Association of Under-Approximation Techniques for Generating Tests from Models

Pierre Bué, Jacques Julliand, Pierre Masson

▶ To cite this version:

Pierre Bué, Jacques Julliand, Pierre Masson. Association of Under-Approximation Techniques for Generating Tests from Models. TAP'11, 5-th Int. Conf. of Tests and Proofs, 2011, Zurich, Switzerland. pp.51–68. hal-01222515

HAL Id: hal-01222515
https://hal.science/hal-01222515

Submitted on 30 Oct 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Association of Under-Approximation Techniques for Generating Tests From Models

Pierre-Christophe Bué, Jacques Julliand, and Pierre-Alain Masson

LIFC, Université de Franche-Comté
16, route de Gray F-25030 Besançon Cedex France
{bue, julliand, masson}@lifc.univ-fcomte.fr

Abstract. In this paper we present a Model-Based Testing approach with which we generate tests from an abstraction of a source behavioural model. We show a new algorithm that computes the abstraction as an under-approximation of the source model. Our first contribution is to combine two previous approaches proposed by Ball and Pasareanu et al. to compute May, Must+ and Must- abstract transition relations. Proof techniques are used to compute these transition relations. The tests obtained by covering the abstract transitions have to be instantiated from the source model. So, following Pasareanu et al., our algorithm additionally computes a concrete transition relation: the tests obtained as sequences of concrete transitions need not be instantiated from the source model. Another contribution is to propose a choice of relevant parameters and heuristics to pilot the tests computation. We experiment our approach and compare it with a previous approach of ours to compute tests from an abstraction that over-approximates the source model.

Keywords: Model-Based Testing, Abstraction, Over and under-approximations.

1 Motivations

The process of software testing can be automated by means of a Model-Based Testing (MBT) approach [1]. A formal behavioural model is designed from which a set of tests is computed that ensure a given coverage of the model. An adaptation layer fills the gap between the model and the implementation to produce executable tests. The conformance of the implementation to the model is assessed by comparing, modulo the adaptation layer, the outputs of the executable tests with the ones as predicted by the model. A frequent limit to the scalability of this approach is the size of the state space defined by the model: it can be infinite or very large, thus making its coverage impossible in practice.

An abstraction of the behavioural model can be used to overcome this limitation. Abstraction techniques [2–4] allow for making finite or drastically reducing the state space representation of a formal specification (or of a program), for example by gathering into one single *abstract* state several concrete states. Our framework is that of predicate abstraction [2, 3], where the states and the transition relation of the abstraction are defined according to a set of predicates over the model variables.

Once computed from the abstraction, the tests have to be instantiated on the concrete model. This may not be possible for some of the tests if the abstraction is an over-approximation of the source model, thus defining execution paths that may not exist concretely. We have previously defined and presented in [5] a method where this could happen. The abstraction was computed by a theorem-prover that tried to prove the potential feasibility of the transitions rather than their reachability. As a result, time could be spent uselessly to search for an instantiation that does not exist. We adopt another approach in this paper by considering only under-approximations. Since all the paths of an under-approximation exist in the concrete model, it is possible to instantiate every test from it on the concrete model.

We present in this paper a new algorithm, based on previous works from Ball [6] and Păsăreanu et al. [7], to compute abstract transition relations based on predicate abstraction for test generation. Our algorithm combines the two approaches of Ball and Păsăreanu et al., and applies to behavioural models instead of programs. We use SAT-solving techniques to compute the transition relations. We also compute a concrete transition relation and try to directly connect concrete states to each other, so as to obtain concrete tests that need not be instantiated from the model. This complements the tests obtained by covering the abstract transition relation, for which an instantiation is required.

But what part of the behavioural model will be covered by tests computed from an under-approximation of it? We have implemented the algorithm, and used it to compute tests for six case studies. These experimental results are compared in terms of coverage of the abstraction with the ones obtained by our over-approximation method of [5]. We consider a set of optimisations of the concretisation computed by the algorithm, and evaluate experimentally their practical impact on our case studies. We also provide a set of parameters to the user as well as heuristics to improve the method.

The paper is organised as follows. Section 2 presents the process for generating tests from an abstraction. The background required for reading the paper is given in Sec. 3. Two examples to illustrate our approach are described in Sec. 4. The algorithm for computing the abstraction as well as its properties are described in Sec. 5. Section 6 presents the experimental results. We present in Sec. 7 the works to which ours are related. We conclude and indicate future research directions in Sec. 8.

2 Test Generation Process from an Abstraction

In a previous work [5], we have presented a test generation method based on abstraction. The abstraction was computed as an *over-approximation* of a formal behavioural model M of the system, written by a validation engineer. The engineer also wrote a test purpose, by means of a language proposed in [8], to describe how he intended to test the system, according to his know-how. A set of abstraction predicates was automatically deduced from the test purpose, from which the abstraction was computed. The tests issued from the over-approximation had to

be instantiated afterwards on M, which was not always possible since an over-approximation defines more executions than the concrete model. We improve in this paper the test generation method of [5] by computing an abstraction that is an *under-approximation* of M rather than an over-approximation. This guarantees that every test generated from it can be instantiated on M. Also, we now instantiate the tests on-the-fly, and not *a posteriori* as was done in [5].

We sketch our process in Fig. 1. Notice that the abstraction predicates on input of the "Predicate Abstraction and Concretisation" box could be obtained from a test purpose, identically to what was proposed in [5]. We also could synchronise such a test purpose with M before the abstraction computation, to reduce the state space of the resulting abstraction. But we adopt in this paper a more generic presentation of our process. We combine predicate abstraction and SAT-solving for generating the under-approximation. The tests are generated from it by applying a selection criterion that ensures a coverage of the states, of the transitions or of the paths of the abstraction. In our implementation of the process, we have used a chinese postman algorithm to compute tests by covering all the transitions of the abstraction. The tests obtained are valid executions of M, intended to be executed, *via* the adaptation layer, on the implementation of the system.

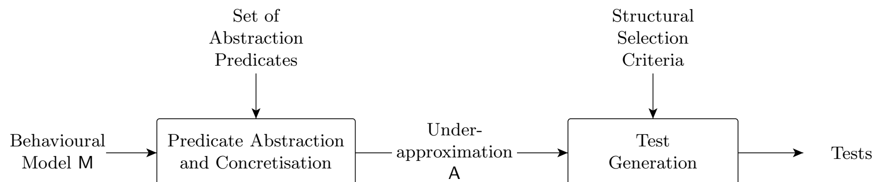


Fig. 1. Process of Test Generation from Test Purpose by Abstraction

3 Background

Our behavioural models are described as B event systems, for which we provide the necessary background. We also present the concept of predicate abstraction and formalise the abstraction notions by means of Symbolic labelled Transition Systems (STS), that define the semantics of the abstractions of event systems.

3.1 Model Syntax

Introduced by J.-R. Abrial [9], a B event system defines a closed specification of a system by a set of events. This syntax is used as an example to present the results in this paper. But these results are generic in the sense that this syntax is a concise form that has the same expressivity as some generic modelling

languages as the guarded actions of [10, 11] or the labelled transition systems. In the sequel, we use the following notations: x, y, z are variables and X, Y, Z are sets of variables. I is an invariant and P is used to denote other predicates. The actions of modification of the variables are called *substitutions* in B, following [12] where the semantics of an assignment is defined as a substitution. In B, substitutions are *generalized*, i.e. the semantics of every kind of action is defined by a substitution calculus that allows to compute the weakest precondition of a substitution a to satisfy the predicate P, denoted $[a]P$ in B. We use a, a_1 and a_2 to denote B generalized substitutions, and E and F to denote B expressions.

All the substitutions allowed in B event systems can be rewritten by means of the five B primitive forms of Def. 1. Notice that the multiple assignment can be generalized to n variables and that all the substitutions terminate.

Definition 1 (Substitution). *The following five substitutions are primitive:*

- *single and multiple assignments, denoted as* $x := E$ *and* $x,\ y := E,\ F$
- *substitution with no effect, denoted as skip*
- *guarded substitution, denoted as* $P \Rightarrow a$
- *bounded non-deterministic choice, denoted as* $a_1 [] a_2$
- *substitution with local variable* z*, denoted as* $@z.a$.

The substitution with local variable is mainly used to express the unbounded non-deterministic choice denoted by $@z.(P \Rightarrow a)$. Let us specify that among the usual structures of specification languages, the conditional substitution IF P THEN a_1 ELSE a_2 END is denoted by $(P \Rightarrow a_1)[](\neg P \Rightarrow a_2)$ with the primitive forms.

Definition 2 defines correct B event systems. The events are defined by an equation $e \mathrel{\widehat{=}} a$ where e is the name of an event and a is a generalized substitutions modifying the state variables.

Definition 2 (Correct Event System). *A correct B event system is a tuple* $\langle X, I, Init, EvDef\rangle$ *where:*

- X *is a set of state variables,*
- I *is an invariant predicate over* X*,*
- *Init is a substitution called* initialisation*, such that the invariant holds in any initial state, i.e.* $[Init]I$ *is valid,*
- *EvDef is a set of event definitions in the shape of* $e \mathrel{\widehat{=}} a$ *such that every event preserves the invariant, i.e.* $I \Rightarrow [a]I$ *is valid.*

3.2 Predicate Abstraction

Predicate abstraction [2] is a special instance of the framework of abstract interpretation [3] that maps the potentially infinite state space C of a transition system onto the finite state space Q of a symbolic transition system *via* a set of predicates $\mathcal{P} \mathrel{\widehat{=}} \{p_1, p_2, \ldots, p_n\}$ over the model variables. A state of C is a valuation of the state variables of the model. The set of abstract states Q contains at most 2^n states. Each state is a tuple $q \mathrel{\widehat{=}} (q_1, q_2, \ldots, q_n)$ with q_i being

equal either to p_i or to $\neg p_i$, and we also consider q as the predicate $\bigwedge_{i=1}^{n} q_i$. We define an abstraction function $\alpha_{\mathcal{P}} : C \rightarrow Q$ such that $\alpha_{\mathcal{P}}(c)$ is an abstract state q where c satisfies q_i for all $i \in 1..n$. By a misuse of language, we say that *c is in q*.

The predicate abstraction is based on five primitive functions denoted as follows. We denote by $wp(a, q')$ the *weakest precondition* [11] of an action a to reach a target state defined by a predicate q'. The weakest precondition $wp(a, q')$ is the largest set of states from which the execution of a necessary leads to a state that satisfies q'. In B, $wp(a, q')$ can be computed by substitution calculus, denoted by $[a]q'$. The weakest conjugate precondition [13], denoted by $wcp(a, q')$ is equal to $\neg[a]\neg q'$ in B, i.e. $\neg wp(a, \neg q')$. The conjugate weakest precondition $wcp(a, q')$ is the largest set of states from which the execution of a can lead in a state that satisfies q'. Since the events always terminate in B, wcp is identical to wp for the deterministic subset of B. In contrast, for a non deterministic substitution, $wp(a, q') \Rightarrow wcp(a, q')$ because there may exist states for which a non deterministic choice leads to q' or not, depending on the chosen branch. These states satisfy $wcp(a, q')$ but not $wp(a, q')$. We denote by $sp(a, q)$ the *strongest postcondition* of an action a from a source state defined by a predicate q. It is the smallest set of states reached by the execution of a from a state that satisfies q. We denote by $grd(a)$ the condition under which the action a is triggerable. It is defined as in B: $grd(skip) \mathrel{\widehat{=}} true$, $grd(x, y := E, F) \mathrel{\widehat{=}} true$, $grd(P \Rightarrow a) \mathrel{\widehat{=}} P \wedge grd(a)$, $grd(a_1 [] a_2) \mathrel{\widehat{=}} grd(a_1) \vee grd(a_2)$, $grd(@z.a) \mathrel{\widehat{=}} \exists z.grd(a)$. Last, we denote by $SAT(P)$ the *satisfiability* value of a predicate P.

Let us now define the abstract transitions as *may-transitions*. Consider two abstract states q and q' and an event e. There exists a *may* transition from q to q' by e, denoted by $q \xrightarrow{e} q'$, if and only if there exists a concrete transition $c \xrightarrow{e} c'$ where c and c' are concrete states with $\alpha_{\mathcal{P}}(c) = q$ and $\alpha_{\mathcal{P}}(c') = q'$. If we assume the event e to be defined by an action a, there is a *may* transition $q \xrightarrow{e} q'$ iff $SAT(wcp(a, q') \wedge q)$.

As in [6], we define two other kinds of abstract transitions: *must-* and *must+*. The *must+* transitions are *may* transitions that are triggerable from all the concrete states of the abstract source state. The *must-* transitions are *may* transitions that reach all the concrete states of the abstract target state. Let $e \mathrel{\widehat{=}} a$ be an event definition. There exists a *must+* transition $q \xrightarrow{e} q'$ iff $q \Rightarrow wp(a, q') \wedge grd(a)$ is valid, i.e. $\neg SAT(\neg(wp(a, q') \wedge grd(a)) \wedge q)$. That is, for any state c concretising the abstract state q, there is a concrete state c' in q' such that $c \xrightarrow{e} c'$. There exists a *must-* transition $q \xrightarrow{e} q'$ iff $q' \Rightarrow sp(a, q)$ is valid, i.e. $\neg SAT(\neg sp(a, q) \wedge q')$. That is for any concrete state c' for which $\alpha_{\mathcal{P}}(c') = q'$, there is a concrete state c for which $\alpha_{\mathcal{P}}(c) = q$ and such that $c \xrightarrow{e} c'$.

3.3 Abstraction Formalisation

We define in Def. 3 a kind of STS well suited to represent abstractions. Definition 4 associates an abstraction defined by an STS to an event system.

Definition 3 (Symbolic Labelled Transition System (STS)). *Let Ev be a finite set of event names. Let A be a finite set of symbolic states on $\mathcal{P}$. Let C be a finite or infinite set of concrete state and $\alpha_{\mathcal{P}}$ an abstraction function from C to A. A tuple $\langle Q, Q_0, C_0, \Delta, \Delta^+, \Delta^-, \Gamma, \Delta^c\rangle$ is an STS if it satisfies the following conditions:*

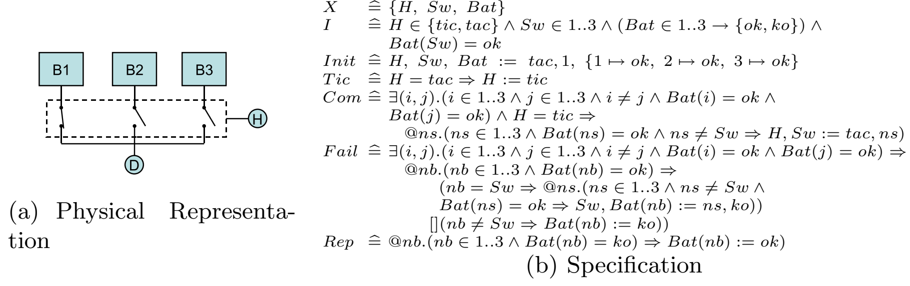
- *$Q(\subseteq \mathsf{A})$ is a finite set of states,*
- *$Q_0(\subseteq Q)$ is a set of abstract initial states,*
- *$C_0(\subseteq C)$ is a set of concrete initial states,*
- *$\Delta(\subseteq Q \times Ev \times Q)$ is a may labelled transition relation,*
- *$\Delta^+(\subseteq Q \times Ev \times Q)$ is a must+ labelled transition relation such that $\Delta^+ \subseteq \Delta$,*
- *$\Delta^-(\subseteq Q \times Ev \times Q)$ is a must- labelled transition relation such that $\Delta^- \subseteq \Delta$,*
- *$\Gamma(\subseteq Q \times C)$ is a concretisation relation that associates concrete states to any abstract state such that $(q, c) \in \Gamma \Rightarrow \alpha_{\mathcal{P}}(c) = q$,*
- *$\Delta^c(\subseteq C \times Ev \times C)$ is a concrete transition relation such that $c \xrightarrow{e} c' \in \Delta^c$ iff $c \in C \wedge c' \in C \wedge q \xrightarrow{e} q' \in \Delta \wedge (q, c) \in \Gamma \wedge (q', c') \in \Gamma$.*

Definition 4 (STS associated to an ES). *Let $ES = \langle X, Init, \{e \mathrel{\widehat{=}} a | e \in Ev\}\rangle$ be an event system and $\mathcal{P} = \{p_1, p_2, ..., p_n\}$ be a set of n predicates defining a set of 2^n abstract states $\mathsf{A} = \{p_1, \neg p_1\} \times \{p_2, \neg p_2\} \times ... \times \{p_n, \neg p_n\}$. A tuple $\langle Q, Q_0, C_0, \Delta, \Delta^+, \Delta^-, \Gamma, \Delta^c\rangle$ is an STS associated to ES and $\mathcal{P}$ if it satisfies the following conditions:*

- *$Q_0 \mathrel{\widehat{=}} \{q | q \in \mathsf{A} \wedge SAT(sp(Init, true) \wedge q)\}$,*
- *$C_0 \mathrel{\widehat{=}} \{c | \exists q_0.(q_0 \in Q_0 \wedge c \in C \wedge c = SAT(wcp(Init, q_0)))\}$,*
- *$\Delta \mathrel{\widehat{=}} \{q \xrightarrow{e} q' | q \in \mathsf{A} \wedge q' \in \mathsf{A} \wedge e \in Ev \wedge SAT(wcp(a, q') \wedge q)\}$,*
- *$Q \mathrel{\widehat{=}} \{q | \exists (q', e).(q \xrightarrow{e} q' \in \Delta \vee q' \xrightarrow{e} q \in \Delta)\}$,*
- *$\Delta^+ \mathrel{\widehat{=}} \{q \xrightarrow{e} q' | q \in \mathsf{A} \wedge q' \in \mathsf{A} \wedge e \in Ev \wedge \neg SAT(\neg(wp(a, q') \wedge grd(a)) \wedge q)\}$,*
- *$\Delta^- \mathrel{\widehat{=}} \{q \xrightarrow{e} q' | q \in \mathsf{A} \wedge q' \in \mathsf{A} \wedge e \in Ev \wedge \neg SAT(\neg sp(a, q) \wedge q')\}$,*
- *$\Delta^c(\subseteq C \times Ev \times C)$ is such that $c \xrightarrow{e} c' \in \Delta^c \Rightarrow SAT(wcp(a, c') \wedge c)$,*
- *$\Gamma \mathrel{\widehat{=}} \{(q, c) | q \in Q \wedge c \in C \wedge \alpha_{\mathcal{P}}(c) = q \wedge \exists (e, c').(c \xrightarrow{e} c' \in \Delta^c \vee c' \xrightarrow{e} c \in \Delta^c)\}$.*

4 Examples

For the sake of readability, let us now introduce two examples that will be used to illustrate our propositions. Each one is intended to illustrate a different point while remaining small and easy to read. The Electrical System example of Sec. 4.1 is a finite state control and command system that illustrates the various kinds of transition relations Δ, Δ^- and Δ^+, as represented in Fig. 3. The example of Sec. 4.2 is a very simple model, but with an infinite state space. Its aim is to illustrate that combining the two under-approximation techniques gives better transition and path coverage than each one separately (see Sec. 5.3, paragraph "Comparison with the other methods").



(a) Physical Representation

(b) Specification

Fig. 2. The Electrical System and its Formal Behavioural Specification

4.1 Electrical System Example

Figure 2(a) shows a device D powered *via* a switch to one of three batteries B_1, B_2, B_3. A clock H periodically sends a signal that causes a commutation of the closed switch. The system has to meet the following requirements: one switch and only one is closed at a time and a clock signal changes the switch that is closed. The batteries may break down. If it happens to the one that is powering D, an exceptional commutation is triggered. We assume that there is always at least one battery working. When there is only one battery working, the clock signals are ignored.

Figure 2(b) models the system by means of three variables. H models the clock and takes two values: *tic* to ask for a commutation and *tac* when it has occurred. Sw models the switches by an integer that indicates which one is closed. Bat models the batteries breakdowns by a total function that associates *ok* or *ko* (for a broken battery) to each battery. The state changes are described by means of four events: **Tic** sends a commutation signal, **Com** changes the closed switch, **Fail** breaks down a battery and **Rep** repairs a battery.

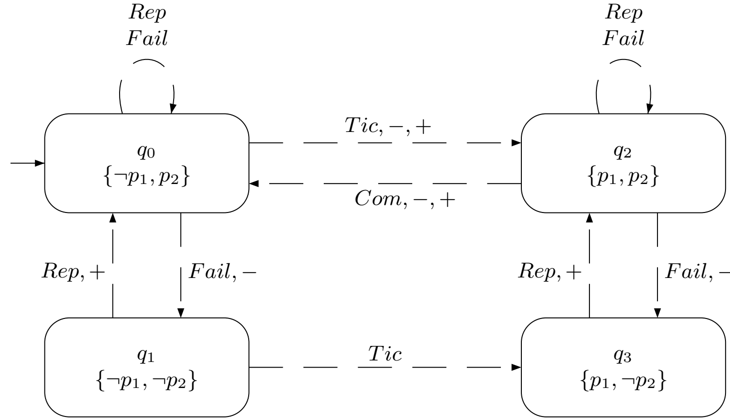


Fig. 3. STS of the Abstract Electrical System

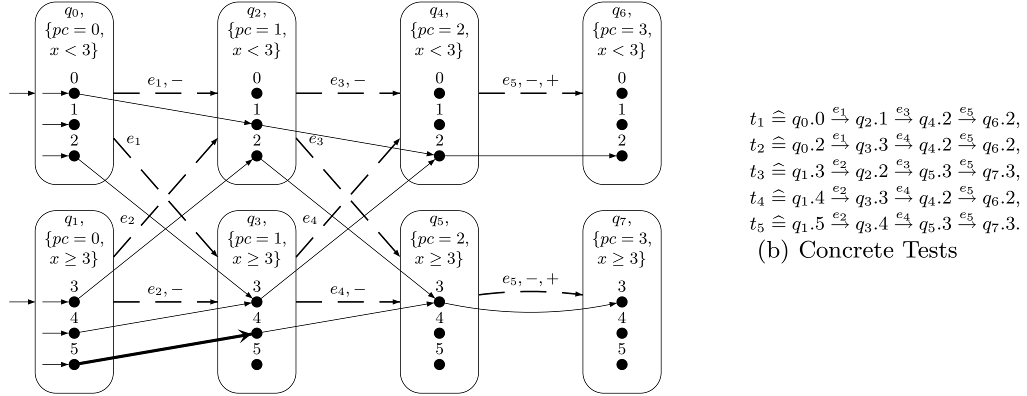
Figure 3 shows the STS without C_0, Γ and Δ^c that abstracts the model of Fig. 2(b) from the set of abstraction predicates $\mathcal{P} \mathrel{\widehat{=}} \{p_1, p_2\}$ where $p_1 \mathrel{\widehat{=}} H = tic$ and $p_2 \mathrel{\widehat{=}} \exists(i, j).(i \in 1..3 \land j \in 1..3 \land i \neq j \land Bat(i) = ok \land Bat(j) = ok)$. All the transitions belong to Δ. Those followed by "-" and/or "+" belong to Δ^- and/or Δ^+. In Fig. 7(a) and Fig. 7(b), we see a fragment of C_0, Γ and Δ^c.

4.2 Simple Illustrative Model

The specification presented in Fig. 4 models a small conditional computation over a variable x. Its semantics is an infinite state transition system for unbounded integers. Our abstraction method computes the finite state symbolic transition system of Fig. 5(a). The variable pc is not abstracted whereas x is abstracted according to the two predicates $x < 3$ and $x \geq 3$.

$$
\begin{array}{ll}
X & \mathrel{\widehat{=}} \{pc,\ x\} \\
I & \mathrel{\widehat{=}} pc \in 0..3 \land x \in \mathbb{Z} \\
Init & \mathrel{\widehat{=}} @z.(x \in \mathbb{Z} \Rightarrow pc, x := 0, z) \\
e_1 & \mathrel{\widehat{=}} pc = 0 \land x < 3 \Rightarrow pc, x := 1, x + 1 \\
e_2 & \mathrel{\widehat{=}} pc = 0 \land x \geq 3 \Rightarrow pc, x := 1, x - 1 \\
e_3 & \mathrel{\widehat{=}} pc = 1 \land x < 3 \Rightarrow pc, x := 2, x + 1 \\
e_4 & \mathrel{\widehat{=}} pc = 1 \land x \geq 3 \Rightarrow pc, x := 2, x - 1 \\
e_5 & \mathrel{\widehat{=}} pc = 2 \Rightarrow pc := 3
\end{array}
$$

Fig. 4. A Simple Illustrative Model



(a) Abstraction of the Small Model of Fig. 4

$t_1 \mathrel{\widehat{=}} q_0.0 \xrightarrow{e_1} q_2.1 \xrightarrow{e_3} q_4.2 \xrightarrow{e_5} q_6.2,$
$t_2 \mathrel{\widehat{=}} q_0.2 \xrightarrow{e_1} q_3.3 \xrightarrow{e_4} q_4.2 \xrightarrow{e_5} q_6.2,$
$t_3 \mathrel{\widehat{=}} q_1.3 \xrightarrow{e_2} q_2.2 \xrightarrow{e_3} q_5.3 \xrightarrow{e_5} q_7.3,$
$t_4 \mathrel{\widehat{=}} q_1.4 \xrightarrow{e_2} q_3.3 \xrightarrow{e_4} q_4.2 \xrightarrow{e_5} q_6.2,$
$t_5 \mathrel{\widehat{=}} q_1.5 \xrightarrow{e_2} q_3.4 \xrightarrow{e_4} q_5.3 \xrightarrow{e_5} q_7.3.$

(b) Concrete Tests

Fig. 5. Abstraction and Tests for the Small Model of Fig 4

Figure 5(a) shows the STS that abstracts the model of Fig. 4 from the set of abstract states $\mathsf{A} \mathrel{\widehat{=}} \{x < 3, x \geq 3\} \times \{pc = 0, pc = 1, pc = 2, pc = 3\}$. The

numbered points represent concrete states by defining the x value. The *may* existential transition relation Δ is represented by the dashed arrows. It defines an over-approximation. The transitions of Δ^- and Δ^+ are labelled with "+" and/or "-". A Δ^c relation is represented by the full arrows. All the concrete states in q_0 and q_1 belongs to C_0.

5 Abstraction Computation

We first present in Sec. 5.1 the basic algorithm to compute an existential over-approximation defined by Δ. Some implementation-dependent heuristics have been abstracted. This algorithm also computes Δ^+ and Δ^-, and we improve it by computing on-the-fly a *concrete* transition relation Δ^c that is an under-approximation obtained by a partial concretisation of Δ. We improve in Sec. 5.2 the under-approximation computation of Δ^c by taking the union with an under approximation defined in [6] from Δ^+ and Δ^-. Last, we discuss in Sec. 5.3 the properties of this combined approach.

5.1 Under-Approximation by Existential Concretisation

The algorithm of Fig. 6 relies on satisfiability evaluations of predicates by means of a SAT-solver. We consider a SAT-procedure as returning either the *false* value if the predicate is not satisfiable, or a concrete state c otherwise, that is also interpreted as the *true* value. In practice, SAT-solvers may also return an *unknown* value when they manage to prove neither the satisfiability nor the unsatisfiability of a predicate. We liken this *unknown* value to *false* in our algorithm.

The algorithm computes an abstraction (an STS) in two steps. Lines 1-5 compute the initial abstract and concrete states Q_0 and C_0 while lines 7-33 compute the set Q of reachable abstract states and the transitions of $\Delta, \Delta^+, \Delta^-$ and Δ^c. QR is the set of source states whose successors have to be computed, either because they are initial, or target states of reachable transitions of Δ. To compute the successors of a state q (lines 9-32), the algorithm enumerates all the possible target states q' (line 10) and all the events (line 11) that could lead to q'. The transition $q \xrightarrow{e} q'$ is added to Δ when $wcp(a, q') \wedge q$ is satisfiable (lines 12-14). Lines 15-19 search for a concretisation c' of the target state q', that would be the target of an existing concrete state c of the source state q. If such a c' is found (line 20), the transition $c \xrightarrow{e} c'$ is added to Δ^c (line 21), otherwise a transition $nc \xrightarrow{e} c'$ is added (line 23-24) where nc is the new concretisation of q resulting of the satisfiability condition $SAT(wcp(a, q') \wedge q)$ that guarantees its existence in lines 12-13. Line 26 stores the concretisation of the target state in Γ. If q' was not already known as reachable ($q' \notin Q$), it is added to QR (line 27). Then, if there exists a *may* transition, lines 28-29 complete the *must+* and *must-* transitions when they exist. Finally, this algorithm computes a first under-approximation defined by C_0 and Δ^c by concretising Δ. After that, a second step, described in the next section, completes this under-approximation by concretising Δ^+ and Δ^-.

```
          Let C be the set of concrete states of all the abstract states of the set Q
Inputs    ⟨X, Init, EvDef⟩ : an Event System where EvDef ≙ {e ≙ a|e ∈ Ev}
Results   ⟨Q, Q0, C0, Δ, Δ+, Δ−, Γ, Δc⟩: a Symbolic Transition System
Variables QR : Set of abstract states remaining to be handled
          CR : Set of concrete states remaining to be handled
          q, q′ : source and target abstract states of the current transition
          c, c′ : source and target concrete states of respectively q and q′
          e : event name of the current transition
          nc : other concrete state of the source state q
Begin
          /* Computation of the initial abstract and concrete states and of a concrete instance */
(1)       Q0 := ∅ ; Γ := ∅ ; C0 := ∅ ;
(2)       ForAll q ∈ A Do
(3)          c := SAT(sp(Init, true) ∧ q) ;
(4)          If c Then Q0 := Q0 ∪ {q} ; C0 := C0 ∪ {c} ; Γ := Γ ∪ {(q, c)} EndIf
(5)       EndForAll ;
(6)       /* Computation of the reachable states Q, the transitions in Δ, Δ+, Δ− and the concrete transitions in Δc */
(7)       Δ := ∅ ; Q := ∅ ; QR := Q0 ; Δc := ∅ ; Δ+ := ∅ ; Δ− := ∅ ;
(8)       While QR ≠ ∅ Do
(9)          Choose q in QR ; QR := QR − {q} ; Q := Q ∪ {q} ;
(10)         ForAll q′ ∈ A Do
(11)            ForAll e ∈ Ev Do /* e ≙ a in EvDef */
(12)               nc := SAT(wcp(a, q′) ∧ q) ;
(13)               If nc Then /* nc is false if wcp(a, q′) ∧ q is not satisfiable, true otherwise */
(14)                  Δ := Δ ∪ {q -e-> q′} ;
(15)                  CR := Γ({q}) ; c′ := false ; /* Γ(Z) is the relational image of the set Z */
(16)                  While ¬c′ ∧ CR ≠ ∅ Do
(17)                     Choose c in CR ; CR := CR − {c} ;
(18)                     c′ := SAT(sp(a, c) ∧ q′)
(19)                  EndWhile ;
(20)                  If c′ Then
(21)                     Δc := Δc ∪ {c -e-> c′}
(22)                  Else /* There is no concrete target state for the existing concrete source states */
(23)                     c′ := SAT(sp(a, nc) ∧ q′) ; /* we compute one from nc */
(24)                     Γ := Γ ∪ {(q, nc)}; Δc := Δc ∪ {nc -e-> c′}
(25)                  EndIf ;
(26)                  Γ := Γ ∪ {(q′, c′)} ;
(27)                  If q′ ∉ Q Then QR := QR ∪ {q′} EndIf ;
(28)                  If ¬SAT(¬(wp(a, q′) ∧ grd(a)) ∧ q) Then Δ+ := Δ+ ∪ {q -e-> q′} EndIf ;
(29)                  If ¬SAT(¬sp(a, q) ∧ q′) Then Δ− := Δ− ∪ {q -e-> q′} EndIf ;
(30)               EndIf
(31)            EndForAll
(32)         EndForAll
(33)      EndWhile
End
```

Fig. 6. Computation of an Existential Concretisation

5.2 Under-Approximation by Universal Concretisation

T. Ball proposes in [6] a method to compute the Δ, Δ^+ and Δ^- transition relations as defined in Sec. 3. An under-approximation L is defined by the set of states that are reachable from the initial ones by a sequence of *must-* transitions, followed by at most one *may* transition and a sequence of *must+* transitions. Ball defines a reachability function for an STS to formalise L. Let Q be a set of states and Δ a transition relation. The reachability function on Δ and $Q' \subseteq Q$ is defined by $\mathcal{R}[\Delta](Q') \mathrel{\widehat{=}} \mu Z.(Q' \cup \Delta(Z, Ev))$ where μ is the least fix-point and $\Delta(Z, Ev)$ is the relational image of the sets Z and Ev by Δ. We adapt the definition of L as:

$$L = \{q'' \mid \exists\, (q, e, q') \cdot (q \in \mathcal{R}[\Delta^-]\ (\Gamma^{-1}(\mathcal{R}[\Delta^c](C_0)))) \wedge (q' = q \vee q \xrightarrow{e} q' \in \Delta) \wedge q'' \in \mathcal{R}[\Delta^+](\{q'\})\}.$$

For example, this process applied to the specifications of Fig. 2(b) and 4, gives the transition relations Δ^- and Δ^+ represented respectively by the transitions labelled with "-" and/or "+" in Fig. 3 and in Fig. 5(a).

A contribution of our paper is to combine a concretisation of the under-approximation L with the one computed by our algorithm of Fig. 6. After the existential concretisation performed by our algorithm of Fig. 6, we concretise the longest executions of L that start in a concrete state that is reachable from the initial ones. The concretisation of the sequences of *must+* transitions is performed by a forward depth search, detecting the cycles from one concrete state beginning such a sequence. The concretisation of the sequences of *must-* transitions is performed by a backward depth search detecting the cycles from one of the final concrete states of these sequences. Finally, the relation Δ^c is the union of the relation computed by the algorithm of Fig. 6 with the one computed by this concretisation of L.

For the example of Fig. 5(a), the final Δ^c relation is represented by the full arrows. The transition $q_1.5 \xrightarrow{e_2} q_3.4$ concretises the first transition of the following sequence in L: $q_1 \xrightarrow{e_2,-} q_3 \xrightarrow{e_4,-} q_5 \xrightarrow{e_5,+} q_7$. Notice that this transition does not exist in the concretisation of Δ, but only in that of L. Finally, the whole under-approximation made of all the concrete transitions allows for generating the five test cases shown in Fig 5(b). They cover all the abstract states and all the concrete transitions of the STS of Fig 5(a). But they do not cover all the *may* executions of the abstract model. For example, the abstract execution $q_0 \xrightarrow{e_1} q_3 \xrightarrow{e_4} q_5 \xrightarrow{e_5} q_7$ is not covered, but it is not executable. This shows the interest of generating the tests from the under-approximation. This avoids generating non executable tests that give birth to a costly process of searching for an execution that does not exist.

5.3 Discussion about the Properties of the Algorithm

In this section, we explain why the algorithm of Fig. 6 terminates and is sound. We then explain that the enumeration order of the states and events in lines 10-11 impacts the accuracy of the computed under-approximation. Last we compare our combined method with the two methods of [6, 7].

Termination. Since A is a finite set, the external loop terminates in the worst case with $Q = \mathsf{A}$. The loops that compute the initial states (lines 1-5) and the two loops of lines 10-11 terminate because the sets A and Ev are finite. The internal one (lines 16-19) also terminates because the number of concrete states built for all the abstract states is finite. This number is bounded by $(|\mathsf{A}| \times |Ev|) \times 2 + 1$ in the worst case where, for any state, any event is fireable leading to any other state and is reachable by any event from every other state.

Soundness. The transition relation Δ^c is an under-approximation of the concrete transition system associated to an event system. On the one hand, the algorithm of Fig. 6 computes a subset of the concrete states of the model associated to an event system. On the other hand, all the states verify the correction conditions. For a transition $c \xrightarrow{e} c'$ concretising a transition $q \xrightarrow{e} q'$, the source state c verifies the condition $SAT(wcp(a, q') \wedge q)$, and the target state c' verifies the condition $SAT(sp(a, c) \wedge q')$ (see lines 18 and 23).

Search Order and Accuracy of the Abstractions. The number of reachable concrete executions and their size depend on the enumeration order of the events and concrete states, as Figs. 7(a) and 7(b) illustrate. Two test cases with a total of five test steps could be generated from the approximation of Fig. 7(a), while no test could be generated from the one of Fig. 7(b). Indeed, c_5, the concrete initial and reachable state of Fig. 7(b), is not connected to any concrete transition. There are three reasons for that. Concretising the reflexive transition *Rep* before $Fail$ generates the state c_4, because *Rep* is not fireable in the state c_5. The *Rep* transition does not connect c_4 to c_5 because c_4 has been built by the solver with an upper limit choice for the connected battery, i.e. battery 3, while the connected battery in c_5 is battery 1 (*Rep* doesn't change the connected battery). Consequently, the state c_3 is generated. Then $Fail$ is applied, and as the possible concrete target states are enumerated in reverse order w.r.t their creation, c_3 is connected to c_4 rather than to c_5. As a result, no transition is reachable from the state c_5. This illustrates that according to the building order of the abstraction, the under-approximation contains more or less reachable concrete transitions. Notice that our implementation optimises the size of the under-approximation by considering the heuristics and parameters presented in Sec. 6.1.

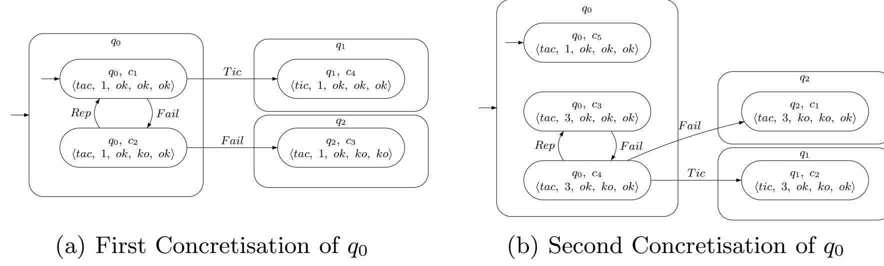
Comparison with the other Methods. The combined method that we propose generates under-approximations that are more accurate than with any of the two methods presented in [6, 7]. These two methods are not comparable as illustrated by Fig. 5(a) and shown in [7]. The concretisation of the existential abstraction, as in [7], allows for generating the four test cases t_1 to t_4 of Fig. 5(b). Our method additionally generates the test case t_5 thanks to the concretisation of L. In contrast, the method of [6] would only generate the test cases t_1 and t_5. Our method finally generates the union of the test cases generated by each of the two methods.

6 Implementation and Experimentations

We present in this section some optimisations to the algorithm of Fig 6. Our experimentations show their impact on test generation.

6.1 Implementation and Optimisation

We have used the SMT-solver Z3 [14] to implement the SAT procedure and the B substitution calculus to implement wp, wcp, grd and sp. Our implementation



(a) First Concretisation of q_0 (b) Second Concretisation of q_0

Fig. 7. Concretisations of q_0 for the Abstract Electrical System

of the algorithm of Fig 6 optimises it in two ways. Firstly, we improve its practical complexity by not examining all the triplets (q, e, q'). Secondly, we better connect the concrete transitions to produce longer executions. But the universal concretisation described in Sec. 5.2 has not been implemented yet.

Improvement of the Practical Complexity. The worst case theoretical complexity of our algorithm is $|\mathsf{A}|^2 \times |Ev|$ when the concretisation is not taken into account. Taking into account the concretisation, i.e. the loop in lines 16-19 of Fig. 6, the complexity is $|\mathsf{A}|^3 \times |Ev|^2$ because the maximal size of $\Gamma(\{q\})$ is $2 \times |\mathsf{A}| \times |Ev|$ when there is one transition for each event towards each abstract state because any abstract transition is concretised by one concrete transition. The number of triplets (q, e, q') to explore can be reduced by computing on the one hand for each abstract state the events that can be fired on it, independently of the target states, and on the other hand, for each abstract state, the events able to reach it, independently of the source states. Let *Fireable* be a partial function that maps each abstract state of A to the set of events fireable on it. Let *Reaching* be a partial function that maps each abstract state of A to the set of events able to reach it. The computation of *Fireable* and *Reaching* is in $|\mathsf{A}| \times |Ev|$ and allows for the following optimisations.

- In the third loop, on line 11, instead of enumerating all the events in Ev, it is possible to enumerate only the events in $Fireable(q) \cap Reaching(q')$: they are both fireable on q and able to reach q'.
- In the second loop (line 10), it is sufficient to enumerate the set of reachable states (domain of *Reaching*) instead of the set of all the abstract states in A.

Improvement of the Connectivity Between the Concrete Transitions. To get a "good" connectivity of the concrete transitions, the algorithm tries to concretise an abstract transition $(q \xrightarrow{e} q')$ by considering all the existing pairs of concrete states of q and q'. If it fails, the algorithm asks the solver to find for each concrete state of q a new concrete state of q' such that the concrete transition exists. If it fails again, a new pair of concrete states is built by the solver. Indeed, many concrete transitions between two different concrete states are transformed into

reflexive *may* transitions on an abstract state. In practice, to go from an abstract state to another, it is often necessary to apply these reflexive transitions to connect inside an abstract state the concrete target state of the incoming abstract transition to the concrete source state of the outgoing abstract transition. The concretisation could be improved at no supplementary computing cost by two implementation choices and three user parameters.

1. Concretise in priority reflexive transitions so as to increase the number of concrete states of q.
2. Enumerate in priority the concrete states reachable in q (line 17) and unreached in q' (line 18). This requires to mark as reachable or not the concrete states during their computation and to arrange Γ in order on-the-fly.
3. Enumerate the events in an order that causes the reflexive transition to chain. For example in the Electrical System, $Fail$ has to be called before Rep since a battery can be repaired only if it is broken.
4. Indicate the number of repetitions of some of the events, that need to be consecutively repeated to effectively make the system progress.
5. Provide a partial concretisation of the numerical variables of some targeted concrete states to guide the global concretisation.

The last three parameters rely on the hypothesis that the tester has a sufficiently good knowledge of the system and its model to be able to provide them. Knowing the "common sense" sequencing order of the events and their number of repetitions seems natural. But a partial concretisation is surely a more difficult thing to know about.

6.2 Experimental Results

In the tables, the symbols "♯", "Pot.", "Trans.", "Inst." and "Abs." respectively stand for *number of*, *Potential*, *Transitions*, *Instantiated* and *Abstract*. Table 1 compares two processes of test generation from abstraction. We compare test sequences that come from the abstraction algorithm defined in Sec. 5 (Process 1) with previous results obtained with a process (Process 2) in which we generate tests as executions of Δ and then instantiate them *a posteriori*. The comparison is based on the coverage of the abstract transitions and the abstract states. For Process 1, we give also the coverage of concrete transitions and concrete states of Δ^c. In Process 2, we have used the theorem prover of Atelier B [15] instead of the SMT-solver Z3.

Since the tools used to generate the abstractions are not the same in Process 1 and Process 2, directly comparing their execution times does not make sense. Except for Demoney, these times are small with both processes. We notice however that the computation of the Demoney abstraction took 1381 s. with Process 1 *vs.* and 12917 s. with Process 2. We have observed that 9522 s. was spent at instantiating the tests *a posteriori* with Process 2. This indicates that instantiating the tests on-the-fly is more efficient.

Process 1 gives better coverage ratios for the abstract states (up to 40%) and transitions (up to 45%), excepted for the small model of Fig. 4, where Process 2

also covers the test t_5 that should be found by the second step of Process 1 described in Sec. 5.2.

Model (loc)	♯Pot. states	Process 1 : Concretisation Process						Process 2 : Process with instantiation			
		♯Tests	♯Tests Steps	Abstract States Coverage	Abstract Trans. Coverage	Concrete States Coverage	Concrete Trans. Coverage	♯Inst. Tests / ♯ Abs. Tests	♯Tests Steps	Abstract States Coverage	Abstract Trans. Coverage
Small Model (20)	∞	7	13	8/8 (100%)	8/10 (80%)	11/15 (73%)	8/11 (73%)	4/4 (100%)	12	8/8 (100%)	10/10 (100%)
SysAlim (100)	36	1	28	4/4 (100%)	11/11 (100%)	8/8 (100%)	15/15 (100%)	4/4 (100%)	21	4/4 (100%)	11/11 (100%)
QuiDonc (170)	13	2	90	5/5 (100%)	22/22 (100%)	13/13 (100%)	40/40 (100%)	2/5 (40%)	16	3/5 (60%)	14/22 (64%)
Robot (100)	384	29	224	4/4 (100%)	21/31 (68%)	44/59 (74%)	50/72 (69%)	2/4 (50%)	25	3/4 (75%)	17/31 (55%)
Partition (80)	∞	5	9	4/7 (57%)	5/23 (22%)	9/41 (22%)	5/28 (18%)	2/14 (14%)	6	3/7 (43%)	5/23 (22%)
DeMoney (330)	∞	34	591	10/14 (71%)	237/368 (64%)	31/161 (19%)	499/1669 (30%)	17/18 (95%)	170	9/14 (64%)	197/368 (54%)

Table 1. Efficiency of the Test Generation Methods

The experimental results of Table 2 are about the impact of the parameters of Sec. 6.1 (items 3, 4 and 5), in terms of number of tests, number of test steps and coverage of the states and the transitions of both the abstraction and its concretisation.

We have observed each parameter variation with the two other parameters fixed at the best value that we found on the examples. The variations are as follows. There are two events sequencing: intuitive or non-intuitive. Each event is repeated either once (default parameter) or the number of times indicated in Table 2. Only Demoney has numerical variables: they are either not at all (default parameter) or partially concretised.

The results show that modifying the sequencing of the events for the Robot and Demoney examples changes the coverage ratios (up to 4 times more) and the number of test and test steps (up to 8 times more) very significantly. Repeating some of the events considerably improves the quality of the results, in terms of abstraction coverage (up to 10 times more) and of number of tests and test steps (up to 26 times more). The partial concretisation of the targeted states also substantially improves the coverage of the abstraction (up to 10 times more) and the number of tests and test steps (up to 26 times more). We have also noticed with Demoney that playing with the three parameters at once could be necessary to better improve the results.

Parameter	Model	Parameter Value	♯Tests	♯Tests Steps	Coverage		Coverage	
					Abstract States	Abstract Trans.	Concrete States	Concrete Trans.
Events Enumeration Order	Robot	Loading order	2	16	50%	26%	21%	26%
		Inverse loading order	1	2	25%	6%	7%	6%
	DeMoney	Valid credit order	34	591	71%	64%	19%	30%
		Inverse valid credit order	20	422	50%	45%	15%	25%
Events Multiple Call	Robot	Part_Arrival: 3	29	224	100%	68%	74%	69%
		Default parameter	7	35	75%	35%	34%	42%
	DeMoney	Put_Data: 4	34	591	71%	64%	19%	30%
		Default parameter	1	22	7%	6%	2%	6%
Partial Concrete Valuation	DeMoney	A partial concretisation	34	591	71%	64%	19%	30%
		Default parameter	1	22	7%	6%	2%	6%

Table 2. Impact of the Use of Parameters for Test Generation

7 Related Works

Some algorithms that compute over-approximations based on predicate abstraction can be found e.g. in [2, 16]. Predicate abstraction is also used by Ball in [6, 17] and Păsăreanu et al. in [7] to compute program abstractions that are under-approximations for generating tests. Ball computes an under-approximation from the reachable sequences of *must-* transitions followed by at most one *may* transition and a sequence of *must+* transitions. Păsăreanu et al. also compute an under-approximation, but by concretisation of the *may* transitions. We combine these two methods in this paper by concretising *may* transitions and the sequences of *must* transitions. We do not refine the set of abstraction predicates as in [7, 18] to search for an exact abstraction. But we present a parametrised algorithm and many optimisations by heuristics to improve the concretisation.

We have defined in [5] a method to extract a set of predicates $\mathcal{P}$ from a test purpose and a behavioural model for generating tests in an MBT approach. In [6], $\mathcal{P}$ is made of all the atomic predicates that appear in the control structure of a C program. In [18], the set of predicates is iteratively refined in order to compute a bisimulation of the initial model when it exists. SYNERGY [19] and DASH [20] also combine under-approximation and over-approximation computations to check safety properties on programs. As we aim at proposing an efficient MBT method, our algorithm always terminates because it does not refine the over-approximation. Moreover, it generates on-the-fly the under-approximations by using the witnesses of satisfiability proofs to build the over-approximation.

Other works are about generating tests from abstraction. The tools Agatha [21], DART [22], CUTE [23], EXE [24] and PEX [25] also compute abstractions from models or from programs, but by means of symbolic execution [4]. This data abstraction approach computes an execution graph. Its set of abstract states is possibly infinite whereas it is finite with the predicate abstraction method. The methods of [26] implemented in STG [27] use abstractions defined by the user and modelled by IOSTS (Input Output Symbolic Transition System). They use test purposes synchronised with abstractions, both defined as IOSTS. Then, the synchronised product allows for generating tests after an optimisation step, which consists of pruning the unreachable states by abstract interpretation. Our approach is very similar in that we also use test purposes and abstractions, as well as synchronisation. But there are three differences. First, our abstractions are computed from a set of predicates defined from the test purposes, whereas STG uses user-defined abstractions. Second, an optimisation is performed by the abstraction computation by using the invariant properties (that do not exist in an IOSTS) specified in the B models used in our experimentations. It allows, for the weakest precondition computation, to minimise the symbolic state space and the feasible transitions. Third, we use SMT solvers, that combine constraint solving and theories for proof, instead of pure constraint solving to instantiate the symbolic tests.

Similarly to the concolic execution in [23], we combine concrete execution with predicate abstraction. But concolic tools use symbolic execution instead of predicate abstraction. Furthermore, we use a dual combination by performing

predicate abstraction and concretising incrementally the abstract transitions, whereas concolic execution performs a concrete execution and at the same time collects the symbolic path constraints. Moreover, hybrid concolic execution [28] combines random generation of input values.

8 Conclusion and Further Works

We have presented a method of model-based test generation and an algorithm that combines two under-approximation computations (defined in [6, 7]) by predicate abstraction. Our contributions are this combination and the design of three adequate parameters to capture the know-how of the tester and the definition of pertinent heuristics for improving the accuracy of the under-approximations. Our experimental results indicate that generating the tests from this under-approximation with an on-the-fly instantiation is more efficient and gives better coverage ratios than an over-approximation based process with afterwards instantiation. Our experimental results also measure the impact of each parameter on the test coverage ratios. This work shows that using under-approximations is decisive for the scalability of the method. This paper also shows that the efficiency of the method depends on capturing of the tester's expertise. We propose three parameters to the tester: the order in which the operations are considered, the number of their consecutive repetition and a subset of the concrete states to target. The practical usability of these parameters by a validation engineer has to be assessed by experimentations on larger case studies. Also, we intend as future works to define better parameters that would depend on the application domains.

We also have to improve the implementation of the concretisation of the under-approximation L defined in Sec. 5.2. Additionally, for the SMT-solvers to be able to deal with our behavioural models, we have by now to restrict our usage of some of the B set operators. To overcome this limitation, it will be necessary either to define new theories for the SMT-solvers, or to automatically rewrite all B predicates as first-order logic formulas.

References

1. Utting, M., Legeard, B.: Practical Model-Based Testing. Morgan Kaufmann (2006)
2. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: CAV'97. Volume 1254 of LNCS. (1997) 72–83
3. Cousot, P., Cousot, R.: Abstract interpretation frameworks. J. Log. Comput. **2**(4) (1992) 511–547
4. Păsăreanu, C.S., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. STTT **11**(4) (2009) 339–353
5. Bouquet, F., Bué, P.C., Julliand, J., Masson, P.A.: Test generation based on abstraction and test purposes to complement structural tests. In: A-MOST'10, 6th int. Workshop on Advances in Model Based Testing, Paris, France (April 2010)
6. Ball, T.: A theory of predicate-complete test coverage and generation. In: FMCO'04. Volume 3657 of LNCS. (2005) 1–22

7. Păsăreanu, C.S., Pelánek, R., Visser, W.: Predicate abstraction with under-approximation refinement. LMCS **3**(1) (2007)
8. Julliand, J., Masson, P.A., Tissot, R.: Generating security tests in addition to functional tests. In: AST'08, ACM Press (2008) 41–44
9. Abrial, J.R.: Modeling in Event-b: System and Software Design. Cambridge Univ. Press (2010)
10. Dijkstra, E.: Guarded commands, nondeterminacy, and formal derivation of programs. C. ACM **18** (1975)
11. Dijkstra, E.: A Discipline of Programming. Prentice-Hall (1976)
12. Hoare, C.: An axiomatic basis for computer programming. CACM **12** (1969) 576–580
13. Bert, D., Cave, F.: Construction of finite labelled transistion systems from b abstract systems. In: IFM 2000. (2000) 235–254
14. de Moura, L., Bjørner, N.: An efficient smt solver. In: TACAS'08. Volume 4963 of LNCS. (2008) 337–340
15. Atelier B: Case tool for developing software proven without default: http://www.atelierb.eu
16. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of c programs. In: PLDI. (2001) 203–213
17. Ball, T., Kupferman, O., Yorsh, G.: Abstraction for falsification. In: CAV'05. (2005) 67–81
18. Namjoshi, K.S., Kurshan, R.P.: Syntactic program transformations for automatic abstraction. In: CAV'00. Volume 1855 of LNCS. (2000) 435–449
19. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Synergy: a new algorithm for property checking. In: SIGSOFT FSE. (2006) 117–127
20. Beckman, N.E., Nori, A.V., Rajamani, S.K., Simmons, R.J., Tetali, S., Thakur, A.V.: Proofs from tests. IEEE Trans. Software Eng. **36**(4) (2010) 495–508
21. Rapin, N., Gaston, C., Lapitre, A., Gallois, J.P.: Behavioral unfolding of formal specifications based on communicating extended automata. In: ATVA'03. (2003)
22. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: PLDI. (2005) 213–223
23. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: ESEC/SIGSOFT FSE. (2005) 263–272
24. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: ACM Conference on Computer and Communications Security. (2006) 322–335
25. PEX: Automated exploratory testing for .NET: http://research.microsoft.com/en-us/projects/pex
26. Calamé, J., Ioustinova, N., van de Pol, J.: Automatic model-based generation of parameterized test cases using data abstraction. ENTCS **191** (2007) 25–48
27. Jeannet, B., Jéron, T., Rusu, V., Zinovieva, E.: Symbolic test selection based on approximate analysis. In: TACAS'05. Volume 3440 of LNCS. (2005) 349–364
28. Majumdar, R., Sen, K.: Hybrid concolic testing. In: ICSE'07. (2007) 416–426