Two Under-Approximation Techniques for 3-Modal Abstraction Coverage of Event Systems: Joint Effort? [*]

J. Julliand, O. Kouchnarenko, P.-A. Masson, G. Voiron

FEMTO-ST Institute, Univ. Bourgogne Franche-Comté, CNRS

16, route de Gray F-25030 Besançon Cedex France

{jacques.julliand, olga.kouchnarenko, pierre-alain.masson, guillaume.voiron}@femto-st.fr

Abstract

Model-based testing of event systems can take advantage of considering abstractions rather than explicit models, for controlling their size. A test is then a connected and reachable event sequence. This paper reports on experiments made when adapting for event systems two known under-approximation techniques of predicate tri-modal (may, must+, must-) abstractions. We first instantiate all the abstract may transitions, preferably as reachable instances. Second, we complete this under-approximation with instantiations of Ball chains, i.e. sequences in the shape of must-*.may.must+* transitions, as such sequences are guaranteed to have connected instantiations. We present a backward symbolic instantiation algorithm for connecting these two under-approximations. We experimentally address the question of their complementarity. Surprisingly, our results show that Ball chains have not enhanced the coverage already achieved by the first step of may-transitions instantiation. However, it has enhanced the number of test steps by prolonging the already existing tests.

Keywords. Predicate abstraction; under-approximation; event systems; may/must transitions; test generation

1 Introduction

Abstracting a program or its specification allows to control the size of its state space description, at the price of a loss in accuracy. That facilitates their algorithmic exploitation, otherwise limited by the huge if not infinite number of concrete states. The general idea of abstraction is to gather states that share common properties into super-states. In predicate abstraction [GS97] the concrete states are mapped onto a finite set of abstract ones, by means of a set of predicates that characterizes each abstract state. We are interested in generating model-based tests from a predicate abstraction of a system formally specified as an event system [Abr10], which is a special kind of action system [Dij75, Dij76]. Such tests will be computed as concrete instantiations of abstract transition sequences, which raises the two issues: to be used as model-based tests, these instantiations have to be 1) connected, and 2) reachable.

Concerning connectivity, an abstract transition links two abstract states when it has at least one concrete instantiation. Such transitions are called *may* [GJ03], meaning that they may be instantiated. But two consecutive *may* instantiations might be disconnected since the concrete transitions may not end and begin on the same concrete state. Reachability is also an issue with event systems: an event specifies state variable modifications by means of a guarded action. The actions are activated whenever their guard becomes true, so that contrarily to programs, there is no natural control flow that one could follow to guarantee

[*]in TASE 2017, 11th Int. Symposium on Theoretical Aspects of Software Engineering, IEEE, sep. 2017

reachability. Hence we seek to under-approximate the abstraction by computing connected and reachable concrete instances of the abstract event sequences.

To this end, two under-approximations approaches from literature have been retained. The first one is by M. Veanes and R. Yavorsky [VY03] for under-approximating an Abstract State Machine specification by a Finite State Machine. It gradually explores the abstraction by widening a frontier of the visited states, while exhibiting concrete instances whose reachability is either guaranteed when possible, or uncertain otherwise. We call this approach *concrete exploration* (CXP). The second one is by T. Ball [Bal04], intended to predicate testing of C programs. It associates the abstract transitions with three possible modalities: *may*, *must*+ and *must*− for exploiting a structural property that the abstract sequences of the form $(must-)^*.may.(must+)^*$ are guaranteed to be connectedly instantiable. We call such sequences "Ball chains". In the context of event systems, we still have to seek for their reachability.

We have recently adapted and applied these methods to event systems, but separately. Indeed, [JKMV17] provides an algorithm that adapts and applies the concrete exploration to event systems. In [BJM15, BJM16] forward symbolic exploration from the initial states to reach the Ball chains has been used. In this paper we build the bridge for combining the two methods, which allows us to implement and experiment with case studies. The main idea is to connect the Ball chains instantiations to the concrete reachable paths *à la* Veanes and Yavorsky, for ensuring their reachability. We present an algorithm that instantiates the Ball chains, and proceeds to a backward symbolic exploration of the *must*− transitions towards these reachable paths. Also, we experimentally investigate the complementarity of the two methods w.r.t. various abstractions of four realistic case studies. Surprisingly, our results show that the Ball chains do not provide additional coverage of the abstract states and transitions than the concrete exploration does. Nevertheless, they prolong the already existing concrete sequences, thus providing more test steps to explore the system.

The technical background of our paper regarding event systems, predicate abstraction and tri-modal transition systems is given in Section 2. Section 3 presents a simple coffee machine as an illustrative example. Our method, and in particular the algorithm for computing and instantiating the Ball chains with backward symbolic exploration towards the concrete exploration, is presented in Section 4. Our experimental results are presented in Section 5. Section 6 describes related work, and Section 7 concludes the paper.

2 Background

In this paper systems are specified by event systems (ES) described using the B syntax [Abr96, Abr10][1]. Notice however that our proposals and results are applicable to all models whose semantics is expressed by the concrete labelled transition systems.

In this section we first present the syntax and the semantics of the B event systems. Then we present the concept of predicate abstraction and formalize the abstraction of ES by means of Tri-modal Transition Systems (3MTS). Last, we present Ball's universal under-approximation based on 3MTS.

2.1 Model Syntax and Semantics

We start by introducing B event systems in Def. 1. Let a be a guarded action, E, F be arithmetic expressions and P, P' be predicates, the events are defined by means of guarded actions [Dij75] composing five primitive actions: *skip* an action with no effect, $x, y := E, F$ a multiple assignment, $P \Rightarrow a$ a guarded action, $a_1 [] a_2$ a non-deterministic choice between a_1 and a_2, and $@z.a$ an unbounded non-deterministic choice for all the values of z satisfying

[1]We could alternatively use a syntax with guarded commands [Dij75], such as Abstract State Machines [GKOT00, Gur00].

the guard of a. Notice that $a_1 \mid\mid a_2$ as well as *IF P THEN a END* can be rewritten with the five primitives actions as explained in [Abr96].

Definition 1 (Event System) *Let* Ev *be a set of event names. A B event system is a tuple $\langle X, I, Init, \mathrm{EvDef}\rangle$ where X is a set of state variables, I is a state invariant, $Init$ is a guarded action that initializes the system, such that I holds in any initial state,* EvDef *is a set of event definitions, each in the shape of $e \overset{def}{=} a$ for any $e \in \mathrm{Ev}$, where a is a guarded action that preserves I.*

Figure 2 depicts an example of an event system that illustrates Def. 1. Following [BC00], Concrete labelled Transition System (CTS) describe the semantics of event systems.

The event systems have also an axiomatic semantics. An event $e \overset{\text{def}}{=} a$ has a *weakest precondition* [Dij76] w.r.t. a set of target states Q' denoted as $wp(a, Q')$. $wp(a, Q')$ is the largest set of states from which applying a always leads to a state of Q'. Let us now formally define wp following [BJM16]. Basically, we directly consider the set of states Q and Q' as predicates of the same name. We define the wp w.r.t. the five primitive actions as:

- $wp(skip, Q') \overset{\text{def}}{=} Q'$,
- $wp(x := E, Q') \overset{\text{def}}{=} Q'[E/x]$ that is the usual substitution of x by E as in [Hoa69],
- $wp(P \Rightarrow a, Q') \overset{\text{def}}{=} P \Rightarrow wp(a, Q')$,
- $wp(a_1[]a_2, Q') \overset{\text{def}}{=} wp(a_1, Q') \wedge wp(a_2, Q')$,
- $wp(@z.a, Q') \overset{\text{def}}{=} \forall z.wp(a, Q')$ where z is not free in Q'.

2.2 Predicate Abstraction and Tri-modal Transition Systems

Predicate abstraction [GS97] is a special instance in the framework of abstract interpretation [CC92] that maps the potentially infinite state space C of a CTS onto the finite state space A of an abstract transition system *via* a set of n predicates $\mathcal{P} \overset{\text{def}}{=} \{p_1, p_2, \ldots, p_n\}$ over the state variables. The set of abstract states A contains 2^n states. Each state is a tuple $q \overset{\text{def}}{=} (q_1, q_2, \ldots, q_n)$ 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 a total abstraction function $\alpha : C \to \mathsf{A}$ such that $\alpha(c)$ 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, or q contains c, or that c is a state of q.

Let us now define the abstract transitions as *may* ones. Consider two abstract states q and q', and an event e. There exists a *may* transition from q to q' by e, denoted by $q \xrightarrow{e} q'$, iff there exists at least one concrete transition $c \xrightarrow{e} c'$ where c and c' are concrete states with $\alpha(c) = q$ and $\alpha(c') = q'$.

As in [Bal04], we define *must*+ and *must*− transitions in addition to *may* ones. The *must*+ transitions are *may* transitions that are triggerable from any concrete state of the abstract source state. The *must*− transitions are *may* transitions that can reach any concrete state of the abstract target state. The modalities are evaluated resolving SAT formulas. As abstraction we define Tri-modal Transition Systems (3MTS) in Def. 2. They are transition systems with abstract states and abstract transitions characterized as *may*, *must*+ or *must*−.

Definition 2 (Tri-modal Transition System) *Let* Ev *be a finite set of event names and $\mathcal{P}$ be a set of n predicates. Let A be a finite set of 2^n abstract states. A 3MTS is a tuple $\langle Q, Q_0, \Delta, \Delta^+, \Delta^-\rangle$ where $Q(\subseteq \mathsf{A})$ is a finite set of states, $Q_0(\subseteq Q)$ is a set of abstract initial states, $\Delta(\subseteq Q \times \mathrm{Ev} \times Q)$, $\Delta^+(\subseteq \Delta)$ and $\Delta^-(\subseteq \Delta)$ are respectively a may, must+ and must− labelled transition relations.*

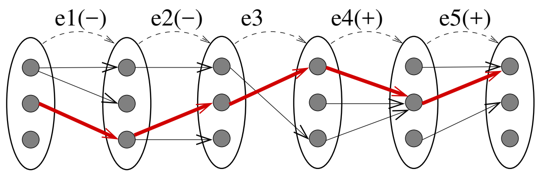

Figure 1. A Concretization of a $(must-)^* \cdot may \cdot (must+)^*$ Sequence of Abstract Transitions

The 3MTS definition above is inspired by the corresponding notion in [Bal04], in its turn coming from the Modal Transition Systems defined in [LT88, GHJ01]. A 3MTS can be associated with every event system (cf. [BJM16] for a formal definition).

An example of a 3MTS, whose ES is described in Fig. 2, is provided in Fig. 3. The four abstract states, named q_1 to q_4, appear as circle boxes. The predicates p_0, p_1 and p_2 from which they are defined are given explicitly in Section 3. The abstract transitions of Δ are represented as arrows labelled with an event name, and the possible mentions $+$ and/or $-$ indicating respectively when they are in Δ^+ or Δ^-.

2.3 Over-/Under-Approximations Based on 3MTS

We denote by $q_0 \xrightarrow{e_0} q_1 \xrightarrow{e_1} \dots$ where $q_i \xrightarrow{e_i} q_{i+1} \in \Delta$ for $i \geq 0$ an abstract execution, and by $c_0 \xrightarrow{e_0} c_1 \xrightarrow{e_1} \dots$ a concrete execution. We say that $q_0 \xrightarrow{e_0} q_1 \xrightarrow{e_1} \dots$ and $c_0 \xrightarrow{e_0} c_1 \xrightarrow{e_1} \dots$ are similar when for all i, c_i is a state of q_i.

An abstraction is an *over-approximation* of a model when, for every execution of the model, there is a similar execution of the abstraction. In other words, the abstraction may define more and/or longer executions than the model but not less. Any safety properties that hold on such an abstraction also hold on the model, which allows for verifying any safety properties on the abstraction rather than on the model. But for testing, since an over-approximation may define more executions than the model, a test extracted as an execution path of the abstraction may possibly not be instantiable as a model execution.

So testing can take advantage of considering under-approximations rather than over-approximations. An abstraction is an *under-approximation* of a model when for every execution of the abstraction, there is a similar execution of the model. In other words, the abstraction may define less and/or shorter executions than the model but not more. Thus every test extracted from such an abstraction is guaranteed to be instantiable on the model, to give rise to a concrete test.

In [Bal04] the authors have defined a method to compute an under-approximation by means of the Δ, Δ^+ and Δ^- abstract transition relations of a 3MTS.

In [Bal04] it is proven that a sequence of *must*$-$ transitions, followed by at most one *may* transition, followed by a sequence of *must*$+$ transitions, is guaranteed to be instantiable as a connected sequence of concrete transitions. Indeed, as illustrated by the bold sequence in Fig. 1, any concrete state of a *must*$-$ target is reached from some concrete source state, while whatever concrete state is reached by a *must*$+$ transition is possible to leave from. A *may* transition in between joins a necessarily reached state to a necessarily left one. We call such a sequence a *Ball chain*, and we denote it by a regular expression[2] $(must-)^*.may.(must+)^*$.

3 Illustrative Example

Our illustrative example is a simple coffee vending machine (see the ES in Fig. 2). It has a *Balance*, which can be augmented by putting coins of value either 50 or 100 (events

[2]Since a *must* transition is also *may*, we see the central *may* transition as mandatory.

$$
\begin{array}{lcl}
\mathsf{X} & \overset{\mathsf{def}}{=} & \{Balance, Pot, Status, CofLeft, AskCof, AskChange\} \\
\mathsf{I} & \overset{\mathsf{def}}{=} & Pot \in 100..MAX\_Pot + 50 \wedge Balance \in 0..MAX\_Bal \wedge \\
 & & CofLeft \in 0..MAX\_Cof \wedge Pot \; mod \; 50 = 0 \wedge Balance \; mod \; 50 = 0 \wedge \\
 & & Status \in 0..2 \wedge AskCof \in 0..1 \wedge AskChange \in 0..1 \wedge \\
 & & AskChange = 1 \Rightarrow (Balance > 0 \wedge AskCof = 0) \wedge \\
 & & AskCof = 1 \Rightarrow (Balance \geq 50 \wedge AskChange = 0) \wedge \\
 & & Balance = 0 \Rightarrow (AskCof = 0 \wedge AskChange = 0) \\
\mathsf{Init} & \overset{\mathsf{def}}{=} & Balance := 0 \; || \; Status := 0 \; || \; Pot := 100 \; || \\
 & & CofLeft := 10 \; || \; AskCof := 0 \; || \; AskChange := 0 \\
\mathsf{insert50} & \overset{\mathsf{def}}{=} & Status = 1 \wedge AskChange = 0 \wedge AskCof = 0 \wedge \\
 & & Balance + 50 \leq MAX\_Bal \Rightarrow Balance := Balance + 50 \\
\mathsf{insert100} & \overset{\mathsf{def}}{=} & Status = 1 \wedge AskChange = 0 \wedge AskCof = 0 \wedge \\
 & & Balance + 100 \leq MAX\_Bal \Rightarrow Balance := Balance + 100 \\
\mathsf{powerUp} & \overset{\mathsf{def}}{=} & Status = 0 \wedge CofeLeft > 0 \wedge Pot > 0 \wedge Pot \leq MAX\_Pot \Rightarrow \\
 & & Status := 1 \; || \; Balance := 0 \; || \; AskCof := 0 \; || \; AskChange := 0 \\
\mathsf{powerDown} & \overset{\mathsf{def}}{=} & (Status = 1 \wedge AskChange = 0 \wedge AskCof = 0 \wedge Balance = 0) \\
 & & \vee Status = 2 \Rightarrow Status := 0 \\
\mathsf{autoOut} & \overset{\mathsf{def}}{=} & Status = 1 \Rightarrow Status := 2 \\
\mathsf{takePot} & \overset{\mathsf{def}}{=} & Status = 0 \Rightarrow (Pot := 200 \; [] \; Pot := 100) \\
\mathsf{cofReq} & \overset{\mathsf{def}}{=} & Status = 1 \wedge Balance \geq 50 \wedge AskCof = 0 \wedge \\
 & & AskChange = 0 \Rightarrow AskCof := 1 \\
\mathsf{changeReq} & \overset{\mathsf{def}}{=} & Status = 1 \wedge Balance > 0 \wedge AskCof = 0 \wedge \\
 & & AskChange = 0 \Rightarrow AskChange := 1 \\
\mathsf{addCof} & \overset{\mathsf{def}}{=} & \exists x.(x \in 1..MAX\_Cof \wedge CofLeft + x \leq MAX\_Cof \\
 & & \wedge Status = 0 \Rightarrow CofLeft := CofLeft + x) \\
\mathsf{serveCof} & \overset{\mathsf{def}}{=} & Status = 1 \wedge Balance \geq 50 \wedge AskCof = 1 \wedge CofLeft > 0 \wedge \\
 & & Pot \leq MAX\_Pot \Rightarrow \\
 & & \quad AskCof := 0 \; || \; Balance := Balance - 50 \\
 & & \; || \; CofLeft := CofLeft - 1 \; || \; Pot := Pot + 50 \\
 & & \; || \; (Pot + 50 > MAX\_Pot \vee CofLeft = 1 \Rightarrow Status := 2 \\
 & & \quad [] \; Pot + 50 \leq MAX\_Pot \wedge CofLeft \neq 1 \Rightarrow skip) \\
 & & \; || \; (Balance > 50 \Rightarrow AskChange := 1 \; [] \; Balance = 50 \Rightarrow skip) \\
\mathsf{backBalance} & \overset{\mathsf{def}}{=} & Status = 1 \wedge Balance > 0 \wedge AskChange = 1 \Rightarrow \\
 & & Balance := 0 \; || \; AskChange := 0
\end{array}
$$

Figure 2. ES Specification of a Coffee Machine

insert50 and insert100 in Fig. 2). *Balance* may not exceed an arbitrary fixed constant named MAX_Bal. There are also arbitrary constants for the maximal number of coffees stored in the machine (MAX_Cof), and the maximal value (MAX_Pot) of the *Pot* (the money kept by the machine). Notice that *Balance* and *Pot* are multiples of 50. The machine has a *Status* which indicates if it is switched on (1) or off (0), or out of order (2). When switched on, the machine can serve coffees, after a request by the user (event cofReq that corresponds to pressing the "request coffee" button), at the price of 50 each (event serveCof): this price is retrieved from the *Balance* and sent to the *Pot*. The number of available coffees is modelled by the *CofLeft* variable. The user can ask for its change (event changeReq that corresponds to pressing the "give change" button). The events changeReq and cofReq are mutually exclusive. The user can then get its unused balance back (event backBalance). When switched off, the machine can be refilled with coffee (event addCof), and its *Pot* retrieved (event takePot). The events powerUp and powerDown are for switching the machine respectively on or off. Finally, a special event (autoOut) sets the machine out of order: it models the unexpected occurrence of a failure while the machine is in use. It also occurs when either there is no more coffee, or the *Pot* is full (see serveCof). Figure 3 represents the 3MTS of the ES in Fig. 2 for the three following predicates $p_0 \overset{\text{def}}{=} Status = 0$, $p_1 \overset{\text{def}}{=} Status = 1$ and $p_2 \overset{\text{def}}{=} (Status = 1 \wedge AskChange = 0 \wedge AskCof = 0 \wedge Balance = 0) \vee Status = 2$ that are respectively the guards of the events takePot, autoOut and powerDown.

4 Ball Chains Computation and Instantiation

This section presents our method for computing both an abstraction that is a 3MTS, and an under-approximation.

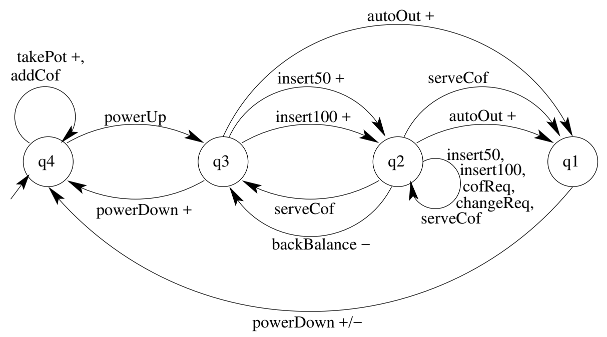

Figure 3. 3MTS of the Coffee Machine w.r.t. Predicates p_0, p_1 and p_2.

4.1 Approximated Transition System ATS

The reunion of both the abstraction and its under-approximation we call an Approximated Transition System (ATS), defined in Def. 3.

Definition 3 (Approximated Transition System) *Let $\langle Q, Q_0, \Delta, \Delta^+, \Delta^- \rangle$ be an 3MTS. A tuple $\langle Q, Q_0, \Delta, \Delta^+, \Delta^-, C, C_0, \Delta^c, \alpha, \kappa, \rangle$ is an ATS whose $\langle C, C_0, \Delta^c, \alpha, \kappa \rangle$ is a concretization of the 3MTS where:*

- *C, C_0 are sets of respectively concrete states and concrete initial states,*
- *$\Delta^c (\subseteq C \times Ev \times C)$ is a concrete labelled transition relation.*
- *α is a total abstraction function from C to Q,*
- *κ is a total coloration function from C to $\{green, blue\}$.*

The concrete part $\langle C, C_0, \Delta^c \rangle$ is an under-approximation of the labelled transition system that is the semantics of the event system from which the 3MTS is deduced. In the rest of the paper, for the abstract states, we distinguish between the *may-reachability* and the *reachability*. The former is the reachability by the abstract *may* transition relation Δ, and the latter is the reachability by the concrete transition relation Δ^c in the ATS. We say that an abstract state q is reachable if there exists at least one concrete instance of q that is reachable thanks to the transition relation Δ^c. By extension, an abstract transition is reachable if there exists at least one concrete instance in Δ^c whose source state is reachable. The concretization of the *may* transitions is performed on the fly during the computation of the 3MTS. The algorithm for doing this is given in [JKMV17], and is summarized as follows. It guarantees that every *may* transition between two abstract states is concretized. A total abstraction function α maps each concrete state of C to an abstract state of Q. Adapted from [VY03], a coloration function κ maps each concrete state of C to a colour: green for the reachable states, and blue for those whose reachability is unknown. It allows colouring the concrete transitions to reflect how they are known to be reachable: green for certain reachability, blue for uncertain.

4.2 Ball Chains Instantiations

In this paper, we complete the under-approximation computed by the algorithm (summarized above) of [JKMV17], with a concretization of the Ball chains defined by the *must*+ and *must*− transitions. We proceed separately with the *must*+ and the *must*− chains. For the former we concretize forwardly, by simple DFS, any *must*+ chain of each *must*+ structure (see Fig. 4(c) and Fig. 4(a)). The DFS is launched from any *must*+ transition without predecessor. If all *must*+ transitions have a predecessor, the DFS is launched from a transition arbitrarily chosen in a strongly connected component. The algorithm is straightforward and is not presented in this paper. In order to reconnect when possible the (as yet) unreached transitions, it connects in priority a green instance of the source state if available, to a blue instance of its target state. Exemplified with Fig. 4(a), it tries to connect a green instance of state 1 to a blue instance of state 2. If no such blue target instance is available, a green one is used instead. And if none of the source instances is green, the transition is concretized in blue.

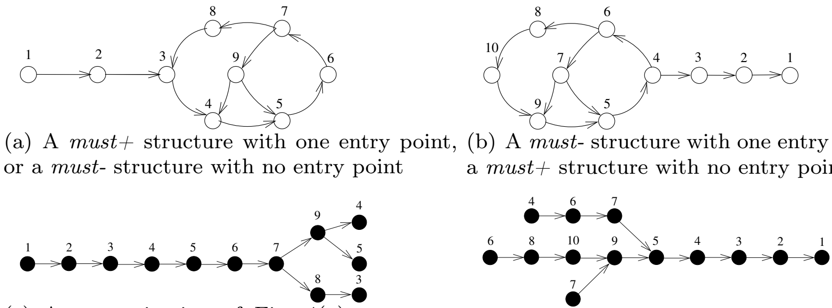
(a) A *must*+ structure with one entry point, or a *must*- structure with no entry point
(b) A *must*- structure with one entry point, or a *must*+ structure with no entry point
(c) A concretization of Fig. 4(a) seen as a *must*+ structure
(d) A concretization of Fig. 4(b) seen as a *must*- structure

Figure 4. Examples of *Must* Structures and their Concretizations

4.3 Concretization Algorithm of the *must*− Transitions

The concretization of the *must*− structure (see Fig. 4(d) and Fig. 4(b)) is performed as described by Alg. 1. It proceeds in two steps. First, it explores–backwardly and symbolically–a path of the *must*− structure from its final state (state 1 in Fig. 4(d) and Fig. 4(b)). So it computes the weakest precondition *wp_q* of the initial state of the path (for example, either state 4, 6 or 7 in Fig. 4(d)), that allow reaching the final state. Second, it chooses if available a green state in *wp_q*, and executes–this time forwardly and concretely–the path until the final state, which it colours in green in this case. If none of the states in *wp_q* is green, the path is coloured in blue.

Algorithm 1 concretizes *must*− structures that are either trees (see Fig. 4(d)) or strongly connected components having possibly a tail (see Fig. 4(b)). The concretization is performed from a "root" transition $qr \overset{e}{\rightarrow} qr'$: actually it is a transition with no successor, such as transition $2 \rightarrow 1$ in Fig. 4(b), called root because the exploration is performed backwardly. In a strongly connected component without tail, the root is chosen arbitrarily. The outer *while* loop, in line 3, enumerates a set of paths spanning the *must*− structure. The paths are computed backwardly from the root. A path in the algorithm is a sequence of *must*− transitions, where each transition $t = q \overset{e}{\rightarrow} q'$ is stacked as a pair $\langle t, wp\_q \rangle$ that associates the transition itself with the weakest precondition w.r.t. q' of the event e it applies. At each transition branching, one of the transition is explored and added to the path, while the other ones are backed up with the current state of the path into a stack *pb*, for a future exploration (*while* loop, lines 4 to 14). This being done, the current path is recorded in the stack *p*, ready for forward concretization, and the other paths are backed up in the stack *pb*. Concretization starts in line 15 by choosing a state in the current *wp_q* (that of the start of the path), possibly green if any and blue otherwise. Then the *while* loop in lines 16 to 21 concretizes forwardly the current path by pooping out from *p* the transitions one by one, each time applying the event to the last obtained state instance, and propagating the green or blue colour. Last, the conditional lines 22 to 28 backtrack to the nearest backed up branch, for exploring the next path of the tree.

Algorithm 1: ATS Computation of Must- Ball Chains

```
Inputs            : $qr \xrightarrow{e} qr'$: the root transition of a Must- tree
                    $\langle X, I, Init, EvDef\rangle$: an Event System where $EvDef \overset{\text{def}}{=} \{e \overset{\text{def}}{=} a \mid e \in Ev\}$

Input-Outputs     : $\langle Q, Q_0, \Delta, \Delta^+, \Delta^-, C, C_0, \Delta^c, \alpha, \kappa\rangle$: an ATS
Variables         : $t$: the current transition in the Must- tree $q \xrightarrow{e} q'$
                    $wp\_q$: $wp(e, q')$
                    $p$: path traveled, stored as a stack, from the root
                    $pb$: stack of the branchings remaining to handle
                    $B$: set of the branches that reach the current state $q$
                    $M^-$: $\Delta^-$ transitions marking function ($\Delta^- \rightarrow boolean$)

$pb := \emptyset$; $p := \emptyset$; $t := qr \xrightarrow{e} qr'$; $wp\_q := wp(e, qr') \wedge qr$;
 $B := \{qr \xrightarrow{e} qr'\}$; $M^- := \Delta^- \times \{false\}$; push$(p, \langle t, wp\_q\rangle)$;
while $B \neq \emptyset$ do /* spanning path whose root is in $B$ */
    while $B \neq \emptyset$ do /* computation of the unmarked branches that reach $q$ */
        $B := \{s \xrightarrow{f} q \mid \exists(s, f).(s \xrightarrow{f} q \in \Delta^- \wedge \neg M^-(s \xrightarrow{f} q)\}$;
        if $B \neq \emptyset$ then /* at least one branch is in $B$ */
            choose $t$ in $B$; $wp\_q := wp(e, wp\_q) \wedge q$; push$(p, \langle t, wp\_q\rangle)$;
            if $B - \{t\} \neq \emptyset$ then /* at least two branches are in $B$ */
                /* back up of the branches remaining to travel through */
                push $(pb, \langle B - \{t\}, p\rangle)$;
            end

        end
    end
    choose a concrete state $c$ in $wp\_q$, green if possible;
    while $\neg empty\_stack?(p)$ do
        $\langle (q, e, q'), wp\_q\rangle :=$ pop$(p)$;
        $c' := e(c); \alpha(c') := q'; \Delta^c := \Delta^c \cup \{c \xrightarrow{e} c'\}$;
        $\kappa(c') := \kappa(c)$;
        $c := c'$;
    end
    if $\neg empty\_stack?(pb)$ then /* backtrack on the next path to explore */
        $\langle B, p\rangle :=$ pop$(pb)$; choose $t$ in $B$; $wp\_q := wp(e, wp\_q) \wedge q$;
        if $B - \{t\} \neq \emptyset$ then
            push $(pb, \langle B - \{t\}, p\rangle)$;
        end
        $B := \{t\}$; push$(p, \langle t, wp\_q\rangle)$;
    end
end
```

Figure 5 shows the reachable (i.e. green) concrete part of the ATS of the 3MTS of Fig. 3 that was obtained by our method. The full experimental results for this example can be found in Table 1 in Section 5.2 on implementation and experiments: see the CM case, line AP=2 (i.e. 2nd set of abstraction predicates). The 3MTS is constituted of 4 abstract states and 17 abstract transitions, amongst which 1 is both *must*− and *must*+, 1 is *must*− only, 6 are *must*+ only and 9 are *may* only. All the abstract states are covered and 15 out of the 17 abstract transitions are covered (only the transitions $q_2 \xrightarrow{\mathsf{serveCof}} q_1$ and $q_2 \xrightarrow{\mathsf{serveCof}} q_3$ are not covered). We obtain 18 reachable concrete transitions, 15 of which coming from the concrete exploration (CXP) as performed by the algorithm of [JKMV17], and 3 of which being added by the Ball chains instantiations (BCI, in bold in Fig. 5). These 3 concrete transitions are additional instances of abstract transitions already covered by CXP. But contrarily to the CXP instances, they are connected: in Fig. 5 the bold sequence $q_2 \xrightarrow{\mathsf{autoOut}} q_1 \xrightarrow{\mathsf{powerDown}} q_4 \xrightarrow{\mathsf{takePot}} q_4$ was not covered as a sequence by CXP.

5 Implementation and Experiments

This section introduces in Section 5.1 our proof-of-concept tool used to evaluate the impact of *must* transitions concretization on the under-approximation. The experimental results

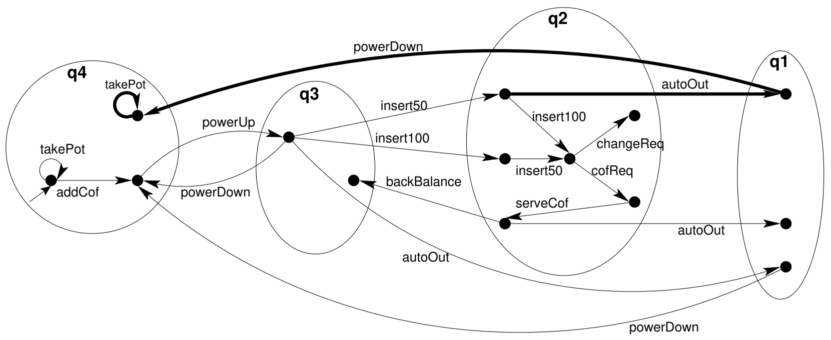

Figure 5. The Reachable Concrete Under-Approximation of the 3MTS of Fig. 3

are then provided in Table 1 in Section 5.2, with their analysis in Section5.3.

5.1 Tool Description

This section describes the general process of test generation using our proof-of-concept tool combining both algorithms. The process is displayed in Fig. 6.

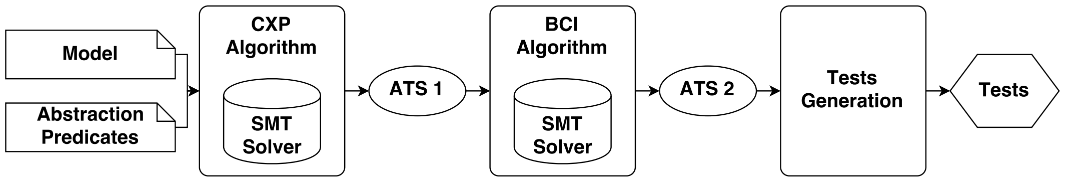

Figure 6. Tool Process

The model, expressed in the event-B language, and the set of abstraction predicates are given as inputs to the tool. With the help of an SMT solver, the abstract transitions modalities as well as a concrete instance of each may transition are computed. This gives a first ATS (see ATS 1 in Fig. 6). Afterwards, again thanks to a solver, a concrete instance of each *must*+ structure and each *must*− structure in the 3MTS is computed. This produces a possibly enriched concretization which, together with the previously computed 3MTS, constitutes an enhanced ATS (see ATS 2 in Fig. 6). The tests generation uses a JAVA implementation of the Chinese Postman algorithm proposed in [Thi03]. The output of this algorithm is the shortest path (a sequence of transitions) covering all *reachable* transitions in the CTS at least once, therefore constituting instantiable tests whose steps are concrete transitions.

Developed in JAVA version 8, the tool is designed as a multipurpose library for event oriented systems analysis and simulation. Among other features, this library proposes pre-implemented easy-to-use functions to compute the abstraction of a system along with its transitions modalities (a 3MTS), compute concrete instances of abstract states or transitions, compute the reachable states and transitions of a CTS, apply a substitution to a state, etc. The library can also be used as a binding for SMT-Lib 2.0 standard compliant SMT solvers (such as Z3 [dMB08]), since it allows building first order logic formulas, translating them into a SMT-Lib 2.0 program, and checking for their satisfiability and looking for a concrete model if applicable.

The complete source code can be downloaded at *https://github.com/stratosphr/_StraTest_*. A complete tutorial about how to install and use the tool can be found at *https://github.com/stratosphr/_StraTest_/wiki.*

Table 1. ATS Computation Results

Sys.	#**Ev**	**AP**	#**AP**	#**AS**	#**AT**	#**M**	#**M**$^-$	#**M**$^+$	**Alg.**	#**AS**$_{reach}$	τ_{AS}(%)	#**AT**$_{reach}$	τ_{AT}(%)	#**CT**	#**CT**$_{reach}$	#**Steps**	**Time (s)**
EL	4	1	2	2	8	6	1	1	CXP	2	100	8	100	14	9	10	00.343
									BCI	2	100	8	100	15	10	11	00.020
		2	2	3	9	5	4	4	CXP	3	100	9	100	18	14	23	00.520
									BCI	3	100	9	100	23	20	34	00.100
		3	2	3	12	9	2	1	CXP	3	100	8	66.67	18	10	17	00.655
									BCI	3	100	8	66.67	20	12	17	00.060
		4	2	4	11	4	5	5	CXP	4	100	11	100	17	16	22	00.914
									BCI	4	100	11	100	23	23	41	00.143
		5	3	6	13	0	13	13	CXP	6	100	13	100	21	16	19	01.233
									BCI	6	100	13	100	32	30	52	00.655
CA	20	1	2	3	23	22	0	1	CXP	3	100	17	73.91	38	19	22	01.579
									BCI	3	100	17	73.91	38	19	22	00.011
		2	6	8	31	28	1	2	CXP	8	100	25	80.65	50	26	41	14.712
									BCI	8	100	25	80.65	52	28	49	00.033
		3	9	9	30	18	7	11	CXP	9	100	28	93.33	52	29	42	25.759
									BCI	9	100	28	93.33	58	35	51	00.201
CM	**11**	1	3	3	13	10	2	3	CXP	3	100	12	92.31	26	14	23	01.188
									BCI	3	100	12	92.31	29	17	27	00.052
		2	**3**	**4**	**17**	**9**	**2**	**7**	**CXP**	**4**	**100**	**15**	**88.24**	**32**	**15**	**25**	**01.428**
									BCI	**4**	**100**	**15**	**88.24**	**35**	**18**	**31**	**00.107**
		3	3	4	32	30	1	1	CXP	4	100	22	68.75	55	24	43	01.903
									BCI	4	100	22	68.75	55	24	43	00.027
		4	3	5	36	32	2	2	CXP	4	80	23	63.89	60	25	46	02.583
									BCI	4	80	23	63.89	61	25	46	00.111
PH	4	1	3	3	12	3	8	3	CXP	3	100	12	100	22	21	37	00.452
									BCI	3	100	12	100	30	29	59	00.144
		2	3	5	19	2	16	15	CXP	5	100	18	94.74	31	29	62	00.930
									BCI	5	100	19	100	45	45	114	00.327
		3	4	7	33	2	30	29	CXP	7	100	32	96.97	54	52	129	01.698
									BCI	7	100	33	100	81	81	202	01.019
		4	6	8	62	1	58	44	CXP	8	100	62	100	88	88	164	02.827
									BCI	8	100	62	100	112	112	198	28.641

5.2 Experimental Results

This section provides the experimental results obtained with the two approaches. We compare the results on a set of four realistic event systems, with three to five different sets of abstraction predicates for each of them. The set of examples is composed of an electrical system (**EL**) taken from [BJM11], a car alarm system (**CA**) taken from [ABJK11], the coffee machine system (**CM**) presented in Section 3, and a phone book system (**PH**) taken from [UL06]. Note that the results for the set of abstraction predicates used for the **CM** system in Section 3 are highlighted using a bold font in Table 1.

The description of the columns appearing in Table 1 is the following: **Sys** indicates the system under test, #**Ev** is the number of events in the system, **AP** is the index of the set of abstraction predicates, #**AP** is the number of abstraction predicates, #**AS** and #**AT** are respectively the number of abstract states and transitions in the 3MTS, #**M** is the number of may transitions having neither the *must*− nor the *must*+ modality, #**M**$^-$ and #**M**$^+$ are respectively the number of *must*− and *must*+ transitions, **Alg.** is the algorithm used (CXP for the *concrete exploration* method presented in [JKMV17] and BCI for the *Ball chains instantiation* method presented in this paper, that completes CXP), #**AS**$_{reach}$ is the number of abstract states covered by the tests, τ_{AS} is the abstract states coverage (in %), i.e. $\frac{\#AS_{reach}}{\#AS}$, #**AT**$_{reach}$ is the number of abstract transitions covered by the tests, τ_{AT} is the abstract transitions coverage (in %), i.e. $\frac{\#AT_{reach}}{\#AT}$, #**CT** is the number of concrete transitions in the 3MTS, #**CT**$_{reach}$ is the number of concrete transitions appearing in the tests, #**Steps** is the number of tests steps, **Time** is the ATS computation time (in seconds).

The main results reside in the τ_{AS}, τ_{AT}, #**CT**, #**CT**$_{reach}$ and #**Steps** columns of Table 1. They show that the Ball chains concretization tends to concretize indeed reachable concrete transitions and to increase the tests steps.

All experiments have been led using the Z3 SMT solver [dMB08] on an Intel(R) Core(TM) i7-4710MQ CPU @ 2.50GHz, with 16GB of RAM.

5.3 Analysing the Results

This section analyses the results presented in Table 1.

Except for two cases (see the **PH** system with the second and third **AP**), the concretization of *must*$-$ and *must*$+$ transitions does not increase the abstraction coverage. This result shows that either all computed *must* transitions were already covered by the CXP method, because they also are *may* transitions, or that the Ball chains concretization did not successfully connect the concretized transition to the reachable part of the first ATS.

While the abstract states and transitions coverage are not improved by the Ball chains concretization, the number of reachable concrete transitions is increased for thirteen cases out of sixteen by 28% on average. This means that, on average, 72% of the total number of reachable transitions in the final CTS where concretized by the CXP algorithm. The difference between the $\#\mathbf{CT}_{reach}$ row for the BCI method and the $\#\mathbf{CT}_{reach}$ row for the CXP method gives the number of new reachable transitions brought by the BCI method. Even if this number may seem low, the contribution brought by the BCI still results in an increased number of test steps by 38.8% on average. Note that on some cases, the Ball chains concretization allows to cover concrete transitions which were previously not reachable in the ATS obtained with the CXP method. This is the case for instance for the **PH** system with the second set of abstraction predicates where twenty-nine concrete transitions were reachable out of the thirty-one. With the Ball chains concretization, fourteen new concrete transitions are computed, which results in sixteen transitions becoming reachable. This means that two previously unreached transitions concretized by the CXP method are reached thanks to the concretization of the Ball chains.

The BCI computation time is correlated with both the number of *must* transitions and the number of branches in the *must* structures (see Fig. 4, Section 4). This correlation clearly appears for the **PH** system with the fourth set of abstraction predicates, where most of the sixty-two abstract transitions have at least one *must* modality, and most have both the *must*$-$ and *must*$+$ modalities. In this case, the *must* concretization is much slower than the CXP ATS computation. This is due to the complexity of the BCI algorithm, that is time exponential in the number of branchings of the *must*$-$ tree.

Finally, let us notice that the progression of reached concrete transitions brought by BCI (i.e. the difference between two successive $\#\mathbf{CT}_{reach}$ rows) is roughly correlated to the number of *must*$-$ transitions (with about 1 or 2 units of difference). This might result from the backward symbolic exploration over the *must*$-$ transitions, that tends to create new start states for the concrete sequences. By now, confirming this point would require further experiments.

As a conclusion, these experimental results show that, although the BCI does not noticeably improve the coverage of the states and transitions of the abstraction, it enhances the concrete exploration of the system by adding new test steps.

6 Related Work

In [NK00] as well as in [PPV07], the set of abstraction predicates is iteratively refined in order to compute a model bisimilar to the initial semantic model when it exists. None of these two methods is guaranteed to terminate, because of the refinement step that sometimes needs to be repeated endlessly. SYNERGY [GHK$^+$06] and DASH [BNR$^+$10] combine under-approximation and over-approximation computations to check safety properties on programs. As we aim to provide an efficient method to build a *reachable* under-approximation of a system that covers all abstract states and all abstract transitions w.r.t. a specification and a set of predicates, our algorithm does not refine the approximation and, consequently, it always terminates.

The closest methods to the algorithm of [JKMV17] are those that are proposed in [GGSV02] and in [VY03]. For Alg. 1 we combine them with those proposed in [Bal04]. These approaches propose algorithms that compute an under approximated concretization of a predicate abstraction covering its abstract states and transitions. Both these methods are exploited for generating tests. In this paper we are particularly concerned with the differences between Alg. 1 and the method presented in [Bal04]. The main difference consists of the fact that the reachability of the under-approximation is guaranteed in [Bal04] and applied to C programs without abstracting the program counter. In contrast, with ES the control flow can be abstracted being implicit and depending of many state variables. To address this problem we propose an heuristics thanks to the coloration method inspired of the algorithm in [VY03]. Our algorithm emphasizes the concretization of Ball's chains from green states, i.e. reached concrete states, computed by the first step (algorithm in [JKMV17]). This heuristics is implemented thanks to an original backward symbolic execution of *must*$-$ chains.

Some other work compute under-approximations for generating tests from an abstraction. The tools Agatha [RGLG03], DART [GKS05], CUTE [SMA05], EXE [CGP$^+$06] and PEX [TdH08] also compute abstractions from models or programs, but only by means of symbolic executions [PV09]. This data abstraction approach computes an execution graph. Its set of abstract states is possibly infinite whereas it is finite with our method.

Our method can be applied to generate tests as the concolic execution of [SMA05]. The concolic method allows the user to generate structural tests of systems covering partially the control flow that must be elaborated. Our approach allows us to generate tests covering the paths defined by the set of abstraction predicates for systems whose control flow is implicitly defined.

7 Conclusion and Perspectives

We have presented a method that computes instantiated model-based tests from a tri-modal abstraction of an event system. It proceeds by adapting to event systems two under-approximation approaches issued from literature: the concrete exploration of [VY03] and the instantiation of the Ball chains of [Bal04]. We have experimented with combining both methods, and evaluated their complementarity. Our experimental results show that the Ball chains instantiations do not improve the coverage of the abstract states and transitions already achieved by the concrete exploration. But they explore more of the concrete system by prolonging the tests sequences issued from the concrete exploration. The added computing cost remains negligible, despite the *must*$-$ instantiation's complexity, as long as the number of *must* transitions is reasonable.

This work opens up a number of perspectives. In particular, the coverage achieved by the added test steps in other terms than abstract states and transitions coverage remains to be evaluated. As presented in [BJM16], the abstraction predicates in our work originate from a *test purpose*, i.e. a peculiar sequencing of some selected event applications. That favours these events to apply as *must*$+$ or *must*$-$ transitions, and thus appearing in the Ball chains. As a result, the additional test steps by our method could cover those peculiar event sequences as defined by the test purpose. This is illustrated by the example in Fig. 5, where the three transitions added by the concretization of the Ball chains are applications of the three events of the test purpose, the guards of which being the abstraction predicates.

More generally, by controlling the size of the abstractions, our method could aim at using k-path coverage criteria for generating tests, rather than abstract state and transition coverage. Finally, among all the Ball chains computed, only those that connected directly to the concrete exploration have been kept in our under-approximation, the reachability of the others being uncertain. Still, seeking for indirect connections towards those uncertain chains could be a valuable effort, since this could provide bridges towards previously ignored connected instance sequences. For example, for the **CM** example of Fig. 5, there are seventeen unreached concrete transitions, and nine of which are in a connected structure (not showed in the figure). Connecting this structure to the one that appears in Fig. 5 would significantly improve the quality of the under-approximation.

References

[ABJK11] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. UML in action: a two-layered interpretation for testing. *ACM SIGSOFT Software Engineering Notes*, 36(1):1–8, 2011.

[Abr96] J.-R. Abrial. *The B Book.* Cambridge Univ. Press, 1996.

[Abr10] J.-R. Abrial. *Modeling in Event-B: System and Software Design.* Cambridge Univ. Press, 2010.

[Bal04] T. Ball. A theory of predicate-complete test coverage and generation. In *FMCO*, volume 3657 of *LNCS*, pages 1–22, 2004.

[BC00] Didier Bert and Francis Cave. Construction of finite labelled transition systems from B abstract systems. In *IFM*, pages 235–254, 2000.

[BJM11] Pierre-Christophe Bué, Jacques Julliand, and Pierre-Alain Masson. Association of under-approximation techniques for generating tests from models. In *TAP*, volume 6706 of *LNCS*, pages 51–68. Springer, 2011.

[BJM15] Hadrien Bride, Jacques Julliand, and Pierre-Alain Masson. Tri-modal under-approximation of event systems for test generation. In *SAC 2015, 30th ACM Symposium On Applied Computing*, pages 1737–1744, Salamanca, Spain, April 2015.

[BJM16] Hadrien Bride, Jacques Julliand, and Pierre-Alain Masson. Tri-modal under-approximation for test generation. *Science of Computer Programming*, 132(2):190–208, 2016.

[BNR$^+$10] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, Robert J. Simmons, SaiDeep Tetali, and Aditya V. Thakur. Proofs from tests. *IEEE Trans. Software Eng.*, 36(4):495–508, 2010.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.

[CGP$^+$06] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *ACM Conference on Computer and Communications Security*, pages 322–335, 2006.

[Dij75] E.W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Com. of the ACM*, 18(8):453–457, 1975.

[Dij76] E.W. Dijkstra. *A Discpline of Programming.* Prentice-Hall, 1976.

[dMB08] L. de Moura and N. Bjorner. An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340, 2008.

[GGSV02] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Generating finite state machines from abstract state machines. In *ISSTA*, pages 112–122, 2002.

[GHJ01] Patrice Godefroid, Michael Huth, and Radha Jagadeesan. Abstraction-based model checking using modal transition systems. In *CONCUR*, pages 426–440, 2001.

[GHK$^+$06] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. Synergy: a new algorithm for property checking. In *SIGSOFT FSE*, pages 117–127, 2006.

[GJ03] P. Godefroid and R. Jagadeesan. On the expressiveness of 3-valued models. In *VMCAI*, volume 2575 of *LNCS*, pages 206–222. Springer, 2003.

[GKOT00] Yuri Gurevich, Philipp W. Kutter, Martin Odersky, and Lothar Thiele. *Abstract State Machines, Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*. Springer, 2000.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[Gur00] Yuri Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Trans. Comput. Log.*, 1(1):77–111, 2000.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Com. of the ACM*, 12(10):576–580, 1969.

[JKMV17] Jacques Julliand, Olga Kouchnarenko, Pierre-Alain Masson, and Guillaume Voiron. Approximating event system abstractions by covering their states and transitions. In *A.P. Ershov Informatics Conference (the PSI Conference Series, 11th edition)*, LNCS, Moscow, Russia, June 2017. To appear.

[LT88] Kim Guldstrand Larsen and Bent Thomsen. A modal process logic. In *LICS*, pages 203–210, 1988.

[NK00] K. S. Namjoshi and R. P. Kurshan. Syntactic program transformations for automatic abstraction. In *CAV*, volume 1855 of *LNCS*, pages 435–449, 2000.

[PPV07] Corina S. Păsăreanu, Radek Pelánek, and Willem Visser. Predicate abstraction with under-approximation refinement. *LMCS*, 3(1:5):1–22, 2007.

[PV09] Corina S. Păsăreanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *STTT*, 11(4):339–353, 2009.

[RGLG03] N. Rapin, C. Gaston, A. Lapitre, and J.-P. Gallois. Behavioral unfolding of formal specifications based on communicating extended automata. In *ATVA*, page 10 pages, 2003.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *ESEC/SIGSOFT FSE*, pages 263–272, 2005.

[TdH08] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .net. In *TAP*, volume 4966 of *LNCS*, pages 134–153, 2008.

[Thi03] H.W. Thimbleby. The directed chinese postman problem. *Software: Practice and Experience*, 33(11):1081–1096, 2003.

[UL06] M. Utting and B. Legeard. *Practical Model-Based Testing*. Morgan Kaufmann, 2006.

[VY03] Margus Veanes and Rostislav Yavorsky. Combined algorithm for approximating a finite state abstraction of a large system. In *ICSE 2003/Scenarios Workshop*, pages 86–91, 2003.