A model-based approach for robustness testing

Jean-Claude Fernandez, Laurent Mounier, and Cyril Pachon

Verimag, - Centre Equation - 2 avenue de Vignate - F38610 Gieres, France
{Jean-Claude.Fernandez,Laurent.Mounier,Cyril.Pachon}@imag.fr,
http://www-verimag.imag.fr/

Abstract. Robustness testing is a part of the validation process which consists in testing the behavior of a system implementation under exceptional execution conditions in order to check if it still fulfills some robustness requirements. We propose a theoretical framework for model-based robustness testing together with an implementation within the IF validation environment. Robustness test cases are generated from both a (partial) operational specification and an abstract fault model. This generation technique is inspired from the ones used in (classical) conformance testing - already implemented in several tools. This framework is illustrated on a small example.

1 Introduction

Among the numerous techniques available to validate a software system, the purpose of testing is essentially to *find defects* on a *system implementation*. When theoretically founded, testing provides an efficient and rigorous way for error detection. For example, formal methods for *conformance testing* have been largely investigated in the telecomunication area, and the so-called "model based" approach was implemented in several tools (e.g., [4, 15, 9, 7, 1]) and taken into account by the standardization bodies (e.g., ISO standard 9646).

Robustness testing. Informally, robustness can be defined as the ability of a software to keep an "acceptable" behavior, expressed in terms of *robustness requirements*, in spite of *exceptional* or *unforeseen* execution conditions (such as the unavailability of system resources, communication failures, invalid or stressful inputs, etc.). Such a feature is particularly important for software critical applications those execution environment cannot be fully foreseen at development time. Robustness requirements can then range from very general considerations ("there is no run-time error", "there is no system deadlock"), to more specific properties ("after entering a degraded mode, the system always goes back to a nominal one", "some system resources remains always available", etc.).

Even if this kind of testing has been less studied than in the hardware community, several approaches have been proposed to automate software robustness testing. Most of them are based on *fault-injection*, i.e., they consist in feeding the system under test with (sequences of) invalid inputs, chosen within a *fault-model*, and supposed to exhibit robustness failures. However, they differ in the way these inputs are chosen and we review below some of them we consider as the most representatives:

• A first approach consists in generating random inputs obtained by considering only the input domain definition. This technique is particularly adequate to test very large softwares (such an operating system), for which neither a specification nor even the source code is available. Several tools implement this technique. In FUZZ [14], inputs are randomly generated and a failure is detected if the system hangs or dumps a `core` file. In BALLISTA [2], test cases consist of combinations of both valid and invalid inputs focusing on particular parts of the system (e.g., the most frequently used systems calls of an operating system). The verdict distinguishes between several failure criteria (crash, restart, abort, silent, etc.). More recently, the RIDDLE tool [8] uses an input grammar to generate combinations of correct, incorrect and boundary inputs with a better coverage of the system functionalities. It also delivers more precise failure criteria than its predecessors.

• When the source code of the system is available, the generation of relevant inputs to test its robustness can be improved. For instance it becomes possible to use some kinds of static analysis techniques to choose the better inputs able to cover all parameter combinations (w.r.t. an equivalence relation) of public method calls. This idea is exploited for instance in the `JCrasher` testing tool [6], those purpose is to detect undeclared runtime exceptions in Java programs. However this tool only targets a particular kind of faults (unforeseen combinations of parameters in method calls), and issue a rather coarse verdict.

• Finally, some techniques may also rely on some abstract specification of the system behavior to select the test inputs. It is the case for instance in the so-called FOTG approach [11] (Fault Oriented Test Generation): starting from a fault introduced in a protocol specification (like a message loss, or a node crash), it consists in looking (forward) for an error state (a state in which a protocol fails to meet its requirement), and then to search (backward) for a test sequence leading from the initial state to this error state. Even if this approach seems well adapted to fault-tolerant protocols it only deals with single faults (one at a time), and uses a rather simple specification formalism (Finite State Machines). A similar technique has been also proposed in the PROTOS project [3]: it consists in mutating a high-level and abstract description (expressed by a context-free grammar) of the system behavior (the set of correct interactions) to introduce abnormal inputs. Test cases are then generated by performing simulations on this abstract description.

A model-based approach for robustness testing. The objective of this paper is to extend the model-based approach used in conformance testing to the robustness testing framework. However, this extension is not straightforward and it raises the following problems:

• First, robustness is defined with respect to a *fault model* that may vary from an application to another. This element usually depends on the application architecture (e.g., unreliability of some communication links, lack of confidence in some external components, input channel feed by an untrusted user, etc.) and needs to be expressed at a rather abstract level. Note that some of these faults may be controllable by an external tester, whereas some others may not.

• Moreover, specifying the system behavior for *any* exceptional and/or invalid execution conditions expressed by a fault model is (by definition) hard to achieve. Therefore the specification should no longer be considered as "exhaustive" in this context (it may not always reflect the expected behavior of the implementation). In addition, in a real size system, the specification of some components may also be over-approximated (for instance when this specification is partially known, or too complex).

• As a consequence, test verdicts should no longer be based on a conformance relation between the implementation and this (approximated) specification, but directly with respect to the initial robustness requirements.

The solution we propose is based on the following elements: The initial *system specification* is expressed in a formalism those operational semantics can be defined in terms of Input-Outputs labelled transition systems and which explicits the system architecture (communication links attributes, component interface and internal structure). The *fault model* is expressed by *syntactic mutations* performed on this specification. The *robustness requirements* are expressed by a linear temporal logic formula describing the expected behavior of the system implementation in terms of tester interactions. Of course, checking whether a given implementation satisfies or not such a formula should remain decidable during a test (i.e., within a finite amount of time).

Paper outline The paper is organized as follows: first (in section 2) we introduce the models we used, and then (in section 3) we define formally our model-based approach for robustness testing. Section 4 presents an implementation of this technique within the IF environment, and section 5 illustrates its use on an example. We terminate by perspectives and future extensions.

2 Models

In this section, we introduce the models and notations used throughout the paper. The basic models we consider are Input-Output Labelled Transition Systems (IOLTS), namely Labelled Transition Systems in which input and output actions are distinguished (due to of the asymmetrical nature of the testing activity).

2.1 Input-Outputs Labelled Transition Systems

We consider a finite alphabet of actions A, partitioned into two sets: *input actions* A_I and *output actions* A_O. A (finite) IOLTS is a quadruplet $\mathrm{M}=(Q^{\mathrm{M}}, A^{\mathrm{M}}, T^{\mathrm{M}}, q^{\mathrm{M}}_{\mathrm{init}})$ where Q^{M} is the finite set of states, $q^{\mathrm{M}}_{\mathrm{init}}$ is the initial state, $A^{\mathrm{M}} \subseteq A$ is a finite alphabet of actions, and $T^{\mathrm{M}} \subseteq Q^{\mathrm{M}} \times A^{\mathrm{M}} \cup \{\tau\} \times Q^{\mathrm{M}}$ is the transition relation. Internal actions are denoted by the special label $\tau \notin A$. We denote by $\mathbb{N}$ the set of non negative integers. For each set X, $\mathtt{card}(X)$ is the number of element of X. For each set X, X^* (resp. $X^\omega = [X \rightarrow \mathbb{N}]$) denotes the set of finite (resp.

infinite) sequences on X. Let $\sigma \in X^*$; σ_i or $\sigma(i)$ denotes the i^{th} element of σ. We adopt the following notations and conventions: Let $\sigma \in A^*$, $\alpha \in A \cup \{\tau\}$, $p, q \in Q^{\mathrm{M}}$. We write $p \xrightarrow{\alpha}_{\mathrm{M}} q$ iff $(p, \alpha, q) \in T^{\mathrm{M}}$ and $p \xrightarrow{\sigma}_{\mathrm{M}} q$ iff $\exists\ p_0, \cdots, p_n \in Q^{\mathrm{M}}$ such that $p_0 = p$, $p_i \overset{\sigma(i+1)}{\rightarrow}_{\mathrm{M}} p_{i+1}$ for $i < n$, $p_n = q$. In this case, σ is called a *trace* or *execution sequence*, and $p_0 \cdots p_n$ a *run* over σ. An infinite run of M over an infinite execution sequence σ is an infinite sequence ρ of Q^{M} such that 1. $\rho(0) = q^{\mathrm{M}}_{\mathrm{init}}$ and 2. $\rho(i) \overset{\sigma(i)}{\rightarrow}_{\mathrm{M}} \rho(i+1)$). $\mathbf{inf}(\rho)$ denotes the set of symbols from Q^{M} occurring infinitely often in ρ: $\mathbf{inf}(\rho)$=$\{q \mid \forall n.\ \exists i.\ i \geq n \wedge \rho(i) = q\}$. Let V a subset of the alphabet A. We define a *projection operator* $\downarrow_V$: $A^* \rightarrow V^*$ in the following manner: $\epsilon \downarrow_V = \epsilon$, $(a.\sigma) \downarrow_V = \sigma \downarrow_V$ if $a \notin V$, and $(a.\sigma) \downarrow_V = a.(\sigma \downarrow_V)$ if $a \in V$. This operator can be extended to a language L (and we note $L \downarrow V$) by applying it to each sequence of L. The language recognized by M is $\mathcal{L}(M) = \{\sigma \mid \exists \rho$ such that ρ is a run of M over $\sigma\}$. The IOLTS M is *complete* with respect to a set of actions $X \subseteq A$ if and only if for each state q^{M} of Q^{M} and for each action x of X, there is at least one outgoing transition of T^{M} from q^{M} labelled by $x \in X$: $\forall p^{\mathrm{M}} \in Q^{\mathrm{M}} \cdot \forall x \in X \cdot \exists q^{\mathrm{M}} \in Q^{\mathrm{M}}$ such that $p^{\mathrm{M}} \xrightarrow{x}_{\mathrm{M}} q^{\mathrm{M}}$. The IOLTS M is said *deterministic* if and only:

$\forall p^{\mathrm{M}} \in Q^{\mathrm{M}} \cdot \forall a \in A^{\mathrm{M}} \cdot p^{\mathrm{M}} \xrightarrow{a}_{\mathrm{M}} q^{\mathrm{M}} \wedge p^{\mathrm{M}} \xrightarrow{a}_{\mathrm{M}} q'^{\mathrm{M}} \Rightarrow q^{\mathrm{M}} = q'^{\mathrm{M}}$. A state p is said *quiescent* [16] in M either if it has no outgoing transition (deadlock), or if it belongs to a cycle of internal transitions (live-lock). Quiescence can be expressed at the IOLTS level by introducing an extra transition to each quiescent state labelled by a special output symbol δ. Formally, we associate to LTS M its so-called "suspension automaton" $\delta\,(M) = (Q^M, A^M \cup \{\delta\}, T^{\delta(M)}, q_0^M)$ where $T^{\delta(M)} = T^M \cup \{(p, \delta, p) \mid p \in Q^M \wedge p \text{ is quiescent}\}$.

2.2 Specification

We consider specifications, expressed in the IF language [1], consisting of components (called *processes*), running in parallel and interacting either through shared variables or asynchronous signals. Processes describe sequential behaviors including data transformations, communications and process creations/destructions. Furthermore, the behavior of a process may be subject to timing constraints. The behavior of a *process* is described as a (timed) automaton, extended with data. A process has a local memory consisting of variables, control states and a FIFO queue of pending messages (received and not yet consumed). A process can move from one control state to another by executing some *transition.* Notice that several transitions may be enabled at the same time, in which case the choice is made non-deterministically. Transitions can be either *triggered* by signals in the input queue or be *spontaneous.* Transitions can also be *guarded* by predicates on variables. A transition is enabled in a state if its trigger signal is present and its guard evaluates to true.

Transition *bodies* are *sequential programs* consisting of elementary actions (variable assignments, message sending, process creation/destruction, etc) and

[1] http://www-verimag.imag.fr/ async/IF/index.shtml.en

structured using elementary control-flow statements (like if-then-else, while-do, etc). In addition, transition bodies can use external functions/procedures, written in an external programming language (C/C++). *Signals* are typed and can have data parameters. Signals can be addressed directly to a process (using its *pid*) and/or to a signal route which will deliver it to one or more processes. The destination process stores received signals in a FIFO buffer. *Signal routes* represent specialized communication media transporting signals between processes. The behavior of a signal route is defined by its connection policy (peer to peer, unicast or multi cast), and finally its reliability ("reliable" or "lossy"). We use below a simplified abstract syntax and we give its corresponding (informal) semantics in terms of IOLTS.

Definition 1 (specification syntax).

A specification SP is a tuple (S, C, P) where S is the set of signals, $C = C^{int} \cup C^{ext}$ is the set of queues (internal and external ones) and P is the set of processes. The external queues describe the interface between the specified system and its environment. A process $p \in P$ is a tuple (X_p, Q_p, T_p, q_p^0) where X_p is a set of local typed variables, Q_p is a set of states, Σ_p is a set of guarded commands which can be performed by p, and $T_p \subseteq Q_p \times \Sigma_p \times Q_p$ is a set of transitions. A guarded command has the form $[\ b\ ]\alpha$ where α can be either an assignment $x := e$, an input $c?s(x)$, or an output $c!s(e)$. Above, b and e are expressions, $x \in X_p$ is a variable, $c \in C$ is a queue and $s \in S$ is a signal. The set of types τ_i is partially ordered by the sub-typing relation $\leq_{s.t.}$.

We give the semantics of specifications in terms of labeled transition systems. For each type τ_i, we consider its domain D_i and we denote by D the disjoint union of all these domains. We define variable contexts as being total mappings $\rho : \bigcup_{p \in P} X_p \to D$ which associate to each variable x a value v from its domain. We extend these mappings to expressions in the usual way. We define internal queue contexts as being also total mappings $\delta : C^{int} \to (S \times D)^*$ which associates to each internal queue c a sequence $(s_1, v_1), ..., (s_k, v_k)$ of messages, that is pairs (s, v) noted also by $s(v)$, where s is a signal and v is the carried parameter value.

Definition 2 (specification semantics).

The semantics of a specification SP is given by a labeled transition system S=$(Q^s, A^s, T^s, q^s_{init})$. States of this system are configurations of the form (ρ, δ, π), where ρ is a variable context, δ is a queue context and $\pi = \langle q_1, ...q_n \rangle \in \times_{p \in P} Q_p$ is a global control state. Transitions are either internal (and labeled with τ), when derived from assignments or internal communication, or visible when derived from external communication. There is a transition from a configuration (ρ, δ, π) to (ρ', δ', π') iff there is a transition $q_p \xrightarrow{[b]\alpha} q'_p$ in the specification such that the guard b is evaluated to true in the environment ρ. The set of actions is partitioned into A^s_I and A^s_O where
$A^s_I = \{c?s(v) \in A^s, c \in C^{ext}, v \in D\}$ *and* $A^s_O = \{c!s(v) \in A^s, c \in C^{ext}, v \in D\}$

2.3 Mutation

The abstract fault model we consider consists in a mutation function defined on the specification syntax. Formally, let (S, C, P) be a specification. A *fault model* is a function that transforms (S, C, P) into (S', C', P'). We give hereafter a (non exhaustive) set of possible transformations. Note that each transformation corresponds to a *fault* that can be produced by an external tester.

- Domain extension for a variable. For a process i, if an input signal has a parameter of type t_i, then we can extend t_i in t'_i where $t_i \leq t'_i$.
- Unreliable channel In a process i, each transition corresponding to an output on a channel c $(p_i \xrightarrow{[b]\ c!s(e)} q_i)$ is "duplicated" into an internal transition $(p_i \xrightarrow{\tau} q_i)$. At the IF level, this transformation simply consists in replacing a reliable channel by a lossy one.
- Input failure In a process i, if a state has only input entries, then we add a new transition from this state, labelled by τ and leading to a sink state.

3 Robustness Testing Framework

In this section we propose a formal framework to test the "robustness" of a software implementation with respect to a set of robustness requirements.

3.1 Robustness requirements and satisfiability relation

A robustness requirement aims at ensuring that the software will preserve an "acceptable behavior" under non nominal execution conditions. This notion of "acceptable behavior" may not only correspond to safety properties (telling that something bad never happens), but also to liveness properties (telling that something good will eventually happen). Liveness properties are characterized by *infinite* execution sequences. From the test point of view, only the existence of a finite execution sequence can be checked on a given IUT (since the test execution time has to remain bounded). This restricts in practice the test activity to the validation of safety properties. Nevertheless, an interesting sub-class of safety properties are the so-called *parameterized liveness*. Such properties allow for instance to a express that the IUT will exhibit a particular behavior within a given amount of time, or before a given number of iterations has been reached. From a practical point of view, it is very useful to express such properties as liveness (i.e., in terms of infinite execution sequences), and then to bound their execution only at test time, depending on the concrete test conditions.

Robustness requirements. In the approach we propose, a robustness requirement φ is directly modelled by an *observer automaton* $O_{\neg\varphi}$ recognizing all (infinite) executions sequences satisfying $\neg\varphi$. Several acceptance conditions (B*ü*chi, Muller, Streett, Rabin, etc) have been proposed to extend finite-state IOLTS to recognize infinite sequences. For algorithmic considerations, it is more efficient to consider a *deterministic* observer. Since any ω-regular languages can be recognized by a deterministic Rabin automaton [2], we choose this kind of acceptance condition to model our robustness requirements. First we recall the definition of a Rabin automaton.

Definition 3. *A Rabin automaton is a pair $(B, \mathcal{T})$ where B= $(Q^B, A, T^B, q^B_{init})$ is an IOLTS and $\mathcal{T} = \{(L_1, U_1), \cdots, (L_n, U_n)\}$ is a set of couple of subsets of Q^B. The language accepted by $(B, \mathcal{T})$ is $\mathcal{L}(B, \mathcal{T}) = \{\sigma \in A^\omega \mid \exists i.$
$\exists$ an infinite run ρ of B over σ such that $\mathbf{inf}(\rho) \cap L_i \neq \emptyset$ and $\mathbf{Inf}(\rho) \cap U_i = \emptyset\}$.*

Clearly, deciding if an execution sequence σ belongs or not to $\mathcal{L}(B, \mathcal{T})$ cannot be performed during a finite test execution. Therefore, this definition needs to be refined in order to approximate $\mathcal{L}(B, \mathcal{T})$ as a set of *finite* execution sequences. The solution we propose is to associate parameters (c_l, c_u) to each pair (L, U) of $\mathcal{T}$ in order to "bound" the acceptance condition. This notion of "parameterized" Rabin automaton is formalized in the following definition.

Definition 4. *Let $(B, \mathcal{T})$ be a Rabin automaton, $\mathcal{C} = \{(c_{l_1}, c_{u_1}), \cdots, (c_{l_n}, c_{u_n})\}$ a set of integer pairs. We define $\mathbf{inf}_a(\rho, n) = \{q \mid \mathtt{card}(\{i \mid \ \rho(i) = q\}) \geq n\}$.*

*The language accepted by the parameterized Rabin automaton $(B, \mathcal{T}, \mathcal{C})$ is then: $\mathcal{L}(B, \mathcal{T}, \mathcal{C}) = \{\sigma \in A^* \mid \exists i.$
$\exists$ a run ρ of B over σ such that $\mathbf{inf}_a(\rho, c_{l_i}) \cap L_i \neq \emptyset$ and $\mathbf{Inf}_a(\rho, c_{u_i}) \cap U_i = \emptyset\}$.*

Implementation. The Implementation Under Test (IUT) is assumed to be a "black box" those behavior is known by the environment only through a restricted interface (a set of inputs and outputs). From a theoretical point of view, this behavior can be considered as an IOLTS IUT=$(Q^{\text{IUT}}, A^{\text{IUT}}, T^{\text{IUT}}, q^{\text{IUT}}_{\text{init}})$, where $A^{\text{IUT}} = A^{\text{IUT}}_I \cup A^{\text{IUT}}_O$ is the IUT interface. We assume in addition that this IUT is complete with respect to A_I (it never refuses an unexpected input).

Satisfiability relation. We are now able to formalize the notion of robustness of an implementation with respect to a robustness requirement φ.

Definition 5. *Let $\mathcal{I}$ be an IOLTS, φ a formula interpreted over execution sequences of $\mathcal{I}$. An* observer *$\mathcal{O}_{\neg\varphi} = (\mathcal{O}, \mathcal{T}^O, \mathcal{C}^O)$ for φ is a parameterized Rabin automaton such that IOLTS $\mathcal{O}$ is* deterministic, complete *and $\mathcal{L}(\mathcal{O}_{\neg\varphi})$ is the set of execution sequences verifying $\neg\varphi$. We say that $\mathcal{I}$* satisfies *φ iff $\mathcal{L}(\mathcal{I}) \cap \mathcal{L}(\mathcal{O}_{\neg\varphi}) = \emptyset$.*

3.2 Test architecture and test cases

Test Architecture. At the abstract level we consider, a test architecture is simply a pair $(\mathcal{A}_c, \mathcal{A}_u)$ of actions sets, each of them being a subset of $\mathcal{A}$: the set of *controllable* actions $\mathcal{A}_c$, initiated by the tester, and the set of *observable* actions $\mathcal{A}_u$, observed by the tester. A test architecture will be said *compliant* with an observer $\mathcal{O}$ those action set is $A^{\mathcal{O}} = A^{\mathcal{O}}_O \cup A^{\mathcal{O}}_I$ iff it satisfies the following constraints : $A^{\mathcal{O}}_I \subseteq \mathcal{A}_c$ and $A^{\mathcal{O}}_O \subseteq \mathcal{A}_u$. In other words the tester is able to control (resp. observe) all inputs (resp. outputs) appearing in the observer.

[2] which is not the case for deterministic Büchi automata

Test Cases. Intuitively, a test case $\mathcal{TC}$ for a robustness requirement φ is a set of execution sequences, controllable, compatible with a given test architecture and accepted by an observer $O_{\neg\varphi}$. This notion can be formalized as parameterized Rabin automaton.

Definition 6. *For a given observer $\mathcal{O}$ those action set is $A^{\mathcal{O}}$, a test architecture (A_c, A_u) compliant with $\mathcal{O}$,* a test case $\mathcal{TC}$ *is a parameterized Rabin automaton (TC,$\mathcal{T}^{TC}$,$\mathcal{C}^{TC}$) with TC=(Q^{TC}, A^{TC}, T^{TC}, q^{TC}_{init}) satisfying the following requirements:*

1. $A^{TC} = A_I^{TC} \cup A_O^{TC}$ *with* $A_O^{TC} \subseteq A_c$ *and* $A_I^{TC} \subseteq A_u$.
2. *TC is deterministic wrt A^{TC}, controllable (for each state of Q^{TC} there is at most one outgoing transition labelled by an action of A_c), and input-complete (for each state of Q^{TC}, for each element a of A_u, there exists exactly one outgoing transition labelled by a).*
3. $\mathcal{L}(TC) \downarrow A^{\mathcal{O}} \subseteq \mathcal{L}(\mathcal{O})$

3.3 Test cases execution and verdicts

A test case $\mathcal{TC}$ for a robustness requirement φ is supposed to be executed against an IUT by a tester. This IUT is then declared *non robust* (for φ) if such a test execution exhibits an execution sequence of the IUT that belongs to $\mathcal{L}(\mathcal{TC})$ (in other words if $\mathcal{L}(\mathrm{TC}) \cap \mathcal{L}(\mathrm{IUT}) \neq \emptyset$). In this case the tester should issue a **Fail** verdict, and it should issue a **Pass** verdict otherwise.

Test execution. More formally, Let IUT=(Q^{IUT}, A^{IUT}, T^{IUT}, $q^{\mathrm{IUT}}_{\mathrm{init}}$) an implementation, (TC,$\mathcal{T}^{\mathrm{TC}}$,$\mathcal{C}^{\mathrm{TC}}$) a test case with TC=($Q^{\mathrm{TC}}$, A^{TC}, T^{TC}, $q^{\mathrm{TC}}_{\mathrm{init}}$), and ($A_c$, A_u) a test architecture. The test execution of TC on IUT can be expressed as a parallel composition between IUT and TC with synchronizations on action sets A_c and A_u. This test execution can be described by an IOLTS $\mathcal{E}$=($Q^{\mathcal{E}}$, $A^{\mathcal{E}}$, $T^{\mathcal{E}}$, $q^{\mathcal{E}}_{\mathrm{init}}$), where $A^{\mathcal{E}} = A^{\mathrm{TC}}$, and sets $Q^{\mathcal{E}}$ and T^{TC} are defined as follows:

• $Q^{\mathcal{E}}$ is a set of *configurations*. A *configuration* is a triplet $(p^{\mathrm{TC}}, p^{\mathrm{IUT}}, \lambda)$ where $p^{\mathrm{TC}} \in Q^{\mathrm{TC}}, p^{\mathrm{IUT}} \in Q^{\mathrm{IUT}}$ and λ is a partial function from Q^{TC} to $\mathbb{N}$, which counts the number of times an execution sequence visits a state.

• $T^{\mathcal{E}}$ is the set of transitions $(p^{\mathrm{TC}}, p^{\mathrm{IUT}}, \lambda) \xrightarrow{a}_{\mathcal{E}} (q^{\mathrm{TC}}, q^{\mathrm{IUT}}, \lambda')$ such that

– $p^{\mathrm{TC}} \xrightarrow{a}_{\mathrm{TC}} q^{\mathrm{TC}}$, $p^{\mathrm{IUT}} \xrightarrow{a}_{\mathrm{IUT}} q^{\mathrm{IUT}}$ and

$$– \ \lambda'(q^{\mathrm{TC}}) = \begin{cases} \lambda(q^{\mathrm{TC}}) & \text{if } q^{\mathrm{TC}} \notin \bigcup\limits_{i \in \{1,\cdots k\}} (L_i^{\mathrm{TC}} \cup U_i^{\mathrm{TC}}) \\ \lambda(q^{\mathrm{TC}}) + 1 & \text{if } q^{\mathrm{TC}} \in \bigcup\limits_{i \in \{1,\cdots k\}} (L_i^{\mathrm{TC}} \cup U_i^{\mathrm{TC}}) \end{cases}$$

The initial configuration $q^{\mathrm{TC}}_{\mathrm{init}}$ is $(q^{\mathrm{TC}}_{\mathrm{init}}, q^{\mathrm{IUT}}_{\mathrm{init}}, \lambda_{\mathrm{init}})$, where for all q, $\lambda_{\mathrm{init}}(q) = 0$.

$T^{\mathcal{E}}$ describes the interactions between the IUT and the test case. Each counter associated with a state of $L_i^{\mathrm{TC}} \cup U_i^{\mathrm{TC}}$ is incremented when an execution sequence visits this state.

Verdicts. Test execution is supposed to deliver some *verdicts* to indicate whether the IUT was found robust or not. These verdicts can be formalized as a function **Verdict** on execution sequences of $\mathcal{E}$ to the set $\{\textbf{Pass}, \textbf{Fail}\}$. More precisely:

- **Verdict**(σ) = **Fail** if there exists a run ρ of $\mathcal{E}$ over σ, $i \in \{1, 2, \ldots, k\}$ and $l \in \mathbb{N}$ such that:
  1. $\rho(l) = (p_l^{\text{TC}}, p_l^{\text{IUT}}, \lambda_l)$, $p_l^{\text{TC}} \in L_i^{\text{TC}}$ and $\lambda_l(p_l^{\text{TC}}) \geq c_{l_i}$, and
  2. $\forall m \in [0 \cdots l].\rho(m) = (q_m^{\text{TC}}, q_m^{\text{IUT}}, \lambda_m) \wedge q_m^{\text{TC}} \in U_i^{\text{TC}} \Longrightarrow \lambda_m(q_m^{\text{TC}}) \leq c_{u_i}$.
- **Verdict**(σ) = **Pass** otherwise.

In practice the test case execution can be performed as follows:

• At each step of the execution the controllability condition may give a choice between a controllable and an observable action. In this situation the tester can first wait for the observable action to occur (using a local timer), and then choose to execute the controllable one.

• Formal parameters $\mathcal{C}^{\text{TC}}$ are instantiated according to the actual test environment. A **Fail** verdict is issued as soon as an incorrect execution sequence is reached (according to definition above), and a **Pass** verdict is issued either if the current execution sequence visits "too many often" a state of U_i^{TC} ($\lambda_m(q_m^{\text{TC}}) > cu_i$), or if a global timer, started at the beginning of test execution, expires. This last case occurs when an execution sequence enters a loop without any state belonging to L_i^{TC} or U_i^{TC}.

3.4 Test graph

Intuitively, the purpose of a test graph TG is to gather a set of execution sequences, computed from a (mutated) specification S and an observer $\mathcal{O}$, defined over a test architecture TA, and belonging to $\mathcal{L}(\mathcal{O})$. The test graph is defined below by computing an asymmetric product $\otimes$ between S and $\mathcal{O}$.

Definition 7. *Let TA = (A_c, A_u) a test architecture, S_0 a specification and S=$(Q^s, A^s, T^s, q^s_{init})$ its deterministic suspension automaton with $A^s \subseteq A_c \cup A_u$. Let $(\mathcal{O}, \mathcal{T}^{\mathcal{O}}, \mathcal{C}^{\mathcal{O}})$ be an observer with $\mathcal{O}$=$(Q^o, A^o, T^o, q^o_{init})$ and*

$\mathcal{T}^{\mathcal{O}} = \langle (L_1^o, U_1^o), (L_2^o, U_2^o), \ldots, (L_k^o, U_k^o) \rangle$ such that TA is compliant *with $\mathcal{O}$. We define the Parameterized Rabin automaton $(TG, \mathcal{T}^{TG}, \mathcal{C}^{TG})$ where*

TG=$(Q^{TG}, A^{TG}, T^{TG}, q^{TG}_{init})$, such that $Q^{TG} \subseteq Q^S \times Q^{\mathcal{O}}$, $A^{TG} \subseteq A^S$, $q_0^{TG} = (q_0^S, q_0^{\mathcal{O}})$, and Q^{TG}, T^{TG} are obtained as follows:

$$(p_S, p_{\mathcal{O}}) \xrightarrow{a}_{T^{\otimes}} (q_S, q_{\mathcal{O}}) \textit{ iff } p_S \xrightarrow{a}_{T^S} q_S \textit{ and } p_{\mathcal{O}} \xrightarrow{a}_{T^{\mathcal{O}}} q_{\mathcal{O}}$$

.

The pair table $\mathcal{T}^{TG}$ is equal to $\langle (L_1^{TG}, U_1^{TG}), (L_2^{TG}, U_2^{TG}), \ldots, (L_k^{TG}, U_k^{TG}) \rangle$ where L_i^{TG} and L_i^{TG} are defined as follows:

$L_i^{TG} = \{(p_S, p_{\mathcal{O}}) \in Q^{TG} \mid q_{\mathcal{O}} \in L_i^{\mathcal{O}}\}$ $\qquad$ $U_i^{TG} = \{(p_S, p_{\mathcal{O}}) \in Q^{TG} \mid q_{\mathcal{O}} \in U_i^{\mathcal{O}}\}$

3.5 Test cases selection

The purpose of the test case selection is to generate a particular test case TC from the test graph TG. Roughly speaking, it consists in "extracting" a sub-graph of TG controllable with respect to the test architecture, and containing a least a sequence of $\mathcal{L}(TG)$ (and hence of $\mathcal{L}(\mathcal{O})$).

Clearly, to belong to $\mathcal{L}(TG)$, an execution sequence of TG has to reach a cycle containing a state belonging to some distinguished set L_i^{TG} (for some i) of the pair table associated to TG. Conversely, any sequence of TG not leading to a strongly connected component of TG containing a state of L_i^{TG} cannot belong to $\mathcal{L}(TG)$. Therefore, we first define on TG the predicate L2L (for *"leads to L"*), to denote the set of states leading to such a strongly connected component:

$$\mathrm{L2L}\,(q) \equiv \exists(q_1, q_2, \omega_1, \omega_2, \omega_3).\ (q \overset{\omega_1}{\Longrightarrow}_{T^{TG}} q_1 \overset{\omega_2}{\Longrightarrow}_{T^{TG}} q_2 \overset{\omega_3}{\Longrightarrow}_{T^{TG}} q_1 \text{ and } \exists i.\ q_2 \in L_i^{\mathrm{TG}})$$

We can now define a sub-graph of TG, controllable, and containing at least a sequence of $\mathcal{L}(\mathcal{O})$. This subset contains all non controllable transitions of T^{TG} (labelled by an element of A_u), and at most one (randomly chosen) controllable transition of T^{TG} leading to a state of L2L when several such transitions exist from a given state of TG. More formally, we introduce a selection function:

$\mathrm{select}\,(T^{TG}) = \{(p, a, q) \in T^{TG} \mid$
$\quad a \in A_u \text{ or } a = \text{ one-of}\,(\{a_i \in A_c \mid p \xrightarrow{a_i}_{T^{TG}} q_i \text{ and } \mathrm{L2L}\,(q_i)\})\}$

Finally, this subset of T^{TG} remains to be extended with all non controllable actions of A_u not explicitly appearing in T^{TG}, to ensure the input completness of the test case. The definition of a test case TC is then the following:

Definition 8. *let $(TG, \mathcal{T}^{TG}, \mathcal{C}^{TG})$ with TG=$(Q^{TG}, A^{TG}, T^{TG}, q_{init}^{TG})$ a test graph and TA = (A_c, A_u) a test architecture. A test case $(TC, \mathcal{T}^{TC}, \mathcal{C}^{TC})$ is a Parameterized Rabin automaton with TC=$(Q^{TC}, A^{TC}, T^{TC}, q_{init}^{TC})$ such that $q_0^{TC} = q_0^{TG}$, $A^{TC} = A^{TG} \cup A_u$, Q^{TC} is the subset of Q^{TG} reachable by T^{TC} from q_0^{TC} and T^{TC} is defined as follows:*

$$T^{TC} = \ \mathit{select}(T^{TG}) \cup \{(p, a, p) \mid a \in A_u \text{ and } \nexists q.\ (p, a, q) \in T^{TG}\}$$

4 Implementation

We present in this section a complete tool chain which automates the generation and execution of robustness test cases for Java programs. This tool chain is built upon the IF validation environment [5], and it integrates some components developed within the AGEDIS project. First we give the overall architecture of this tool chain, and we briefly explain how the main operations described in the previous sections have been implemented. Then we illustrate its use on a running example.

4.1 Platform Architecture

The overall architecture of the tool chain is depicted in figure 1. It is built upon several existing tools: model exploration is performed using the IF simulator integrated into the IF environment, test generation uses some of the algorithmic techniques borrowed from the TGV tool [12], and test execution relies on the SPIDER [10] test engine developed in the AGEDIS project. This platform is dedicated to particular specification and target languages (IF and JAVA), but a similar architecture could be used with other specification formalisms or programming languages. The platform inputs are detailed below.

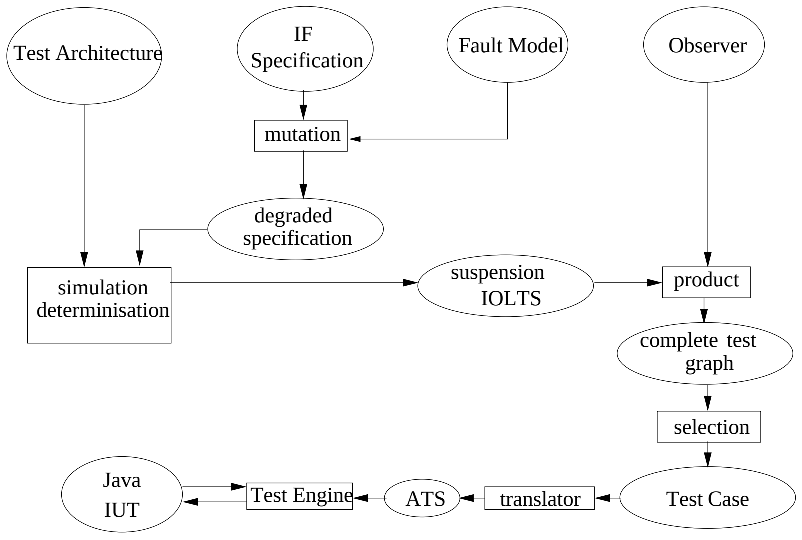

Fig. 1. Platform for the robustness testing

Implementation Under Test. the IUT is a (distributed) multi-threaded Java program, only accessed through a set of public methods (black box IUTs). The corresponding formal model is an IOLTS where input (*resp.* output) actions correspond to method calls (*resp.* return values).

Test architecture. Formally, the test architecture is a pair $(\mathcal{A}_c, \mathcal{A}_u)$ of actions sets (section 3.2). In this particular platform the controllable actions $(\mathcal{A}_c)$ are the set of methods that can be called by an external tester and the observable actions $\mathcal{A}_u$ are the values returned to the tester when these method calls terminate.

Specification. In our context, the specification (partially) describes the expected behavior of the IUT under some nominal execution conditions. It is written using the IF formalism (see section 2.2 for a short description). In practice this IF specification can be automatically produced from high-level specification languages (like SDL or UML).

Fault model. The fault model lists the potential failures and/or incorrect inputs supposed to occur within the actual execution environment. It is directly expressed by a set of syntactic mutations to be performed on the specification.

Observer. The observer is a parameterized Rabin automaton.

4.2 Implementation issues

We now briefly sketch the main operators used in this platform to implement the test generation and test execution technique proposed in this paper. These operators are depicted by square boxes on figure 1.

Mutation. The mutation operation is a purely syntactic operation performed on the abstract syntax tree of the IF specification.

Simulation and Determinisation. This operator produces a deterministic suspension IOLTS from the mutated IF specification. It consists in three steps that are combined *on-the-fly*: *1.* generation of an IOLTS from the mutated IF specification, *2.* computation of the suspension IOLTS ; this step introduces the δ actions, and *3.* determinisation and minimization with respect to the bisimulation equivalence.

Product. This operator computes the test graph from the deterministic suspension IOLTS associated to the mutated specification (S_{Sd}) and the observer (S_{obs}), as defined in section 3.4. It is implemented as a joint traversal of these two IOLTSs.

Test Selection. The test selection operation consists in extracting a test case TC from the complete test graph TG. TC is a parameterized Rabin automaton, controllable, and such that $\mathcal{L}(TC) \subseteq \mathcal{L}(TG)$. Practically this operation is performed in two successive steps (section 3.5):

Computation of state predicate L2L: This computation is based on an algorithm due to R.E. Tarjan to compute in linear time the strongly connected components (SCCs) of TG. When necessary, an SCC can be refined into sub-SCCs to obtain the elementary cycles containing a distinguished state of the test graph.

Computation of function select*:* Once the state predicate L2L has been computed on TG, it remains to extract a sub-graph of TG containing only controllable execution sequences leading to a state satisfying L2L.

Test case Translator and Test Execution Engine Test execution is performed using the SPIDER test engine [10] developed in the AGEDIS project. This tool allows the automatic execution of test cases on multi-threaded (distributed) JAVA programs. Test cases are described in an XML-based format defined within Agedis. Extra test execution directives (supplied by the user) can also be used to map this abstract test case onto the actual implementation interface.

5 Example

We illustrate our approach on a small example describing a simple ticket machine. The system architecture is presented on figure 2. It consists of two components: a *coin tray*, able to store coins received from a user, and a machine

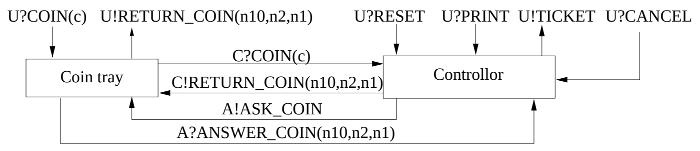

Fig. 2. Ticket machine architecture

controller, managing the interactions with this user. These components communicate each other by message exchanges via two channels C and A. The external user communicates with both components via the channel U.

Under nominal conditions, the expected behavior of the system is the following: the user puts some coins in the *coin tray* (U?COIN(c)), where possible coins values c belong to the set {1,2,10}. The *controller* receives these coins from the *coin tray*, ones by ones, (C?COIN(c)), and increases the user credit. The user can then ask for a ticket (U?PRINT). If his credit is sufficient, a ticket is delivered by the *controller* (U!TICKET), otherwise the machine simply waits for more coins. When a ticket is delivered the machine also needs to return some change to the user. This change is computed according to the coins available in the *coin tray*. To do that, the *controller* asks the *coin tray* about its current contain (A!ASK). The *coin tray* then returns its answer (A!ANSWER(n10, n2, n1)), where n10, n2 and n1 denote the number of coins available in each category. From this information the *controller* can then compute the change (function CompChange()) and ask the *coin tray* to return it to the user (C!RETURN_COIN(...)). Finally, instead of asking for a ticket the user may also choose to cancel the transaction (U?CANCEL) and the machine should then returns to him all the coins he put (C!RETURN_COIN(...)). This specification is formalized on Figure 3 (left, without considering the dashed transitions).

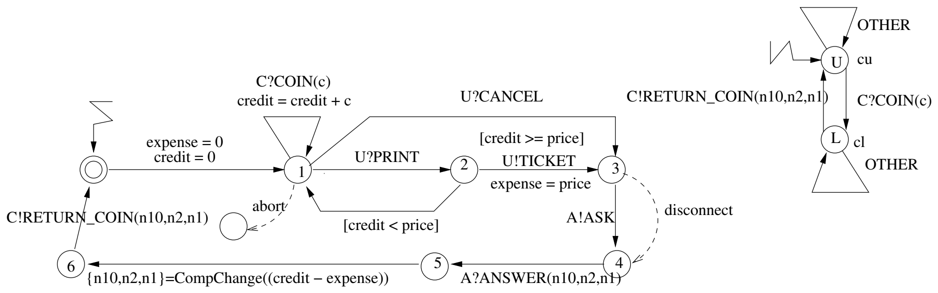

Fig. 3. Controller specification and robustness property.

5.1 Robustness test cases generation

We focus here on the *controller* component and we consider a test architecture where inputs received on channels U and C are controllable, outputs sent on channel C are observable, and communications on channel A are internal. The robustness property we want to ensure is the following: "If the *controller* receives at least one coin (C?COIN(c)), then it *must* output a C!RETURN_COIN action". Figure 3 (right) gives a parameterized Rabin automaton expressing the negation of this property. In this particular example, we assume that in the real execution environment two "faults" may happen: the user may silently stop at any time all interaction with the machine, and communication failures may happen on channel A. Here the mutation operation introduces two new controllable actions (Figure 3, dashed lines): abort, starting from state 1 and leading to a sink state, and disconnect starting from state 3 and leading to state 4. The test generation technique described in the previous section produces the two (parameterized) test cases depicted in figure 4 to invalidate the robustness property. The first one involves a user abortion and the second one a communication failure.

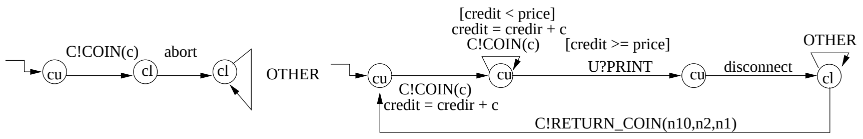

Fig. 4. Possible test cases.

5.2 Implementation and test execution

Two Java implementations of the *controller* have been written. The first one simply reproduces its expected behavior under nominal conditions. Running the above test cases on such an implementation (after instantiation of parameters c_l and c_u) leads to a **Fail** verdict: for both tests there exists a controllable execution sequence for which the limit value of the c_l parameter is reached. The second one (Figure 5) uses a timer T to detect user quiescence and communication failures. In such situations it calls a special function (CompDefChange()) to compute defaults coin values to return back to the user. This implementation is now considered as *robust*: the verdict obtained is **Pass**.

6 Conclusion

The current work extends a model based approach used in conformance testing to the validation of robustness properties. Starting from a (possibly incomplete) system specification it consists in producing a "mutated" specification by applying syntactic transformations described by an abstract fault model. This new specification is model-checked against the robustness requirements to produce diagnostic sequences. These diagnostic sequences are then turned into abstract test cases to be executed on the implementation. Robustness requirements are

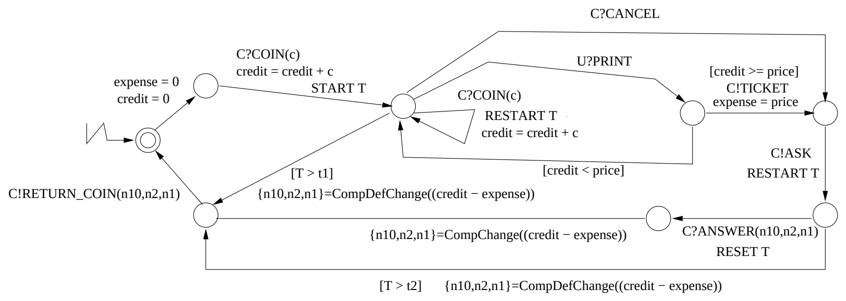

Fig. 5. A "robust" implementation of the controller.

bounded liveness properties expressed by parameterized automata on infinite words. The corresponding test sequences are instantiated at test time to keep the test execution finite. This technique has been implemented inside a complete tool chain (integrating the test generation and test execution phases) and experimented on small Java programs.

Compared to existing robustness testing techniques (based on fault injection), the main advantage of this approach is to much better target the test cases with respect to expected robustness requirements. In particular "faults" are injected by the tester only when necessary, and the verdicts produced are sound (a **Fail** verdict always indicate a violation of a robustness requirement). However, this approach is effective only if there exists some (basic) formal specification of the software under test, describing at least some expected execution scenarios under nominal conditions (like UML use cases, or sequence diagrams).

A first perspective is to improve our implementation to validate this approach on larger case studies. A particular point that would require more investigations is the (static) refinement of the fault model according to a given specification and robustness property. This would allow to consider more accurate mutations and would contribute to limit the state explosion inherent to this kind of approach. Another perspective is to extend this framework to deal with *timed* models [13]. Thus, it would be possible to consider other kinds of faults (stress testing) or properties (response time). The IF specification language already includes a timed model which makes this extension relevant.

Acknowledgement We thank the anonymous reviewers for their many helpful comments. This work was partly initiated in a joint work on *robustness testing* inside a French action supported by the Cnrs and gathering members of Irisa, Laas, Labri, Lri and Verimag laboratories (http://www.laas.fr/TSF/AS23/).

References

1. *The Agedis Project.* http://www.agedis.de.
2. *The Ballista Project.* http://www.ece.cmu.edu/ koopman/ballista/.

3. *The Protos Project*. http://www.ee.oulu.fi/research/ouspg/protos/.
4. A. Belinfante, J. Feenstra, R. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal Test Automation : a Simple Experiment. In *12th International Workshop on Testing of Communicating Systems*, G. Csopaki et S. Dibuz et K. Tarnay, 1999. Kluwer Academic Publishers.
5. M. Bozga, S. Graf, and L. Mounier. If-2.0: A validation environment for component-based real-time systems. In K. L. Ed Brinksma, editor, *Proceedings of CAV'02 (Copenhagen, Denmark)*, volume 2404 of *LNCS*, pages 343–348. Springer-Verlag, July 2002.
6. C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software - Practice and Experiece*, 1(7), 2000.
7. J.-. Fernandez, C. Jard, T. Jéron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In *CAV'96*. LNCS 1102 Springer Verlag, 1996.
8. A. Ghosh, V. Shah, and M. Schmid. An approach for analyzing the Robustness of Windows NT Software. In *Proceedings of the 21st National Information Systems Security Conference*, pages 383–391, Crystal City, VA, 1998.
9. R. Groz, T. Jeron, and A. Kerbrat. Automated test generation from SDL specifications. In R. Dssouli, G. von Bochmann, and Y. Lahav, editors, *SDL'99 The Next Millenium, 9th SDL Forum, Montreal, Quebec*, pages 135–152, Elsevier, Juin 1999.
10. A. Hartman, A. Kirshin, and K. Nagin. A test execution environment running abstract tests for distributed software. In *Proceedings of Software Engineering and Applications, SEA 2002*, Cambridge, MA, USA, November 2002.
11. A. Helmy, D. Estrin, and S. K. S. Gupta. Fault-oriented test generation for multicast routing protocol design. In *Proceedings of the FIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XI) and Protocol Specification, Testing and Verification (PSTV XVIII)*, pages 93–109. Kluwer, B.V., 1998.
12. C. Jard and T. Jron. Tgv: theory, principles and algorithms. In *The Sixth World Conference on Integrated Design & Process Technology (IDPT'02)*, Pasadena, California, USA, June 2002.
13. M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. In *SPIN'04 Workshop on Model Checking Software*, 2004.
14. B. Miller, D. Koscki, C. Lee, V. Maganty, R. Murphy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliabilty of UNIX utilities and services. Technical report, University of Wisconsin, Computer Science Dept., 1995.
15. M. Schmitt, B. Koch, J. Grabowski, and D. Hogrefe. Autolink - A Tool for Automatic and Semi-Automatic Test Generation from SDL Specifications. Technical Report A-98-05, Medical University of Lübeck, 1998.
16. J. Tretmans. Test Generation with Inputs, Outputs, and Quiescence. In T. Margaria and B. Steffen, editors, *Second Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer-Verlag, 1996.