracts in order to check whether callers can use this particular implementation. The same can be said of works that aim at generating transition systems modeling the behavior of programs, either as message sequence charts as in [16], or as scenarios expressed under the form of live sequence charts, as in [17].

8 Conclusion

In this work, we extended the IOSTS framework with programs which are specified with contracts and we adapted symbolic execution techniques to deal with them. This gives rise to two main results. First, we study how contracts impact path conditions and describe the feasibility condition of the entire symbolic execution tree. Second, we show that path conditions can be used to infer contracts for programs in order to specify what these programs should do in the context of the system under test. Such contracts can then be used for unitary testing purposes, while feasibility preservation theorem gives some guarantees that program calls will not get in the way during integration testing.

The contribution of this paper is mainly theoretical, only illustrated by a toy example by lack of space. In [4], we provide a more realistic example involving a program call for giving the money change by considering three possible values for coins and the state of the coin reserve of the vending machine.

Implementation of the technique presented in this paper is currently under development in the Diversity [8] symbolic execution tool and the Frama-C [15] C code analysis framework using the ACSL specification language [2] as target for contract inference.

This work is in its early stages, nevertheless it provides a promising framework to explore integration testing for systems whose user scenarios are described using some IOSTS extensions (e.g. UML Sequence Diagrams [19]) and whose unitary bricks are program calls.

References

1. B. Bannour. Symbolic analysis of scenario based timed models for component-based systems: Compositionality results for testing. PhD thesis, Ecole Centrale Paris, CEA, 2012
2. Baudin, P., Filliâtre, J.-C., Hubert, T., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, v1.9, March 2015
3. Bjørner, N., Tillmann, N., Voronkov, A.: Path feasibility analysis for string-manipulating programs. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 307–321. Springer, Heidelberg (2009)
4. Boudhiba, I., Gaston, C., Le Gall, P., Prevosto, V.: Input ouput symbolic transitions systems enriched by program calls and contracts : a detailed example of a vending machine. Technical report hal-01191890, MAS Laboratory, CentraleSupelec (2015)
5. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond assertions: advanced specification and verification with JML and ESC/Java2. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 342–363. Springer, Heidelberg (2006)
6. Cousot, P., Cousot, R., Logozzo, F.: Precondition inference from intermittent assertions and application to contracts on collections. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 150–168. Springer, Heidelberg (2011)
7. Csallner, C., Tillmann, N., Smaragdakis, Y.: Dysy: Dynamic symbolic execution for invariant inference. In Proceedings of ICSE (2008)
8. Deltour, J., Faivre, A., Gaudin, E., Lapitre, A.: Model-based testing: an approach with SDL/RTDS and DIVERSITY. In: Amyot, D., Fonseca i Casas, P., Mussbacher, G. (eds.) SAM 2014. LNCS, vol. 8769, pp. 198–206. Springer, Heidelberg (2014)
9. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. Trans. Soft. Eng. **27**, 99–123 (2001)
10. Floyd, R.W.: Assigning meanings to programs. In: Proceedings AMS Symposium on Applied Mathematics, vol. 19 (1967)
11. Gaston, C., Le Gall, P., Rapin, N., Touil, A.: Symbolic execution techniques for test purpose definition. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, pp. 1–18. Springer, Heidelberg (2006)
12. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–583 (1969)
13. Jehan, S. Pill, I., Wotawa, F.: Functional SOA testing based on constraints. In: Automation of Software Test (2013)
14. King, J.C.: Symbolic execution and program testing. Comm. ACM **17**, 385–395 (1976)
15. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. Formal Aspects Comput. **27**, 573–609 (2015)
16. Kumar, S., Khoo, S.-C., Roychoudhury, A., Lo, D.: Inferring class level specifications for distributed systems. In: Proceedings ICSE (2012)
17. Lo, D., Maoz, S.: Scenario-based and value-based specification mining: better together. Autom. Softw. Eng. **19**, 423–458 (2012)
18. Meyer, B.: Applying "design by contract". IEEE Comput. **25**(10), 40–51 (1992)
19. Object Management Group. OMG Unified Modeling Language^{TM} (OMG UML), version 2.5 edition (2013)

20. Schmitt, P.H., Weiß, B.: Inferring invariants by symbolic execution. In: VERIFY Workshop. CEUR Workshop Proceedings, vol. 259 (2007)
21. Tretmans, J.: Conformance testing with labelled transition systems: implementation relations and test generation. Comput. Netw. ISDN Syst. **29**, 49–79 (1996)
22. Wang, Y., Xing, Y., Zhang, X.: A method of path feasibility judgment based on symbolic execution and range analysis. Int. J. Future Gener. Commun. Networking **7**, 205–212 (2014)
23. Wei, Y., Furia, C.A., Kazmin, N., Meyer, B.: Inferring better contracts. In: Proc ICSE (2011)
24. Zhang, L., Yang, G., Rungta, N., Person, S., Khurshid, S.: Invariant discovery guided by symbolic execution. In: The Java PathFinder Workshop (2013)