Symbolic Execution Techniques for Refinement Testing*

Pascale Le Gall[1], Nicolas Rapin[2], and Assia Touil[1]

[1] Université d'Évry, IBISC - FRE CNRS 2873,
523 pl. des Terrasses F-91000 Évry
{pascale.legall,assia.touil}@ibisc.univ-evry.fr
[2] CEA/LIST Saclay
F-91191 Gif sur Yvette
{nicolas.rapin}@cea.fr

Abstract. We propose an approach to test whether an abstract specification is refined or not by a more concrete one. The specifications are input / output symbolic transition systems (IOSTS). The refinement relation requires that all traces of the abstract system are also traces of the concrete system, up to some signature inclusion. Our work takes inspiration from the conformance testing area. Symbolic execution techniques allow us to select traces of the abstract system and to submit them on the concrete specification. Each trace execution leads to a verdict $Fail$, $Pass$ or $Warning$. The verdict $Pass$ is provided with a formula which has to be verified by the values only manipulated at the level of the concrete specification in order to ensure the refinement relation. The verdict $Warning$ reports that the concrete specification has not been sufficiently explored to give a reliable verdict. This is thus a partial verification process, related to the quality of the set of selected traces and of the exploration of the concrete specification. Our approach has been implemented and is demonstrated on a simple example.

Keywords: refinement, conformance testing, symbolic execution, symbolic transition system.

1 Introduction

Formal specifications serve as references for the rigorous definition of correct implementations. Implementation correctness is usually based on some hypotheses stating that implementations can be modelled as a formal model. For example, specifications can be used to generate test cases in order to verify whether an implementation conforms or not to its specification. However, it is widely recognised that it is often difficult to write the right formal specifications in adequacy to the informal requirements given by the users. To overcome this difficulty, refinement techniques are often advocated to help the designers to incrementally design a detailed specification. Implementation design choices (non-determinism

* This work was partially supported by the RNRT French project STACS.

B. Meyer and Y. Gurevich (Eds.): TAP 2007, LNCS 4454, pp. 131–148, 2007.

elimination, data types concretisation) are progressively introduced in specifications such that the specification design becomes a general stepwise refinement process from the more abstract specification to the more concrete specification [24]. Then, the executable implementation may be simply derived, by handwriting code or by automatic code generation techniques. Intuitively, a concrete specification Sp_2 refines another abstract one Sp_1 if it has the same behaviours, up to some formal refinement relation. According to the considered underlying formalism, the refinement process is more or less equipped with verification techniques and tools. For example, model-oriented frameworks like the B method [2] or property-oriented frameworks like algebraic specifications [13] are provided with a theory of specification refinement, mainly based on proof-based verification : proof obligations are associated to each refinement step. For formalisms based on transition systems (labelled transition systems, input/output transition systems, Petri Net, etc), the refinement relation is generally expressed using some relations of simulation or notions of trace containment (see for example [3,23]). In this paper, we focus on specifications described with symbolic transitions systems (STS). They are finite state transition automata including first-order data used both to characterise internal states and to guard transitions by means of first-order conditions. They provide us with an appropriate level of abstraction and are useful to avoid the classical state explosion problem. We find these symbolic models under different names STG [14], STS [5,6] or IOSTS [16,10,11]. We use IOSTS formalism defined in [12,20] that is very similar to the systems used in [16,10]. The aim of the paper is to verify a refinement step following the reasoning of conformance testing. Thus, the key idea is to extract from the abstract specification Sp_1 some representative behaviours, or test cases, and then to submit them to the concrete specification Sp_2 in order to get a verdict. Symbolic execution techniques will be used not only to select test cases from Sp_1 as in [12] but also to execute test cases on Sp_2. Indeed, unlike conformance testing for which verdicts come from the execution of the system under test with test cases as input data, refinement testing requires to be able to analyse Sp_2 with respect to the abstract requirements. Symbolic execution techniques precisely allow us to explore Sp_2 according to the selected abstract behavior given as a trace. The verdict depends on the satisfiability of the associated path condition computed on Sp_2. Related works (e.g. [15]) on verification of STS mainly concern symbolic bisimulation relations. They involve an algebra of regions over the data type part provided with operations supposed to be decidable. Unlike such works, we take into account the fact that generally, Sp_2 has often a larger interface than Sp_1, and thus, the signature of Sp_2 may strictly contain the one of Sp_1. Thus, refinement verification precisely requires to automatically compute data emitted and received at the concrete level ensuring the abstract requirements. Symbolic execution provided with some constraint solving mechanisms allows us to perform such computations on Sp_2. Moreover, from a practical point of view, our testing-based approach allows us to more easily debug the concrete specification Sp_2 when a verdict $Fail$ is emitted. Indeed, the corresponding unsatisfiable path condition gives some clues to modify Sp_2 in order to ensure the refinement

relation with Sp_1. Testing and refinement have already been linked in previous works. Most of them [9,22] study the relationship between abstract tests selected from an abstract specification and concrete tests which are submitted to the implementation under test (IUT). Generally, the IUT interface is such that an abstract action (or function) of the specification may be decomposed into elementary actions making explicit how the abstract action is concretely achieved by the IUT. We are not interested in this problem but we rather focus on the testing-based method for the partial verification of a refinement step between two specifications.

The paper is structured as follows. In Section 2 we present IOSTS, their syntax and semantics. The refinement relation is introduced in Section 3. A theorem relates the refinement relation with all symbolic executions of a concrete specification with respect to all traces of the abstract specification. This result will found our method given in Section 4 which aims at testing whether a concrete specification verifies or not an abstract one. Our approach is illustrated by an example and some details on algorithms and implementations are given. Finally, Section 5 contains concluding remarks.

2 Input Output Symbolic Transition Systems

Reactive systems are open systems interacting with their environment. Such systems can be modeled by using Input/Output Symbolic Transition Systems (IOSTS). Communications consist of sending or receiving messages represented by first-order terms through communication channels. IOSTS specify dynamic aspects of reactive systems by describing possible evolutions of system states. This is done by modifying values associated to some variables, called *attribute variables*, in order to denote system state modifications. Each elementary modification is given by a transition labelled by a communication action (sending or receipt of messages, or an internal action), guards expressed with first-order properties, and assignments of attribute variables.

2.1 Data Types

Let us first introduce the data part of the IOSTS formalism. Data types are specified with a many-sorted first-order equational logic.

Syntax. A data type signature is a couple $\Omega = (S, Op)$ where S is a set of type names, Op is a set of operation names, each one provided with a profile $s_1 \cdots s_{n-1} \rightarrow s_n$ (for $i \leq n$, $s_i \in S$). Let $V = \bigcup_{s \in S} V_s$ be a set of typed variable names. The set of *Ω-terms* with variables in V is denoted $T_\Omega(V) = \bigcup_{s \in S} T_\Omega(V)_s$ and is inductively defined as usual over Op and V. $T_\Omega(\emptyset)$, simply denoted T_Ω, is the set of all ground terms that have no occurrences of variables. A *Ω-substitution* is a function $\sigma : V \rightarrow T_\Omega(V)$ preserving types. In the following, we denote by

$T_\Omega(V)^V$ the set of all Ω-substitutions of the variables V. Any substitution σ may be canonically extended to terms (and will be also noted σ). The set $Sen_\Omega(V)$ of all typed equational *Ω-formulae* contains the constant symbols $\top$, $\bot$ (denoting the usual truth values *truth* and *false*) and all formulae built using the equality predicates $t = t'$ for $t, t' \in T_\Omega(V)_s$, and the usual connectives $\neg, \vee, \wedge, \Rightarrow$.

Semantics. A *Ω-model* is a family $M = \{M_s\}_{s \in S}$ with, for each $f : s_1 \cdots s_n \rightarrow s \in Op$, a function $f_M : M_{s_1} \times \cdots \times M_{s_n} \rightarrow M_s$. We define *$\Omega$-interpretations* as applications ν from V to M preserving types, extended to terms in $T_\Omega(V)$. A model M satisfies a formula φ, denoted by $M \models \varphi$, iff, for all interpretations ν, $M \models_\nu \varphi$, where $M \models_\nu t = t'$ is defined by $\nu(t) = \nu(t')$, and where the constant symbols $\top$ and $\bot$ and the connectives are handled as usual. M^V is the set of all Ω-interpretations from V to M. Given a model M and a formula φ, φ is said *satisfiable* in M, if there exists an interpretation ν s.t. $M \models_\nu \varphi$.

In the sequel, we suppose that data types of all IOSTS correspond to an arbitrary common data signature $\Omega = (S, Op)$ and are interpreted in a fixed model M. So, the data type signature Ω will be left implicit in the sequel. Moreover, elements of M will be called *concrete data* and denoted by ground terms in T_Ω. The examples illustrating our approach will be built on data types issued from Presburger arithmetics and from some enumerated types. So, concrete data will be natural numbers or boolean values provided with some usual operations as addition, comparison operators, etc. Moreover, expressions such as $\leq (5, x) = \top$ will be simply denoted $5 \leq x$.

2.2 Syntax

Definition 1 (IOSTS-signature). *An $IOSTS$-*signature *Σ is a couple (A, C) where $A = \bigcup_{s \in S} A_s$ is a set of variable names, called* attribute variables*, over the signature Ω and where C is a set of* communication channel *names.*

Let $\Sigma_1 = (A_1, C_1)$ and $\Sigma_2 = (A_2, C_2)$ two $IOSTS$-signatures. Σ_1 is said to be included in Σ_2, denoted by $\Sigma_1 \subseteq \Sigma_2$, iff $C_1 \subseteq C_2$.

For a given $IOSTS$-signature $\Sigma = (A, C)$, the set C of communication channels represents the interface of the corresponding IOSTS while the set A of attribute variables is used to characterize the different states of the IOSTS, and thus are internal information of the IOSTS. It explains why signatures are only compared with respect to their respective sets of communication channels. In the sequel, signature inclusions will be denoted[1] as $\rho : \Sigma_1 \subseteq \Sigma_2$ or simply ρ.

Example 1. Let us introduce a $IOSTS$-signature $\Sigma_1 = (A_1, C_1)$ to specify a drink machine:

• $A_1 = \{coin, m, price, B\}$ where the *coin* variable will denote the value of the coin introduced by the user, m the value of the available amount to be spent, *price* the price of the beverages, B the selected beverage.

[1] Clearly, signatures and signature inclusions constitute a category.

• $C_1 = \{introduce, select, screen, refund, serve, take\_cup\}$ where *introduce* allows the user to introduce coins, *select* denotes the button used to select a beverage, *screen* the place where some messages are displayed, *refund* the way to give back money in excess, *serve* the fact that the cup is filled with the beverage and lastly, *take_cup* the fact that the user is taking off his beverage.

An *IOSTS* communicates through communication actions consisting in receipts (inputs) and emissions (outputs) of values through channels.

Definition 2 (Actions). *The* set of communication actions*, denoted* $Act_\Sigma = Input(\Sigma) \cup Output(\Sigma)$ *where:*
$Input(\Sigma) = \{(c, ?, y) \mid c \in C, y \in A\} \quad Output(\Sigma) = \{(c, !, t) \mid c \in C, t \in T_\Sigma(A)\}$

In the sequel we will note $c?y$ for $(c, ?, x)$ and $c!t$ for $(c, !, t)$. Actions are interactions with the environment: $c?x$ represents a receipt of a value from its environment which will be assigned to the attribute variable x. $c!t$ represents the emission of the value t through the channel c. Interactions with no exchange of values (i.e. pure signals) on a channel c are conventionally modelled by $c!\top$ or $c?x_\top$ with $x_\top$ a variable reserved for that purpose, and simply written resp. as $c!$ or $c?$ in the sequel.

Definition 3 (Observable traces). *An observable trace* r *over* Σ *is a finite sequence of observations belonging to* $Obs_\Sigma = (C \times \{?, !\} \times M)$*. We note* $ObsTr(\Sigma)$ *the set of observable traces*[2] *over* Σ*.*

Let us consider the signature inclusion $\rho : \Sigma_1 \subseteq \Sigma_2$*. Given an observable trace* r *over* Σ_2*, the projection of* r *on* Σ_1*, denoted* $r_{|\rho}$*, or simply* $r_{|\Sigma_1}$*, is the observable trace over* Σ_1 *obtained by removing from* r *observations not belonging to* Obs_{Σ_1}*: thus, if* r *is decomposed as "*$e\ r'$*" with* e *an observation of* Obs_{Σ_2} *and* r' *the ending trace, then* $r_{|\rho} = e\ r'_{|\rho}$ *if* e *belongs to* Obs_{Σ_1}*, else* $r'_{|\rho}$*.*

Observable traces represent observations which can be done on a IOSTS: they give which values are exchanged, as emissions or receipts, with the environment, and according to which order. Projections of traces on a subsignature allow us to restrict the signature to be considered as exported, and thus as observable.

Definition 4 (*IOSTS*). *An IOSTS over a signature* $\Sigma = (A, C)$ *is a 4-tuple* $(Q, q_0, Trans, \iota)$ *where* Q *is a set of* state names*,* $q_0 \in Q$ *is the* initial state*,* $Trans \subseteq Q \times Act_\Sigma \times Sen_\Omega(A) \times T_\Omega(A)^A \times Q$ *and* ι *is a substitution associating to each attribute of* A *a term in*[3] $T_\Omega(V \cup A)$*. A transition* $tr = (q, act, \varphi, \rho, q')$ *of* $Trans$ *is composed of a source state* q *denoted by* $source(tr)$*, an action act denoted by* $act(tr)$*, a guard* φ*, a substitution of variables* ρ *and a target state* q' *denoted by* $target(tr)$*. For each state* $q \in Q$*, there is a finite number of transitions of source state* q*.*

An IOSTS over the signature Σ *is said to be initialized if for every attribute variable* v *in* A*,* $\iota(v)$ *is a ground term of* T_Ω*.*

[2] In the sequel, for an observable trace r, we will denote $r[n]$ the nth element of the trace when it exists.

[3] V is any set of variables disjoint with the set A.

Using initialized $IOSTS$ allows to precisely specify initial values of the attribute variables in order to restrict the set of admissible initial states. On the contrary, non initialized $IOSTS$ admit several (or maybe all) initial conditions for the attribute variables : in this case, the first communications are often used to restrict the set of acceptable states which are reachable from the initial states.

Example 2. We present an $IOSTS$, denoted Sp_1, in Figure 1. It represents an abstract coffee machine over Σ_1. That machine accepts coins as input from the environment ($introduce?coin$). After inserting coins, there is two possibilities, either the machine is out of order ($screen!''out\ of\ order''$) then, the user is refunded ($refund!m$), or the user selects the drink ($select?B$). Here, there is again three possibilities: there is no cups ($screen!$"$no\ cups$") and the user is refunded, or there is no enough money, then the machine asks more coins ($screen!(price - m)$) with $price$ the price of the drink and m the total amount that has been already introduced by the user), or the machine serves the drink ($serve!B$). Lastly, if the amount introduced is more than the price of the drink, the user receives the difference back.

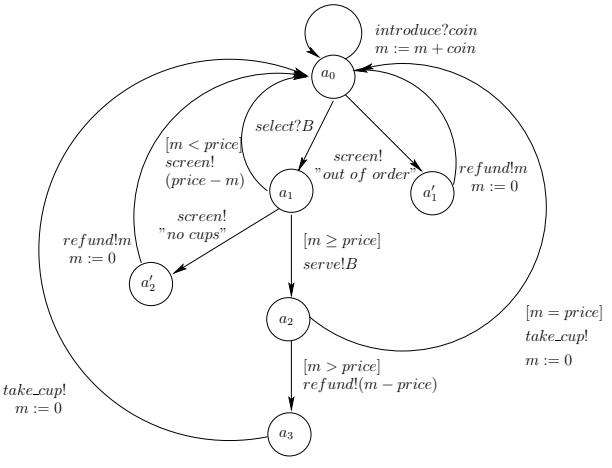

Fig. 1. Specification Sp_1

2.3 Semantics

Definition 5 (Runs of a transition). *Let* $tr = (q, act, \varphi, \rho, q') \in Trans$. *The set* $Run(tr) \subseteq M^A \times Obs_\Sigma \times M^A$ *of* runs *of* tr *is s.t.* $(\nu^i, act_M, \nu^f) \in Run(tr)$ *iff:*

- *if* act *is of the form* $c!t$ *then* $M \models_{\nu^i} \varphi$, $\nu^f = \nu^i \circ \rho$ *and* $act_M = c!\nu^i(t)$,
- *if* act *is of the form* $c?y$ *then* $M \models_{\nu^i} \varphi$, *there exists* ν^a *such that* $\nu^a(z) = \nu^i(z)$ *for all* $z \neq y$, $\nu^f = \nu^a \circ \rho$ *and* $act_M = c?\nu^a(y)$.

For a run $r = (\nu^i, act_M, \nu^f)$, we denote $source(r)$, $obs(r)$ and $target(r)$ respectively ν^i, act_M and ν^f.

Definition 6 (Finite Paths of an *IOSTS*). *Let $G = (Q, q_0, Trans, \iota)$ be an IOSTS over Σ. The set of finite paths in G, denoted $FP(G)$ contains all finite sequences $tr_1 \dots tr_n$ of transitions in $Trans$ s.t. $source(tr_1) = q_0$ and for all $i < n$, $target(tr_i) = source(tr_{i+1})$.*

The runs of a finite path $tr_1 \dots tr_n$ in $FP(G)$ are sequences $r_1 \dots r_n$ such that for all $i \leq n$, r_i is a run of tr_i, there exists an Ω-interpretation ν_1 such that $source(r_1) = \nu_1 \circ \iota$ and for all $i < n$, $target(r_i) = source(r_{i+1})$. The set of observable traces of a finite path *$p = tr_1 \dots tr_n$, denoted $ObsTr(p)$ is the set of finite observation sequences $obs(r_1) \dots obs(r_n)$ for any run $r_1 \cdots r_n$ of p.*

Definition 7. *Let G be an IOSTS over Σ. The* semantics *of G is $ObsTr(G) = \bigcup_{p \in FP(G)} ObsTr(p)$.*

Let $\rho : \Sigma_1 \subseteq \Sigma_2$ be an inclusion signature and G an IOSTS over Σ_2. The semantics of G with respect to ρ is $ObsTr_{|\rho}(G) = \{r_{|\rho} \mid r \in ObsTr(G)\}$.

3 Refinement

3.1 Definition

The refinement relation between IOSTS allows the specifier to relate an IOSTS specification Sp_1 defined over a signature Σ_1 to a more concrete one, Sp_2, in a formal way. Intuitively, Sp_2 should not only include all behaviors of the abstract specification Sp_1, but also may incorporate some specific behaviors that the specifier could not have anticipated at the abstract level. In particular, Sp_2 may involve some concrete actions, emissions or receipts on some new channels, that are not previously known at the abstract level. Such a point of view is similar to the refinement relation given in [8] in the framework of interface automata: the set of legal inputs of the concrete specification (or implementation) may strictly contain the one of the abstract specification. In our setting, we require that all the behaviors of Sp_1 are preserved by SP_2. Obviously, all the behaviors of Sp_1 are given by its semantics: they simply correspond to the set of observable traces of Sp_1. Thus, to refine Sp_1, a specification Sp_2 should be defined over a signature Σ_2 including Σ_1, and should preserve the semantics of Sp_1 in the sense that the semantics of Sp_2 w.r.t. Σ_1 contain the one of Sp_1.

Definition 8 (Refinement). *Let $\rho : \Sigma_1 \subseteq \Sigma_2$ be a signature inclusion. Let Sp_1 and Sp_2 be two IOSTS over Σ_1 and Σ_2 respectively. Sp_2 is a* refinement of *Sp_1, denoted by $Sp_1 \overset{\rho}{\rightsquigarrow} Sp_2$ iff*

$$ObsTr(Sp_1) \subseteq ObsTr_{|\rho}(Sp_2)$$

In the sequel, in the context of a refinement relation $Sp_1 \overset{\rho}{\rightsquigarrow} Sp_2$, the elements of Act_{Σ_2} (resp. Obs_{Σ_2}) expressed on a channel in[4] $C_2 \backslash C_1$ are said to be concrete actions (resp. observations).

3.2 Our Approach for Refinement Testing

As presented in the Introduction, we propose to check if a specification refines another one by following a testing approach. The underlying principle is quite simple. First, we extract an observable trace θ from Sp_1 and then we execute it on Sp_2. During the execution, we check if Sp_2 accepts all observations specified by θ. However, since Sp_2 may involve concrete actions, we have to take them into account during the execution of θ. The difficulty is to manipulate intermediate concrete actions of Sp_2 in a generic way so that we can avoid evaluating concrete actions too early. Indeed, this could limit the execution of θ on Sp_2, or even worse, could forbid its execution though it would be possible with other values. Indeed, blindly choosing some arbitrary values for these intermediate concrete actions can clearly eliminate some possibilities of executing θ on Sp_2 since these particular values can unnecessarily constraint the next execution steps. A convenient way to handle this problem is to use, as inputs, some symbols instead of values to represent any of them. The symbolic execution technique [7] is well adapted to perform this. Such a point of view has already been applied for parameterized unit tests [21]: symbolic execution and constraint solving are advocated to instantiate parameter data according to some unit coverage issues.

3.3 Symbolic Execution

In previous papers [20,12], we have shown that *symbolic execution* [7] is a powerful technique in order to explore the semantics of IOSTS models. As stated in those papers a symbolic execution path can be considered as an *intensional* definition for many concrete executions (or runs): a symbolic execution introduces new fresh variables, also called symbolic inputs, and is characterized by its so-called *path condition* which defines the possible interpretation of the terms involved in the execution path. Obviously, interpretations of all execution paths preserve the IOSTS semantics.

Such symbolic execution paths may be systematically built, or at least with respect to any given arbitrary path length. We can also look for building only symbolic execution paths satisfying some constraint. In particular, we are interested by defining symbolic execution paths matching some particular patterns given as observable traces. As previously explained, for refinement testing, symbolic execution will be exercised on the concrete specification with traces selected from the abstract specification. We will say that such a symbolic execution is constrained by an observable trace. As usual, the main idea is to replace concrete input values and initialization values of attribute variables by symbols and to execute transitions. Substitutions are executed in a natural way. At a given

[4] Given E and F two sets, $E \backslash F$ denotes the set $\{x \in E \mid x \notin F\}$.

step of the execution, encountered guards induce an accessibility constraint on the last constructed state. This constraint is stored in this state as its so-called *path condition*. In the sequel we assume that symbols used as inputs are fresh variables chosen in a set $F = \bigcup_{s \in S} F_s$ disjoint from the set of attribute variables A.

We first give the intermediate definition of *symbolic extended state* which is a structure allowing to store information about a symbolic behaviour: the IOSTS current location (target state of the last transition of the symbolic behaviour), the path condition, the symbolic values associated to attribute variables and a mark given as a natural number.

Definition 9 (Symbolic extended state). *A* symbolic extended state over F *for an IOSTS* $G = (Q, q_0, Trans)$ *is a quadruple* $\eta = (q, \pi, \sigma, n)$ *where* $q \in Q$, $\pi \in Sen_\Omega(F)$ *is called a* path condition, $\sigma \in T_\Omega(F)^A$ *and* n *is a natural number.* $\eta = (q, \pi, \sigma, n)$ *is said to be satisfiable if* π *is satisfiable*[5]. *One notes* $\mathcal{S}$ *(resp.* $\mathcal{S}_{sat}$*) the set of all the (resp. satisfiable) symbolic extended states over* F.

The natural number associated to each symbolic state will serve us to mark them with respect to some external information. In particular, we will use them to synchronise the reading of an abstract observable trace θ over Σ_1 given as a parameter of the symbolic execution of an IOSTS Sp_2 defined over Σ_2 with $\Sigma_1 \subseteq \Sigma_2$. Constraining the symbolic execution of Sp_2 by θ consists in developing all the symbolic executions compatible with θ. For that, all the states will be labelled by a natural number less or equal than k, the length of the trace θ: if a symbolic extended state η is labelled by n, it will simply mean that the n first observations of θ have already been recognized before reaching η and that all other transitions of the corresponding symbolic path concern concrete actions.

Definition 10 (Symbolic execution of an IOSTS constrained by an observable trace). *Let* $\Sigma_1 \subseteq \Sigma_2$ *an inclusion signature. We assume that* $\Sigma_1 = (A_1, C_1)$ *and* $\Sigma_2 = (A_2, C_2)$. *Let* $G = (Q, q_0, Trans, \iota)$ *an IOSTS over* Σ_2. *Let us note* $\Sigma_F = (F, C_2)$. *Let* $\theta \in ObsTr(\Sigma_1)$ *an observable trace of length* k. *The full symbolic execution of* G *constrained by* θ *is a triple* $(\mathcal{S}, init, R)$ *with* $init = (q_0, true, \sigma_0, 0)$ *where* σ_0 *is an injective substitution in* F^A *and* $R \subseteq \mathcal{S} \times Act(\Sigma_F) \times \mathcal{S}$ *such that for any two transitions in* R *respectively of the form* $(\eta^i, c?x, \eta^f)$ *and* $(\eta'^i, d?y, \eta'^f)$, *the variables* x *and* y *are distinct and* $\forall a \in A, \sigma_0(a) \neq x$. *For any* $\eta \in \mathcal{S}$ *of the form* (q, π, σ, n), *for all* $tr \in Trans$ *of the form* $(q, act, \varphi, \rho, q')$, *then there exists a symbolic transition* $st = (\eta, sa, \eta')$ *in* R *iff one of the following conditions detailed below is satisfied:*

- *if* $act = c!t$ *and* $c \notin \Sigma_1$ *then* $sa = c!\sigma(t)$ *and* $\eta' = (q', \pi \wedge \sigma(\varphi), \sigma \circ \rho, n)$,
- *if* $act = c?x$ *with* x *in* A_2 *and* $c \notin \Sigma_1$ *then* $sa = c?z$ *with* z *in* F, *and* $\eta' = (q', \pi \wedge \sigma(\varphi), \sigma \circ (x \mapsto z) \circ \rho, n)$,

[5] Let us recall that here, π is *satisfiable* if and only if there exists $\nu \in M^F$ such that $M \models_\nu \pi$ since variables of π are by construction in F.

- *if* $act = c!t$ *and* $c \in \Sigma_1$ *and* $\theta[n] = c!u$ *then*[6] $sa = c!t_u$ *and* $\eta' = (q', \pi \wedge \sigma(\varphi) \wedge (t = t_u), \sigma \circ \rho), n + 1)$.
- *if* $act = c?x$ *with* x *in* A_2 *and* $c \in \Sigma_1$ *and* $\theta[n] = c?u$, *then* $sa = c?t_u$ *and* $\eta' = (q', \pi \wedge \sigma(\varphi), \sigma \circ (x \mapsto t_u) \circ \rho, n + 1)$,

The symbolic execution of G over F *is the triple* $SE(G) = (\mathcal{S}_{sat}, init, R_{sat})$ *where* R_{sat} *is the restriction of* R *to* $\mathcal{S}_{sat} \times Act(\Sigma_F) \times \mathcal{S}_{sat}$. *It is said consistent if there exits at least a symbolic state of the form* (q, π, σ, k). *Such states are called* terminal.

Let us point out that in the above construction, for the case $act = c!t$ (resp. $c?x$) with $c \in \Sigma_1$, if $\theta[n]$ can be written as $d!u$ with $d \neq c$ or $d'?u$ (resp. $d?y$ with $d \neq c$ or $d'!t$), then no transition is built. It means that when a non compatible observation is encountered, the observable trace θ over Σ_1 cannot be pursued beyond its n^{th} observation.

As previously indicated, the integer constituting the fourth parameter of a symbolic state is used to synchronise observable actions deduced from the symbolic execution with the ones involved in the observable trace θ constraining the execution. From state $(q, \pi, \sigma, n-1)$, if a transition uses the channel involved in the n^{th} element of the trace, we require the compatibility by reinforcing the path condition at the next state. Consequently, if there exists a consistent symbolic state η with the number k, this means that θ as been totally *matched* over a symbolic execution path from the initial state to η. In particular, the symbolic execution only involving concrete actions allows us to retrieve the usual symbolic execution as given in [12], all symbolic states being marked with the natural number 0. Since the natural numbers associated to the symbolic states have been introduced for technical reasons, in the sequel, they are left implicit in the examples. Now we can state the main theorem:

Theorem 1. *Let us consider* Sp_2 *an IOSTS over* Σ_2 *and* Sp_1 *an IOSTS over* Σ_1 *with* $\Sigma_1 \subseteq \Sigma_2$. $Sp_1 \overset{\rho}{\rightsquigarrow} Sp_2 \Longleftrightarrow \forall\ \theta \in ObsTr(Sp_1)$ *the symbolic execution of* Sp_2 *constrained by* θ *is consistent.*

Example 3. Figure 2 illustrates a part of the symbolic execution of the abstract drink machine Sp_1 presented in Figure 1 constrained with the empty path.

4 Refinement Verification by Testing

4.1 Our Approach

Just as for conformance testing our approach consists in executing some observable traces extracted from a specification, Sp_1 on an entity which is supposed to be a realization of this specification. Here the entity under test is also a specification, Sp_2, called the concrete specification. The execution will be naturally performed by means of the symbolic execution constrained by an observable

[6] For a value u of M, t_u denotes a ground term of T_Ω of value u.

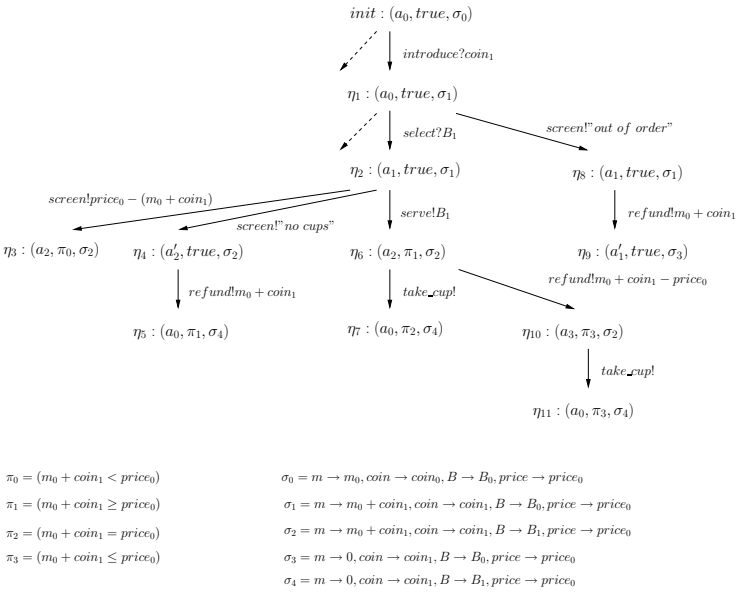

Fig. 2. Symbolic execution of Sp_1

trace. Our approach could be then qualified as *half-symbolic*. One could ask: why not being full symbolic, since Definition 10 could be slightly modified to deal with a full symbolic trace? The main reason for this choice is that industrial users are more familiar with explicit state approaches[7]. Moreover, by choosing to only take observable traces from Sp_1, we ensure that the computed path conditions are expressed on variables of Sp_2 instead of mixing variables of both specifications in the formula. Such mixed formulas would be difficult to analyse.

Now let us notice that Sp_2 may contain loops, involving only concrete actions: the unfolding of those loops during execution may lead to produce paths of an huge size, maybe of an infinite size. To ensure that the computation terminates, we need to define a bound to limit the unfolding of those loops. We decide to allow at most $N (N \in \mathbb{N}^*)$ consecutive occurrences of concrete actions in any path of the symbolic execution tree. Consequently three verdicts, $Warning$, $Fail$, and $Pass$, are necessary to represent the possible conclusions of the execution of a trace θ of Sp_1. The verdict $Warning$ occurs when the execution ends before reaching any terminal state, the bound N has been reached in some paths and all the other states are (implicitly) maximal[8]. In this case we do not know if θ belongs or not to Sp_2. Perhaps with a larger bound we could have found it. The verdict $Fail$ occurs when the execution ends before reaching any terminal state and when all paths are maximal. This means that we are sure that the

[7] "explicit state" means here that variables are instantiated by values.

[8] A path is said to be maximal when any extension has a non satisfiable path condition.

refinement relation is not satisfied since θ does not belong to Sp_2. The verdict $Pass$ occurs when at least a terminal state has been reached. This means that the trace under test belongs to Sp_2 up to the inclusion ρ.

Our algorithm can be described informally as follows. It admits three inputs: an observable trace θ derived from Sp_1, the concrete specification Sp_2 and the bound N. It is a bread-first algorithm which instantiates Definition 10. To take into account the bound N, a parameter, called the distance, denoted by d, is added in the definition of a symbolic extended state. We also add a label $l \in \{stop, go, wrg, rch\}$ (wrg is for *warning* and rch for *reached*). So a symbolic extended state is now of the form $(q, \pi, \sigma, n, d, l)$ with $d = 0$, $l = go$ in the initial state. In an execution step, we consider all states whose label is go. We execute all their outgoing transitions. For a state such that no transition can be executed (because all targets would have an un-satisfiable path condition) its label becomes $stop$. Now a target state obtained by execution satisfies those requirements: if its incoming transition carries an abstract action, its distance parameter is set to 0; if it is a concrete action, the distance is the distance of the source state plus one; if this distance is N then its label is wrg; if $n = length(\theta)$ then its label is rch. The algorithm stops when the set of states labelled by go is empty. If there is at least a state labelled rch the verdict is $Pass$. If leaves of the execution tree are all labelled by $stop$ the verdict is $Fail$. If those leaves are all labelled by wrg, or some by wrg or and others by $stop$ then the verdict is $Warning$. The corresponding path conditions are collected to help the tester to analyse the situation: under which conditions on the concrete variables Sp_2 refines Sp_1 ? Is there a loop in Sp_2 to justify the $Warning$ verdict ?

Example 4. We illustrate the process described above with the following example. Figure 3 represents a concrete coffee machine on Σ_2 where $\Sigma_2 = (A_2, C_2)$ with $A_2 = \{coin, m, price, B, G, ok\}$ and $C_2 = \{introduce, select, screen, refund, serve, take\_cup, error, agent\_put\_cups\}$.

In this drink machine, the attribute variable G represents the number of goblets available in the machine. When a drink is served, one withdraws 1 to the value of the variable G. When G becomes equal to 0, the machine can no more serve a drink and the attribute variable ok is put at $false$ to mean that the machine is out of order ($screen!"out\ of\ order"$). An agent of maintenance can put goblets in the machine ($agent\_puts\_cups?NewG$), or repair the machine. Then he puts the variable ok at $true$ (to mean that the machine is ready again). The number $NewG$ of introduced goblets is added to the variable G.

Let us choose an arbitrary observable trace, denoted t_1, from Sp_1 given in Figure 1), defined on $\Sigma_1 \subseteq \Sigma_2$, introduced in Example 1. The selected observable trace t_1 is the following: $introduce?20\ select?coffee\ screen!"no\ cups"$ $refund!20\ introduce?20\ select?coffee\ serve!coffee\ take\_cup!$

Figure 4 gives the symbolic execution of Sp_2 constrained by t_1. For lack of space, this tree is still partial and does not contain the whole branches : cut branches are represented by dotted transitions.

The first state is $init$. The first observation $introduce?20$ of t_1 is matching with a symbolic transition of Sp_2 issued from $init$, annotated with the symbolic

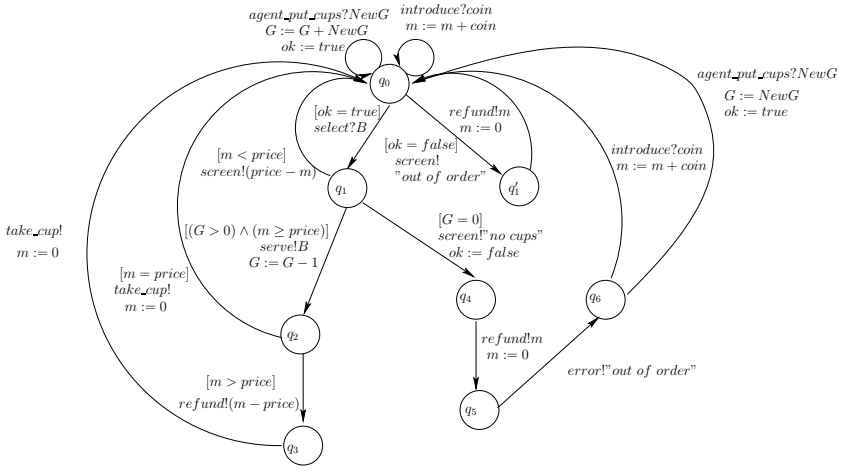

Fig. 3. Concrete specification Sp_2

action *introduce*?*coin*. The matching with the observable trace is required by considering the transition of action *introduce*?20 and adding the constraint $m = m_0 + 20$ in the path condition of the target state η_1. From *init*, there is also a transition (dotted in Figure 4) labelled by $agent\_put\_cups?NewG$ because the action is in Σ_2 and not in Σ_1. Indeed, this represents a hidden concrete action, not observable at the abstract level. The tree construction is pursued and we can recognize two traces including the observable trace t_1, and possibly adding some intermediate concrete actions (as $agent\_put\_cup?NewG_1$ in the right-hand side trace). We can remark that the symbolic state η_5 gives rise two transitions stemming from η_5, respectively with the actions *introduce*?20 and $agent\_put\_cups?NewG_1$. Both branches should be considered in order to search for symbolic states labelled by 8, the length of t_1, meaning that the last action of t_1 has been recognized. For the right-hand side trace, the state η_{12} is labelled by 8, and the associated path condition π_4 gives some sufficient conditions (on the initial values of the attribute variables, denoted by symbolic variables indexed by 0, and on intermediate interaction variables used for hidden concrete actions, here $NewG_1$ for example) under which Sp_2 may refine Sp_1. When applying our algorithm, two cases are thus possible depending on whether the chosen bound N is less or equal to 2 or is strictly greater than 2.

- In the first case, the exploration is stopped before encountering the second *introduce*?20 action of t_1. Indeed, there are two consecutive non observable actions $error!''outoforder''$ and $agent\_puts\_cup?NewG_1$ preceding the next required observable action *introduce*?20. Since the exploration is unfortunately stopped too early, we only get a $Warning$ verdict.
- On the contrary, in the second case (N is strictly greater than 2), we can observe the second *introduce*?20 action of the trace t_1 and pursue the reading of the observable trace in Sp_2 until the last state η_{12} is reached. So, when

Fig. 4. Symbolic execution of Sp_2 constrained by the observable trace t_1

the bound is strictly greater than 2, for the observable trace t_1 selected from Sp_1, we get a $Pass$ verdict for Sp_2 since the path condition π_4 associated to the symbolic state η_{12} is satisfiable.

Example 5. Let us suppose that $t_2 = introduce?20\ select?coffee\ take\_cup!$ would be a second observable trace of Sp_1. In fact, t_2 is not an observable trace of Sp_1 as one may verify it on Figure 1. Let us introduce in Figure 5 the (partial) symbolic execution of Sp_2 constrained by t_2.

The two first actions of t_2 are recognized since the symbolic state η_2 is labelled by the natural number 2. When building the following symbolic states, there are three possible transitions given by Sp_2 respectively with the actions $screen!(price - m)$, $screen!"no\ cups"$ and $serve!B$, but there is no $take\_cup!$ action like in the trace t_2. All the three actions are observable since they are expressed on the signature Σ_1. But they are not compatible with the next action of t_2 to be recognized. Thus, the symbolic execution of Sp_2 constrained by t_2 reveals that t_2 is not an observable trace of Sp_2, up to the inclusion morphism $\Sigma_1 \subseteq \Sigma_2$. So, we get a $Fail$ verdict. Finally, provided that t_2 would be really an observable trace of Sp_1, such a scenario would mean that Sp_2 does not refine Sp_1.

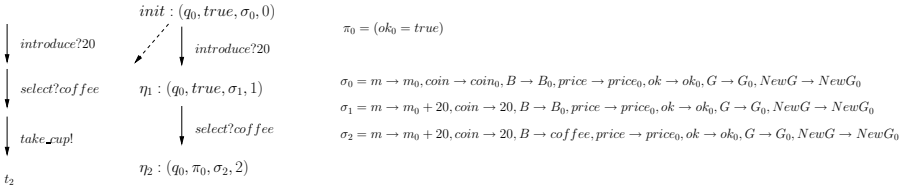

Fig. 5. Symbolic execution of Sp_2 constrained by t_2

4.2 Trace Selection and Implementation Issues

In order to test whether Sp_2 refines Sp_1, the first step is to select some observable traces from Sp_1 to be symbolically executed on Sp_2. Algorithms implementing some classical coverage criteria [25] can be applied to extract traces from Sp_1. Here we propose, like in [17,7], to select among symbolic execution paths. The idea is to compute a finite sub-tree of the symbolic execution of Sp_1 (without being constrained by a given trace). Afterwards, a constraint solver is used at each leaf of this sub-tree to extract an observable trace from the symbolic path ended by the considered leaf. All these observable traces will be executed on Sp_2. The question is how to define this finite sub-tree. In a previous paper [12], we have proposed the so-called *k-inclusion* criterion. The idea is to perform a symbolic execution such that each path carries $k.n$ symbolic transitions[9]. Clearly, this definition depends on a deepness parameter $k.n$. k is a non null integer arbitrarily chosen by the user while n is the result of a calculus. It is the length of the longest path of a symbolic execution reduced by the *inclusion criterion* which, as explained in [12,20], eliminates redundancy in the symbolic execution tree.

The work presented here has been implemented as an extension of the AGATHA tool set [18,20] which as been designed to perform symbolic execution of IOSTS. Presburger arithmetics [19] constitutes the data part of IOSTS treated by AGATHA. The Omega Library [1] has been chosen to handle this data part and is used for two purposes.

5 Conclusion

In this paper, we propose a testing based approach to check whether a concrete specification is a legitimate refinement of an abstract specification. Our approach is based on a combination of concrete and symbolic execution of the specifications. These specifications are described using a first order automata based formalism, namely IOSTS. Our method is strongly inspired from the well-known framework of conformance testing based on the use of test purposes for test case selection. Like conformance testing, some behaviours (observable traces) are selected from the abstract specification Sp_1 by solving a path constraint over

[9] When a path carries less actions, this is because it cannot be extended with non-consistent states.

every execution path of a bounded length from Sp_1. For each observable selected trace θ, the concrete specification Sp_2 is symbolically executed in a way that is constrained by the observable trace θ. This symbolic execution is parameterized by a bound given by the user which controls the number of loop unrollings in Sp_2. It allows us to get a verdict about the refinement relation. Either a counter example is found (verdict $Fail$), a proof is found (verdict $Pass$) or the result remains inconclusive because only a bounded number of loop unrollings has been considered (verdict $Warning$). Contrarily to conformance testing techniques, the execution of selected behaviours is not a black box procedure but a white box procedure based on static analysis of the specification to be tested. The involved static analysis is based on symbolic execution techniques associated to a constraints solver. This white box approach brings the advantage that there is no more the inconclusive verdict related to non-determinism of reactive systems, even if for some cases, the algorithmic limitations do not allow us to fully conclude about the verdict. Moreover, some path conditions are associated to the verdict $Pass$, given information about the appropriate initialisations of the attribute variables of the concrete specification, and about intermediate interaction variables used at the concrete level, and not observable at the abstract level. Such kind of information allows us either to debug a concrete specification detected as not refining the abstract specification or to analyse design choices made by the specifier, for example for reverse-engineering purposes.

In this paper, we perform a blinded exploration of the concrete specification Sp_2 with respect to an observable trace extracted from the abstract specification Sp_1. The given of a bound allows us to arbitrarily stop the exploration between two consecutive observable actions of the trace. Obviously, by making some static analysis of Sp_2, we could get some additional indications on how to appropriately compute a bound of exploration or on how to adapt our algorithm in order to do without a bound. In particular, following the approach developed for Bounded Model Checking in [4], we could try to detect the presence of loops in Sp_2 in order to compute verdicts in the same way bounded model checking is performed. Indeed, the verification process for bounded model checking can terminate when considering finite paths including loops and existential properties. As we look for the existence of a path in Sp_2 with respect to an abstract property (the observable trace), it would be interesting to study if such a finite technique can be transposed in our context in the goal of having less $Warning$ verdicts.

References

1. Omega 1.2. The Omega Project: Algorithms and Frameworks for Analyzing and Transforming Scientific Programs (1994)
2. Abrial, J.-R.: The B book - Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
3. Alur, R., Henzinger, T.A., Kupferman, O., Vardi, M.Y.: Alternating refinement relations. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 163–178. Springer, Heidelberg (1998)

4. Biere, A., Cimatti, A., Clarke, E., Strichman, O., Zhu, Y.: Bounded model checking. In: Highly Dependable Software, vol. 58 of Advances in Computers (2003)
5. Calder, M., Maharaj, S., Shankland, C.: An adequate logic for full lotos. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 384–395. Springer, Heidelberg (2001)
6. Choppy, C., Poizat, P., Royer, J.-C.: A global semantics for views. In: Rus, T. (ed.) AMAST 2000. LNCS, vol. 1816, pp. 165–180. Springer, Heidelberg (2000)
7. Clarke, L.-A.: A system to generate test data and symbolically execute programs. IEEE Transactions on software engineering 2(3), 215–222 (1976)
8. de Alfaro, L., Henzinger, T.A.: Interface automata. In: ESEC/FSE-9. Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 109–120. ACM Press, New York, USA (2001)
9. Derrick, J., Boiten, E.A.: Testing refinements by refining tests. In: Bowen, J.P., Fett, A., Hinchey, M.G. (eds.) ZUM 1998. LNCS, vol. 1493, pp. 265–283. Springer, Heidelberg (1998)
10. Frantzen, L., Tretmans, J., Willemse, T.A.C.: Test generation based on symbolic specifications. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 1–15. Springer, Heidelberg (2005)
11. Frantzen, L., Tretmans, J., Willemse, T.A.C.: A symbolic framework for model-based testing. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) Formal Approaches to Software Testing and Runtime Verification. LNCS, vol. 4262, Springer, Heidelberg (2006)
12. 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, Springer, Heidelberg (2006)
13. Gaudel, M.-C., Bernot, G.: The role of formal specifications. In: Astesiano, E., Kreowski, H.-J., Krieg-Brckner, B. (eds.) Algebraic Foundations of Systems Specification, IFIP State-of-the-Art Report, pp. 1–12. Springer, Heidelberg (1999)
14. Hennessy, M., Lin, H.: Symbolic bisimulations. In: MFPS '92. Selected papers of the meeting on Mathematical foundations of programming semantics, Amsterdam, The Netherlands, pp. 353–389. Elsevier Science Publishers B.V., Amsterdam (1995)
15. Henzinger, T.A., Majumbar, R., Raskin, J.-F.: A classification of symbolic transition systems. ACM Transactions on Computational Logic V, 1–31 (2006)
16. Jeannet, B., Jéron, T., Rusu, V., Zinovieva, E.: Symbolic test selection based on approximate analysis. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, Springer, Heidelberg (2005)
17. King, J.-C.: A new approach to program testing. In: Proceedings of the international conference on Reliable software, Los Angeles, California, vol. 21-23, pp. 228–233 (April 1975)
18. Lugato, D., Rapin, N., Gallois, J.-P.: Verification and tests generation for SDL industrial specifications with the AGATHA toolset. In: Petterson, P., Yovine, S. (eds.) Proceedings of the Workshop on Real-Time Tools affiliated to CONCUR01. Department of Information Technology UPPSALA UNIVERSITY Box 337, August 2001, Sweden, vol. SE-751 05 (2001)
19. Presburger, M.: Über die Vollständigkeit eines gewissen Systems der Arithmetic. Comptes rendus du premier Congres des Math. des Pays Slaves 395, 92–101 (1929)
20. Rapin, N., Gaston, C., Lapitre, A., Gallois, J.-P.: Behavioural unfolding of formal specifications based on communicating automata. In: Proceedings of first Workshop on Automated technology for verification and analysis, Taiwan (2003)

21. Tillman, N., Schulte, W.: Parameterized unit tests. In: 10th European Software Engineering Conference, pp. 253–262. ACM Press, New York (2005)
22. van der Bijl, M., Rensink, A., Tretmans, J.: Action refinement in conformance testing. In: Khendek, F., Dssouli, R. (eds.) TestCom 2005. LNCS, vol. 3502, Springer, Heidelberg (2005)
23. van Glabbeek, R.J., Goltz, U.: Refinement of actions and equivalence notions for concurrent systems. Acta Informatica 37(4/5), 229–327 (2001)
24. Wirth, N.: Program development by stepwise refinement. Commun. ACM 14(4), 221–227 (1971)
25. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Comput. Surv. 29(4), 366–427 (1997)