Tri-Modal Under-Approximation for Test Generation

Bride, Hadrien Julliand, Jacques
Masson, Pierre-Alain
FEMTO-ST/DISC, Université de Franche-Comté
16, route de Gray F-25030 Besançon Cedex France
{hadrien.bride, jacques.julliand, pierre-alain.masson}@femto-st.fr

Abstract

This paper presents a method for under-approximating behavioural models with the guarantee that the abstract paths can be instantiated as executions of the models. This allows a model-based testing approach to operate on an abstraction of an infinite or very large behavioural model. We characterize the abstract transitions as may, must+ or must-. This allows us to benefit from Thomas Ball's result that any abstract sequence in the shape of must-*.may.must+* (a Ball chain) can be instantiated as a sequence of connected concrete transitions. We adapt Ball's work aiming at abstracting C programs to the case of event systems, where the instantiated Ball chains might not be reachable from a model's initial state because our method can abstract the control structure. We propose as a solution to this problem to symbolically explore the set of states reachable after a finite number of steps. The Ball chains that start in any of these states are reachable and instantiable. By keeping track of the paths that lead to these starting states, we are able to instantiate with certainty the sequences made of the reached Ball chains with their prefix. This method improves the usual methods that often look for instantiations even though they don't exist for some sequences. We also propose in this paper a way to compute the abstraction w.r.t. predicates that are automatically extracted out of a dynamic property expressed in the Dwyer et al. language. We give experimental results obtained via a proof-of-concept prototype that we have implemented. They show that many Ball chains are exhibited, reached and concretized by our method, and that despite the complexity of symbolic exploration, the Ball chains can be reached within a small number of exploration steps.

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

Revised and extended version of a paper originally published in SAC 2015, 30th ACM/SIGAPP Symposium On Applied Computing, Salamanca, Spain, April 2015, doi: 10.1145/2695664.2695731

1 Motivations

Model-based testing [BJK+05, UL06] provides a valuable means of revealing inconsistencies between a specification and an implementation early during the development process. On the one hand, a formal behavioural model of a system is designed from which a set of tests is computed that ensure a given coverage of the model. The implementation is developed on the other hand by a different team. The explicit state space of the model can be very large or even infinite, which makes it hard to select all the paths in it that conform to a given criterion. Abstraction provides a means to control the size of the state space by gathering the explicit states into symbolic ones.

This paper presents a model-based testing approach that generates tests from an abstraction of a system's behavioural model, to respond to the challenging problem of dealing with state space explosion. But generating tests from an abstraction raises two problems, namely: selecting paths of the abstraction that indeed exist in the model (as it may be over-approximated), and concretizing them so as to obtain executions of the model.

Predicate abstraction [GS97] is usually used for verifying programs as in [FQ02, BHT08]. Its principle is to map the (potentially infinite) set of concrete states onto a finite number of abstract ones, by means of a set of predicates that characterizes each abstract state. This leads to over-approximations when an abstract transition between two abstract states only expresses that there are concrete instances of the transition that go from the source state to the other ; these are called *may* transitions. A path of *may* transitions cannot always be concretized as a path (thus connected) of concrete transitions, because the concrete instances might be disconnected from each other. For test generation, under-approximations are preferable to over-approximations since only feasible paths are considered, though maybe not all of them. This is adequate with the testing paradigm: check some judicious paths though not all of them. This paper focuses on computing paths of abstract transitions that are guaranteed to be instantiable as connected concrete paths. This requires more knowledge on the abstract transitions than their *may* modality.

We consider the tri-modal systems of Thomas Ball [Bal04], which have two additional $must+$ and $must-$ modalities for the abstract transitions. The $must+$ transitions can leave any of the concrete instances of their abstract source state, while the $must-$ ones can reach any of the concrete instances of their abstract target state. It is proved in [Bal04] that a sequence in the shape of $(must-)^* \cdot may \cdot (must+)^*$ (henceforth called a *Ball chain*) is guaranteed to be concretizable as a connected chain of concrete transitions.

We adapt [Bal04] to perform model-based testing from models in the shape of an event system, whose semantics is a transition system. Ball aims in [Bal04] at performing control flow coverage of C programs. In event systems, contrarily to programs, the control structure is implicit and becomes abstracted by the predicate abstraction process. In this context w.r.t. [Bal04], transforming a Ball chain into a model execution requires additional guarantee that it is reachable by a prefix that links it to an initial concrete state of the model. We had proposed in [BJM11] a first step towards an adaptation of Ball's work to the case of event systems. The present paper is a revised and extended version of a SAC 2015 conference paper [BJM15] on the problem of reaching the Ball chains. The main contribution is to compute Ball chains whose reachability is guaranteed. We propose to perform a few steps of symbolic exploration (through $must-$ transitions) of the reachable states, as a prefix to the Ball chains eventually reached. This gives a symbolic state that is added and used as the initial state of our abstraction. We use SMT solvers to concretize the prefixed Ball chains. The result is a set of instantiated executions of the model, that can be used as model-based tests.

W.r.t. the SAC 2015 paper [BJM15], we present here four main additional contributions: (1) we completely formally define the transition modalities characterization; (2) we introduce a new illustrative example (a car alarm system) which is a realistic reactive system; (3) we dedicate a section to a proposition for automatically extracting a set of abstraction predicates out of a requirement, expressed as a dynamic property in the Dwyer et al. [DAC99] style; (4) we enrich the experimental evaluation of our method, in a dedicated section, thanks to a set of now six case studies, some of which issued from industrial collaborations.

The number of tests computed by our method, as well as the behaviours that they exercise, strongly depend on the predicates chosen to perform the abstraction. A method is proposed in [BBJM10] for automatically computing abstraction predicates from a given test purpose. It is based on a partition of the variables domains, which is not well suited for the present work where our intention is to make Ball chains appear. The new method that we propose here relies on collecting the guards or the postconditions, in a purpose of fostering the presence of transitions of the $must+$ or $must-$ type in the abstraction.

The background regarding event systems, their semantics as concrete transition systems, predicate abstraction, tri-modal systems and Ball chains is given in Sec. 2. Two illustrative examples, one of a computation model and the other of a realistic reactive system, are specified in Sec. 3. Our proposition of a method for extracting a set of abstraction predicates from a test purpose is presented in Sec. 4. The main contribution is described in Sec. 5, where we present the method to compute by symbolic exploration some reachable Ball chains and their prefix, and to concretize them as model executions. We also discuss the soundness of the method in this section. In Sec. 6, we briefly describe the proof-of-concept type tool that we have developed on purpose, we illustrate the application of our method on an example, and show its practical feasibility through experimental results. Section 7 positions our approach w.r.t. related work. We conclude the paper in Sec. 8 and indicate further research directions.

2 Background

Our behavioural models are described as Event Systems (ES) whose semantics is defined by means of Concrete labelled Transition Systems (CTS). In this paper we describe the ES in the B syntax [Abr96, Abr10] but our results are generic since they are based on transition systems. We first briefly present the syntax and the semantics of the B event systems. Then we present the concept of predicate abstraction and formalize the abstraction of event systems by means of Tri-modal Transition Systems (3MTS).

2.1 Model Syntax and Semantics

We define B event systems in Def. 1. The events are defined by means of an equation $e \mathrel{\widehat{=}} a$ where e is the name of the event and a is a generalized substitution that defines a guarded action [Dij75] using five primitive substitutions:

- *skip* that is the substitution with no effect,
- $x, y := E, F$ that is a multiple assignment,
- $P \Rightarrow a$ that is a guarded substitution,
- $a_1 [] a_2$ that is a bounded non-deterministic choice,
- $@z.P \Rightarrow a$ that is an unbounded non-deterministic choice $a_{z_1} [] a_{z_2} [] \ldots$ for all the values of z satisfying the condition P.

The guard of a substitution a is denoted as $\mathsf{grd}(a)$. It is defined on the primitive substitutions as:

- $\mathsf{grd}(skip) \stackrel{\text{def}}{=} true$,
- $\mathsf{grd}(x := E) \stackrel{\text{def}}{=} true$,
- $\mathsf{grd}(P' \Rightarrow a) \stackrel{\text{def}}{=} P' \wedge \mathsf{grd}(a)$,
- $\mathsf{grd}(a_1 [] a_2) \stackrel{\text{def}}{=} \mathsf{grd}(a_1) \vee \mathsf{grd}(a_2)$
- $\mathsf{grd}(@z.a) \stackrel{\text{def}}{=} \exists z \cdot \mathsf{grd}(a)$.

By extension, the guard of an event is defined as the guard of its generalized substitution: let $e \mathrel{\widehat{=}} a$ be an event, then $\mathsf{grd}(e) \stackrel{\text{def}}{=} \mathsf{grd}(a)$.

Figure 5 and Fig. 7 (see the whole model in Fig. 12 in appendix A) are examples of event systems illustrating Def. 1.

Definition 1 (Event System) *Let Ev be a set of event names. A B event system is a tuple $\langle X, I, Init, EvDef \rangle$ where:*

- *X is a set of state variables;*
- *I is the invariant: it is a predicate that defines the domain of each variable x of X, and possibly describes invariant properties of the system,*
- *$Init$ is a substitution called* initialization*, such that the invariant holds in any initial state,*
- *$EvDef$ is a set of event definitions, each in the shape of $e \mathrel{\widehat{=}} a$ for any $e \in Ev$, and such that every event preserves the invariant.*

Remark. In Def. 1, according to [Abr10], the events as well as the initialization of a B event system preserve the invariant. In practice the specifier is in charge of carrying the proof of it thanks to a theorem prover in the B method.

The semantics of an event system is defined in [BC00] as a concrete labelled transition system (CTS). Classically, in the rest of the paper, the variables or set of variables are primed to denote their value *after* a substitution, whereas the unprimed versions stand for their *before* value.

Let $e \mathrel{\widehat{=}} a$ be an event. It has a *weakest precondition* w.r.t. a set of target states Q': it is the largest set of states from which applying a, when applicable, always leads to a state of Q'. We denote it as $wp(a, Q')$. An event also has a *strongest postcondition* [Dij76] w.r.t. a set of source states Q: it is the smallest set of states that are guaranteed to be reached after a has been applied from a state of Q. We denote it as $sp(a, Q)$.

As in [BC00], and in order to formally express the strongest postcondition computation, we also introduce the *weakest conjugate precondition* of an event $e \mathrel{\widehat{=}} a$ w.r.t. a set of target states Q': it is defined in [BC00] as the weakest precondition which asserts that it is possible for a to establish Q'. Thus it is always possible to reach Q' by applying a, even if non deterministically a state outside of Q' can also be reached. We denote it as $wcp(a, Q')$. Notice that wp and wcp differ for the guarded and the nondeterministic events. For the latter, wcp captures in addition to wp the states from which applying a leads non-deterministically to a state of Q'.

Let us now formally define wp, wcp and sp. Classically, we directly consider the set of states Q and Q' as predicates of the same name: a set of states Q defines a predicate Q that holds in any state of Q but does not holds in any state not in Q.

We define the weakest precondition w.r.t. the five primitive substitutions:

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

The weakest conjugate precondition is defined as:

$$wcp(a, Q') \stackrel{\text{def}}{=} \neg wp(a, \neg Q').$$

As for the strongest postcondition, it is defined as:

$$sp(a, Q) \stackrel{\text{def}}{=} \exists X.(Q \wedge wcp(a, x'_1 = x_1 \wedge \ldots \wedge x'_n = x_n))$$

where $X = \{x_1, \ldots, x_n\}$ is the set of state variables in the source states of Q and $X' = \{x'_1, \ldots, x'_n\}$ is the set of state variables in the target states of Q'.

2.2 Predicate Abstraction

Predicate abstraction [GS97] is a special instance of the framework of abstract interpretation [CC92] that maps the potentially infinite state space C of a transition system onto the finite state space A of a symbolic transition system *via* a set of n predicates $\mathcal{P} \stackrel{\text{def}}{=} \{p_1, p_2, \ldots, p_n\}$ over the state variables. The set of abstract states A contains 2^n states. Each state is a tuple $q \stackrel{\text{def}}{=} (q_1, q_2, \ldots, q_n)$ 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 $\alpha_{\mathcal{P}} : C \to \mathsf{A}$ 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, 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 $q \xrightarrow{e} q'$, if and only if there exists at

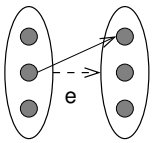

Figure 1: A *may* Transition

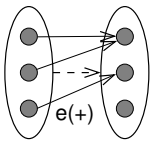

Figure 2: A *must*+ Transition

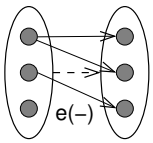

Figure 3: A *must*− Transition

least one concrete transition $c \xrightarrow{e} c'$ where c and c' are concrete states with $\alpha_{\mathcal{P}}(c) = q$ and $\alpha_{\mathcal{P}}(c') = q'$ (see Fig. 1).

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 (see Fig. 2), i.e. they define a total relation between the concrete states of two abstract states. The *must*− transitions are *may* transitions that can reach any concrete state of the abstract target state (see Fig. 3), i.e. they define an onto relation between the concrete states of two abstract states. We do not need to enumerate all the concrete states (there might be an infinity of them) to decide if a transition is of the *may*, *must*+ and/or the *must*− type: we characterize these modalities by means of SAT formulas entrusted to SMT solvers. Let $e \mathrel{\widehat{=}} a$ be an event definition, $q \xrightarrow{e} q'$ is a *may* transition iff $SAT(wcp(a, q') \wedge q)$; it is a *must*+ transition iff $q \Rightarrow wcp(a, q')$ is valid, i.e. $\neg SAT(\neg wcp(a, q') \wedge q)$; it is a *must*− transition iff $q' \Rightarrow sp(a, q)$ is valid, i.e. $\neg SAT(\neg sp(a, q) \wedge q')$. Due to the large expressive power of B event systems, different underlying theories may be needed to express the above first-order formula. Indeed, the examples presented in the paper include only integer and enumerated variables which predicates can be expressed within the theory of *Arithmetic*. However B event systems allow the use of sets and set operations which predicates could efficiently be expressed within the theory of *Bitvectors*. Likewise B event systems allow the use of functions, the predicates involving functions of finite domains and co-domains could efficiently be expressed within the theory of *Arrays* while functions of infinite domains and co-domains could be expressed within the theory *Uninterpreted functions*. Let us note that depending on the underlying theories used, the characterization of the modalities by means of SAT formulas may be undecidable.

2.3 Tri-modal Transition Systems

We define a Tri-modal Transition System (3MTS) in Def. 2. It is a transition system 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 $\mathcal{P} \stackrel{def}{=} \{p_1, p_2, \ldots, p_n\}$ be a set of predicates. Let A be a finite set of abstract states defined by $\{p_1, \neg p_1\} \times \{p_2, \neg p_2\} \times \ldots \times \{p_n, \neg p_n\}$. A tuple $\langle Q, Q_0, \Delta, \Delta^+, \Delta^- \rangle$ is a 3MTS 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,*
- *$\Delta (\subseteq Q \times Ev \times Q)$ is a may labelled transition relation,*
- *$\Delta^+ (\subseteq \Delta)$ is a must+ labelled transition relation,*
- *$\Delta^- (\subseteq \Delta)$ is a must− labelled transition relation.*

The 3MTS that we define are defined by Ball in [Bal04]. They come from the Modal Transition Systems defined in [LT88, GHJ01]. Now, Def. 3 associates an abstraction defined by a 3MTS to an event system.

Definition 3 (3MTS associated to an ES) *Let $ES \stackrel{def}{=} \langle X, I, Init, EvDef \rangle$ be an event system and $\mathcal{P} \stackrel{def}{=} \{p_1, p_2, ..., p_n\}$ be a set of n predicates over variables of X defining a set of 2^n abstract states $\mathsf{A} \stackrel{def}{=} \{p_1, \neg p_1\} \times$*

$\{p_2, \neg p_2\} \times ... \times \{p_n, \neg p_n\}$. *A tuple* $\langle Q, Q_0, \Delta, \Delta^+, \Delta^- \rangle$ *is a 3MTS associated to ES and* $\mathcal{P}$ *if it satisfies the following conditions:*

- $Q \stackrel{def}{=} \{q \in \mathsf{A} | \exists (q', e).(q \stackrel{e}{\rightarrow} q' \in \Delta \vee q' \stackrel{e}{\rightarrow} q \in \Delta)\}$,
- $Q_0 \stackrel{def}{=} \{q | q \in \mathsf{A} \wedge SAT(sp(Init, I) \wedge q)\}$,
- $\Delta \stackrel{def}{=} \{q \stackrel{e}{\rightarrow} q' | q \in \mathsf{A} \wedge q' \in \mathsf{A} \wedge e \widehat{=} a \in EvDef \wedge SAT(wcp(a, q') \wedge q)\}$,
- $\Delta^+ \stackrel{def}{=} \{q \stackrel{e}{\rightarrow} q' | q \in \mathsf{A} \wedge q' \in \mathsf{A} \wedge e \widehat{=} a \in EvDef \wedge \neg SAT(\neg wcp(a, q') \wedge q)\}$,
- $\Delta^- \stackrel{def}{=} \{q \stackrel{e}{\rightarrow} q' | q \in \mathsf{A} \wedge q' \in \mathsf{A} \wedge e \widehat{=} a \in EvDef \wedge \neg SAT(\neg sp(a, q) \wedge q')\}$.

An example of a 3MTS, whose ES is described in Fig. 5, can be seen in Fig. 6. The four abstract states named q_0 to q_3 appear as rounded rectangular boxes. The predicates p_0 and p_1 from which they are defined are given explicitly in Sec. 3. The abstract transitions of Δ are represented as dashed arrows labelled by an event name, with the possible mentions + and/or − indicating respectively when they are in Δ^+ or Δ^-.

2.4 Over and Under Approximations Based on 3MTS

An *execution* of a CTS or of a 3MTS is a finite or infinite sequence of transitions that begins in an initial state. We denote by $q_0 \stackrel{e_0}{\rightarrow} q_1 \stackrel{e_1}{\rightarrow} \ldots$ where $q_i \stackrel{e_i}{\rightarrow} q_{i+1} \in \Delta$ for $i \geq 0$ an abstract execution, and by $c_0 \stackrel{e_0}{\rightarrow} c_1 \stackrel{e_1}{\rightarrow} \ldots$ a concrete execution. We say that $q_0 \stackrel{e_0}{\rightarrow} q_1 \stackrel{e_1}{\rightarrow} \ldots$ and $c_0 \stackrel{e_0}{\rightarrow} c_1 \stackrel{e_1}{\rightarrow} \ldots$ 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 property that holds on such an abstraction also holds 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 smaller 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 a concrete test.

The Δ transition relation of Def. 3 defines an over-approximation. Indeed, the existence of a concrete transition gives birth to a *may* abstract transition. But an execution of two consecutive *may* transitions $q \stackrel{e}{\rightarrow} q'$ and $q' \stackrel{e'}{\rightarrow} q''$ may not always have a similar connected concrete counterpart. Think for example of the case where no concrete target state of e is a concrete source state of e'.

T. Ball defines in [Bal04] a method to compute an under-approximation by means of the Δ, Δ^+ and Δ^- abstract transition relations of a 3MTS.

2.5 Ball's Universal Under-Approximation

Thomas Ball proves in [Bal04] 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. 4, 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 write it as a regular expression[1] by means of $(must-)^* \cdot may \cdot (must+)^*$. In [Bal04], Ball defines an under-approximation called L as the set of abstract states reachable from an abstract initial state by a Ball chain.

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

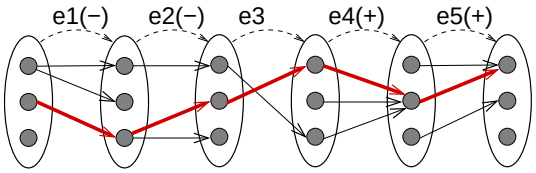

Figure 4: A Concretization of a $(must-)^* \cdot may \cdot (must+)^*$ Sequence of Abstract Transitions

In order to formalize L, Ball defines a reachability function for a 3MTS. Let Q be a set of states and Δ ($\subseteq Q \times Ev \times Q$) be a transition relation. The reachability function on Δ and $Q' \subseteq Q$ is defined by $\mathcal{R}[\Delta](Q') \stackrel{\text{def}}{=} \mu Z.(Q' \cup \Delta(Z, Ev))$ where μ is the least fix-point and $\Delta(Z, Ev)$ is the relational image of the sets Z and Ev by Δ. L is defined by Ball from the set of initial states Q_0 as:

$$L \stackrel{\text{def}}{=} \{q'' \mid \exists\ (q, e, q') \cdot (q \in \mathcal{R}[\Delta^-]\ (Q_0)\ \wedge (q' = q \vee q \stackrel{e}{\rightarrow} q' \in \Delta)\ \wedge q'' \in \mathcal{R}[\Delta^+](\{q'\})\}.$$

Thus an abstract state q'' is in L if there is: a (possibly empty) sequence of $must-$ transitions leading from an initial abstract state to q; one or zero may transition from q to q'; a (possibly empty) sequence of $must+$ transitions from q' to q''.

3 Illustrative Example

We illustrate our approach in this paper by means of two examples whose descriptions follow in this section. The first one is a simple computational model. Although its state space is infinite, the abstraction size is small enough to be entirely drawn graphically, when we apply our method to it in Sec. 6.1. The second one is more realistic. It is a reactive system with which we illustrate our method of Sec. 4, for computing sets of abstraction predicates according to dynamic properties expressed in a practical language. Its state space is finite, but the size of the produced abstractions is greater than the size of the abstractions produced from the first example.

3.1 Small Computational Model

The specification is given in Fig. 5. It models a conditional computation over three variables x, y, z. Its semantics is an infinite CTS for unbounded integers. Our abstraction method computes the finite 3MTS of Fig. 6 from the set of predicates $\mathcal{P}_0 = \{p_0, p_1\}$ where $p_0 \stackrel{\text{def}}{=} z = 1$ and $p_1 \stackrel{\text{def}}{=} x > y$.

$$\begin{array}{lcl}
X & \widehat{=} & \{x, y, z\} \\
I & \widehat{=} & x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge z \in 0..1 \\
Init & \widehat{=} & x, y, z := 0, 0, 0 \\
e_1 & \widehat{=} & z = 1 \wedge x > y \Rightarrow \\
 & & @a.(@b.(a \in \mathbb{N} \wedge b \in \mathbb{N} \wedge b \geq a \Rightarrow x, y := a, b)) \\
e_2 & \widehat{=} & z = 1 \wedge y \geq x \Rightarrow x := y + 1 \\
e_3 & \widehat{=} & z = 1 \wedge x = 7 \wedge y = 11 \Rightarrow x := 17 \\
e_4 & \widehat{=} & z = 0 \Rightarrow \\
 & & @a.(@b.(a \in \mathbb{N} \wedge b \in \mathbb{N} \wedge b < a \Rightarrow x, y, z := a, b + 5, 1)) \\
e_5 & \widehat{=} & z = 0 \Rightarrow \\
 & & @a.(@b.(a \in \mathbb{N} \wedge b \in \mathbb{N} \wedge b < a \wedge b \leq 5 \Rightarrow x, y, z\ := a, b, 1))
\end{array}$$

Figure 5: A Small Illustrative Model

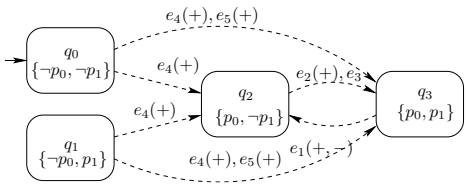

Figure 6: An Abstraction for the Small Model of Fig. 5

3.2 Reactive System - Car Alarm System Model

This case study, issued from [ABJK11], is a protection system for cars. The system triggers an alarm sound and turns on the warning lights in case of a person trying to open the doors illegally. The model fragment shown in Fig. 7 specifies the properties and behaviours of the vehicle parts that are concerned with the car alarm system: Bell (Be), Alarm Controller (AC), Doors (Do), Lock of doors (Lo), Warning lights (Wa), Glasses (Gl), Children Security System (CS), other Lights (than warning ones) (Li), Movement of the car (Mv) and a chronometer that counts the delay between the locking of the doors and the activation of the alarm. The chronometer is represented by means of two variables: Tr models if the chronometer is triggered or not, and De models the delay from the release of the chronometer. This model describes events that modify the state of the car alarm system: closing or opening the doors and the glasses, locking and unlocking the doors, triggering or deactivating the bell and the children security system, switching on or off the warnings and the lights, engaging or stopping the alarm, making the car move or stop, triggering or stopping the chronometer, etc. We specify a User (Us) to be either authorized or unauthorized, for modelling that the doors are opened either legally (with the keys of the car) or in an illegal way. The behaviour of this system has to meet the following requirements:

- the alarm is automatically engaged in a delay of five to ten seconds once the authorized user locks the doors,
- the bell and the warning lights are activated if an unauthorized user opens a door when the alarm is engaged,
- the alarm is automatically disengaged when an authorized user unlocks the doors,
- the user cannot lock the doors if a glass is opened,
- to put the car in movement, the doors must be unlocked and closed,
- to switch on or off the children security, the doors must be opened, for accessing the manual lever allowing to do it,

A fragment of the specification is given in Fig. 7. It is modelled by an event system whose semantics is a finite CTS. It potentially has a maximum of $2^{11} \times 11$ (=22528) states. There are fewer reachable states because of the constraints between variables that are formalized by the invariant properties I in Fig. 7.

4 Automatic Computation of Abstraction Predicates From a Test Purpose

The 3MTS computed from an event system depends on a set of predicates chosen to perform the abstraction. This section contains a proposition for automatically computing a set of abstraction predicates for an event system according to a *test purpose*.

$$
\begin{array}{rcl}
X & \widehat{=} & \{Be, AC, Do, Lo, Wa, Us, Gl, CS, Li, Mv, Tr, De\} \\
I & \widehat{=} & Be, AC, Do, Lo, Wa, Us, Gl, CS, Li, Mv, Tr \in 0..1 \wedge De \in 0..11 \wedge \\
 & & (AC = 1 \wedge Us = 1) \Rightarrow Do = 0 \wedge \\
 & & \text{/* if alarm engaged and user authorized, then doors closed */} \\
 & & Be = 1 \Rightarrow AC = 1 \wedge \\
 & & \text{/* if bell noisy, then alarm controller activated */} \\
 & & AC = 0 \Rightarrow Be = 0 \wedge \\
 & & \text{/* if alarm controller deactivated, then bell silent */} \\
 & & Wa = 1 \Leftrightarrow Be = 1 \wedge \\
 & & \text{/* warning lights iff bell noisy */} \\
 & & Wa = 0 \Leftrightarrow Be = 0 \wedge \\
 & & \text{/* warning lights switched off iff bell silent */} \\
 & & Lo = 1 \Rightarrow (Do = 0 \vee Us = 0) \wedge \\
 & & \text{/* if doors locked, then doors closed and user authorized */} \\
 & & Us = 1 \Rightarrow Be = 0 \wedge \\
 & & \text{/* if user authorized, then bell silent */} \\
 & & Lo = 1 \Rightarrow Gl = 0 \wedge \\
 & & \text{/* if doors locked, then glasses closed */} \\
 & & \ldots \\
Init & \widehat{=} & Be, AC, Do, Lo, Wa, Us, Gl, CS, Li, Mv, Tr, De := \\
 & & 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0 \\
Doors\_Opening & \widehat{=} & Tr = 0 \wedge Mv = 0 \wedge Do = 0 \wedge \\
 & & (Us = 0 \vee (Us = 1 \wedge Lo = 0 \wedge AC = 0)) \Rightarrow Do := 1 \\
Bell\_Activation & \widehat{=} & Tr = 0 \wedge Mv = 0 \wedge AC = 1 \wedge Do = 1 \Rightarrow Wa, Be := 1, 1 \\
Bell\_Stop & \widehat{=} & Tr = 0 \wedge Mv = 0 \wedge Be = 1 \wedge AC = 0 \Rightarrow Wa, Be := 0, 0 \\
Children\_Security\_Activation & \widehat{=} & Tr = 0 \wedge Mv = 0 \wedge Lo = 0 \wedge CS = 0 \wedge Do = 1 \Rightarrow CS := 1 \\
Children\_Security\_Deactivation & \widehat{=} & Tr = 0 \wedge Mv = 0 \wedge Lo = 0 \wedge CS = 1 \wedge Do = 1 \Rightarrow CS := 0 \\
Incr\_Chronometer & \widehat{=} & Tr = 1 \wedge De < 10 \Rightarrow De := De + 1 \\
 & & \ldots \\
User\_Authorized & \widehat{=} & Tr = 0 \wedge Mv = 0 \wedge Us = 0 \wedge Be = 1 \\
 & & \Rightarrow Us, Be, Wa, AC, Lo := 1, 0, 0, 0, 0 \\
User\_Unauthorized & \widehat{=} & Tr = 0 \wedge Mv = 0 \wedge Us = 1 \wedge Do = 0 \wedge AC = 1 \wedge Lo = 1 \\
 & & \Rightarrow Us := 0
\end{array}
$$

Figure 7: Car Alarm System Model

A test purpose (TP) formalizes a test intention expressed by a validation engineer, in the shape of a scenario that this engineer wishes to exercise. This comes in complement to a pure structural model-based testing approach, by providing a "selected paths" coverage criterion. This provides a set of tests that target some particular executions, as can be useful for example in security testing [PMG04, JMT08].

4.1 Test Purpose Language

We propose to make use of the well-known pattern language of Dwyer *et al.* [DAC99] to express a test purpose. This language provides an approach for specifying dynamic properties in the shape of a pattern delimited by a scope. While each combination of a pattern and a scope can be given a semantics into temporal logics such as LTL or CTL, the language is intended to be more intuitive to use than logics by specification practitioners. Actually we use the test property specification language of [CCDJT11], with the compositional automata-based semantics of scopes and patterns of [TJD+14], to express our test purposes. It is based on the Dwyer *et al.*'s pattern language.

Patterns capture recurring temporal properties (e.g., *Always A*, *Never B*, *B Responds to A*, ...), while scopes delimit the execution intervals in which the patterns should hold (e.g., *After Q*, *Between Q and R*, ...). The parameters A, B, Q, R referred to in these examples may either stand for event calls or for state predicates. A detailed technical presentation of the language and its semantics is out of the scope of our paper, so we refer the reader to [DAC99, CCDJT11] for a more comprehensive presentation of this pattern/scope language.

As an example, consider the following dynamic requirement, taken from the Car Alarm System case study (see Section 3.2): *The bell and the warning lights are activated if an unauthorized user opens a door when the alarm is engaged.* It can be formalized as the TP "*Bell_Activation* **Responds to** *Doors_Opening* **Between** *User_Unauthorized* **and** *User_Authorized*". Here the pattern "*Bell_Activation* **Responds to** *Doors_Opening*" occurs within the scope "**Between** *User_Unauthorized* **and** *User_Authorized*". Such a property is in the shape of "P' **Responds to** P **Between** Q **and** R". It corresponds (in [DAC99, DAC98]) to the following LTL property: $\Box((Q \wedge \neg R \wedge \Diamond R) \Rightarrow (P \Rightarrow (\neg R \,\mathsf{U}\, (P' \wedge \neg R)))\,\mathsf{U}\, R)$ when P', P, Q and R are state properties. Using the compositional automata-based semantics of scopes and patterns of [TJD+14], its semantics is the set of infinite executions accepted by the Büchi automaton represented in Fig. 8.

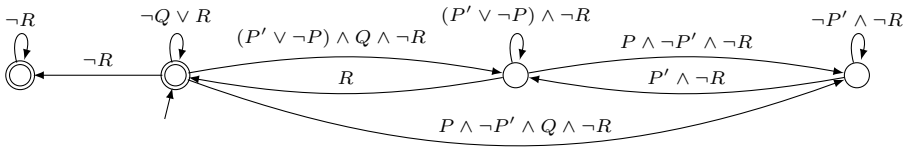

Figure 8: Büchi Automaton for the Property P' responds to P between Q and R

4.2 Abstraction Predicates Computation

We propose in this section different ways for automatically computing a set of abstraction predicates from a test purpose and an event system. These computations rely on collecting the predicates that appear in the test purpose either as explicit state predicates, or as guards of the events called in the test purpose, or as post-conditions of the events called in the test purpose.

Let TP be a test purpose expressed in the Dwyer et al. pattern/scope shape. We denote by $\mathsf{PR}_{\mathsf{TP}} = \{pr_1, \ldots, pr_{n_p}\}$ the set of all the state predicates explicitly cited in TP. We denote by $Ev_{\mathsf{TP}} = \{e_1, \ldots, e_{n_e}\}$ the set of names of all the events explicitly called in TP.

We first define $\mathsf{AP}_{\mathsf{Pred}}$ as the set of abstraction predicates issued from the explicit state predicates of TP. We simply have that

$$\mathsf{AP}_{\mathsf{Pred}} = \mathsf{PR}_{\mathsf{TP}}.$$

Then we propose two methods for completing this set of abstraction predicates with predicates issued either from the guards of the event called in TP, or from their post-conditions.

4.2.1 Method 1: Collection of the Guards

The idea behind this method is to collect the guards of the events called in TP, expectedly to have abstraction predicates favouring the appearance of *must*+ transitions (every concrete state is left by an event whose guard is true). The set $\mathsf{AP}_{\mathsf{Guard}}$ of the guards of the events called in TP is coarsely defined as:

$$\mathsf{AP}_{\mathsf{Guard}} = \bigcup_{e \in Ev_{\mathsf{TP}}} \{\mathsf{grd}(e)\}.$$

Notice that it is possible to enhance the granularity of this set by first putting each guard into either CNF (conjunctive normal form) or DNF (disjunctive normal form), and to collect as abstraction predicates

every clause (resp. conjunctive clause) of the CNF (resp. DNF). The set of clauses can then be reduced by removing duplicates, negated forms of previous clauses, and true or false clauses.

For method 1, the final set of abstraction predicates is $\mathsf{AP}_g = \mathsf{AP}_{\mathsf{Pred}} \cup \mathsf{AP}_{\mathsf{Guard}}$.

4.2.2 Method 2: Collection of the Post-Conditions

The idea behind this method is to collect the post-conditions of the events called in TP, expectedly this time to have abstraction predicates favouring the appearance of $must-$ transitions (every concrete state on which the post-condition holds is reached). The set $\mathsf{AP}_{\mathsf{Post}}$ of the post-conditions of the events called in TP is defined as:

$$\mathsf{AP}_{\mathsf{Post}} = \bigcup_{e \in Ev_{\mathsf{TP}}} \{sp(e, I)\}$$

where I is the invariant in the event system.

As explained for $\mathsf{AP}_{\mathsf{Guard}}$, this set can be applied a CNF or DNF decomposition to enhance its granularity.

For method 2, the final set of abstraction predicates is $\mathsf{AP}_p = \mathsf{AP}_{\mathsf{Pred}} \cup \mathsf{AP}_{\mathsf{Post}}$.

4.2.3 Application to an example

We consider the test purpose TP given as an example in Sec. 4.1:

$\mathsf{TP} \stackrel{\text{def}}{=}$ *Bell_Activation* **Responds to** *Doors_Opening*
Between *User_Unauthorized* **and** *User_Authorized*.

No state predicate is explicitly used in TP so that

$$\mathsf{AP}_{\mathsf{Pred}} = \emptyset.$$

Method 1 - collection of the guards This method collects the guards of the events explicitly called in TP. There are four of them:
Bell_Activation, *Doors_Opening*, *User_Unauthorized* and *User_Authorized*. Their guards are respectively (see Fig. 7):

- $Tr = 0 \wedge Mv = 0 \wedge AC = 1 \wedge Do = 1$,
- $Tr = 0 \wedge Mv = 0 \wedge Do = 0 \wedge (Us = 0 \vee (Us = 1 \wedge Lo = 0 \wedge AC = 0))$,
- $Tr = 0 \wedge Mv = 0 \wedge Us = 1 \wedge Do = 0 \wedge AC = 1 \wedge Lo = 1$,
- $Tr = 0 \wedge Mv = 0 \wedge Us = 0 \wedge Be = 1$.

Once put in DNF and reduced, these guards give the resulting set of five abstraction predicates:

$$\begin{aligned}\mathsf{AP}_{\mathsf{Guard}} \stackrel{\text{def}}{=} \{&Tr = 0 \wedge Mv = 0 \wedge AC = 1 \wedge Do = 1,\\ &Tr = 0 \wedge Mv = 0 \wedge Do = 0 \wedge Us = 0,\\ &Tr = 0 \wedge Mv = 0 \wedge Do = 0 \wedge Us = 1 \wedge Lo = 0 \wedge AC = 0,\\ &Tr = 0 \wedge Mv = 0 \wedge Us = 1 \wedge Do = 0 \wedge AC = 1 \wedge Lo = 1,\\ &Tr = 0 \wedge Mv = 0 \wedge Us = 0 \wedge Be = 1\}.\end{aligned}$$

In this example $\mathsf{AP}_g = \mathsf{AP}_{\mathsf{Guard}}$ since $\mathsf{AP}_{\mathsf{Pred}} = \emptyset$.

Method 2 - collection of the postconditions This method collects the postconditions, from the states that satisfy the invariant I, of the events explicitly called in TP.

If no DNF decomposition is performed then we simply have that $\mathsf{AP}_p = \mathsf{AP}_{\mathsf{Post}}$ (since $\mathsf{AP}_{\mathsf{Pred}}$ is empty) where the set of postconditions is:

$$\begin{aligned}\mathsf{AP}_{\mathsf{Post}} \stackrel{\text{def}}{=} \{&Wa = 1 \wedge Be = 1 \wedge I,\\ &Do = 1 \wedge I,\\ &Us = 0 \wedge I,\\ &Us = 1 \wedge Be = 0 \wedge Wa = 0 \wedge AC = 0 \wedge Lo = 0 \wedge I\}.\end{aligned}$$

5 Instantiation Method for the Tri-Modal Under-Approximations

In [Bal04], Ball performs control flow coverage of C programs, and his abstraction predicates do not abstract the program counter, that is kept explicit in the abstraction. By contrast in our case, we have no such thing as a program counter. As the control structure of an event system is implicitly defined by the guards of the events, it depends on any state variable. Since the tester defines its abstraction predicates from test purposes, intended at exercising targeted functionalities, he can abstract the control flow. In this framework the Ball chains, though concretizable, may not be reachable: the initial abstract state may include other concrete states than the initial ones. This does not occur with programs where the unabstracted program counter guarantees all executions to begin in an initial abstract state. We propose in this section to symbolically explore the reachable abstract states, in order to detect those that start a Ball chain. The states reachable from a set of states Q after a number n of event applications are characterized as an initial abstract state $R_Q(n)$. We add it to the 3MTS with Q being the states reached by initialization. The Ball chains that start from it are guaranteed to be concretizable as tests. We also describe in this section how we instantiate these prefixed Ball chains as model executions, and discuss how they can be played as tests. Then we discuss the soundness of the method and illustrate its application to the example of Sec. 3.

5.1 Symbolic Execution from the Initial States

In order to expand the set of initial states, we use static symbolic execution [Kin76]. For that we define $R_Q(n)$ in Def. 4, where $sp(a, q)$ refers to the *strongest postcondition* [Dij75] 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.

Definition 4 *Let $R_Q(n)$ be the set of states of an Event System reachable from the set of states Q after applying a maximum of $n \geq 0$ event(s). This set is characterized by the following predicate:*
$R_Q(0) = Q$,
$R_Q(i+1) = R_Q(i) \vee \bigvee_{\{a|e \widehat{=} a \in EvDef\}} \mathrm{sp}(a, R_Q(i))$.

Let $ES \stackrel{\text{def}}{=} \langle X, I, Init, EvDef\rangle$ be an event system and $Q_0 = sp(Init, I)$. $R_{Q_0}(n)$ abstractly specifies all states reachable from the initial states of an event system after applying at most n events. We define in Def. 5 a data structure called *reachable state tree*, denoted as RST_n, to store $R_{Q_0}(n)$ with the detail of the symbolic execution paths that lead to any abstract state of $R_{Q_0}(n)$.

Fig. 9 shows the RST_1 of the ES of Fig. 5. Each node appears as a rounded rectangular box featuring the node label and its characteristic predicate.

Definition 5 (Reachable State Tree of an ES) *A Reachable State Tree of an $ES \stackrel{def}{=} \langle X, I, Init, \{e \widehat{=} a | e \in Ev\}\rangle$ is a directed acyclic graph $\langle L, l_0, R, C\rangle$ where:*

- *L is a non-empty set of nodes,*

- $l_0 \in L$ *is the root node,*
- $R \in L \times Ev \times L$ *is a labelled transition relation that links a father to its children,*
- $C \in L \times F(X)$ *(where $F(X)$ is the set of first order logic formulas over the set of variables X) is a one-to-one relation that associates every node of L with a predicate characterizing a set of concrete states of the ES.*

By initializing the initial node l_0 such that $(l_0, Q_0) \in C$, the structure RST_n of ES can simply be obtained by a depth-first application of each event until depth n. The cost of this computation is exponential with respect to n.

With this structure, the disjunction of all the predicates stored in an RST_n is equivalent to $R_{Q_0}(n)$ (i.e. $R_{Q_0}(n) = \cup\{p \mid \exists l.(l \in L \wedge (l,p) \in C)\}$). Also, we can recover the symbolic execution path of any state in $R_{Q_0}(n)$ by traversing the tree from the root node l_0 to the abstract state(s) of L containing it.

5.2 Test Computation and Instantiation Method

We now describe how to compute symbolic test sequences (i.e sequences of abstract transitions starting by the initialization of an ES), and instantiate them.

We characterize $R_{Q_0}(n)$ as a predicate p_{d_0}. It characterizes an abstract state q_{d_0}, that we add to the 3MTS of the ES, and make it its unique initial state. We link it to the other states by computing the *must*− and *may* transitions that come out of it and join them. Notice that we do not need to compute the *must*+ transitions since they will be considered as *may* ones when they start a Ball chain. This modified 3MTS is called an n-Derived 3MTS (n-D3MTS). Its definition is given by Def. 6. Figure 10 shows the 1-D3MTS of the 3MTS of Fig. 6, with the RST_1 of Fig. 9.

Definition 6 (n-Derived 3MTS) *Let $\langle Q, Q_0, \Delta, \Delta^+, \Delta^- \rangle$ be the 3MTS and $\langle L, l_0, R, C \rangle$ be the RST_n associated with an ES provided with a set of events $EvDef \stackrel{def}{=} \{e \mathrel{\widehat{=}} a | e \in Ev\}$. An n-Derived Tri-modal Transition System (n-D3MTS) is a 3MTS $\langle Q_d, q_{d_0}, \Delta_d, \Delta^+, \Delta_d^- \rangle$ where:*

- q_{d_0} *is a new abstract state characterized by*

$$p_{d_0} \stackrel{def}{=} \bigvee_{q \in \{p | \exists l.(l \in L \wedge (l,p) \in C)\}} q,$$

- $Q_d \stackrel{def}{=} Q \cup \{q_{d_0}\}$,
- $\Delta_d \stackrel{def}{=} \Delta \cup \{q_{d_0} \xrightarrow{e} q' | q_{d_0} \xrightarrow{e} q'$ *is a* may *transition*$\}$,
- $\Delta_d^- \stackrel{def}{=} \Delta^- \cup \{q_{d_0} \xrightarrow{e} q' | q_{d_0} \xrightarrow{e} q'$ *is a must− transition*$\}$.

We then compute, using the n-D3MTS, the set of abstract transition sequences in the shape of $(must-)^* \cdot may \cdot (must+)^*$ that start from q_{d_0}. We perform this path enumeration by means of a Depth-First Search based exploration algorithm. It first traverses the *must*− transitions (possibly none), then traverses if possible a *may* transition, and finally traverses the *must*+ transitions (possibly none). As this set can be infinite (e.g., when there is a *must*− or a *must*+ loop), we use a parameter m that restricts the number of times an abstract transition appears in the sequence and denote this set T_m. Every abstract transition sequence of T_m can then be instantiated, which means that there exists at least one corresponding concrete transition sequence starting in an initial concrete state of the Event System. This concrete sequence is obtained by instantiating first the *may* abstract transition by SMT valuation. The *must*− (resp. *must*+) transitions are then instantiated by recursive backward (resp. forward) SMT valuation. Finishing the backward instantiation of the *must*− transitions gives a concrete state of $q_{d_0}(R_{Q_0}(n))$ that starts this instantiated sequence. Using the RST_n allows for recovering the abstract transition sequence that led to it. These transitions are also *must*− ones (see Sec. 5.4), and are in their turn concretized by recursive backward SMT valuation. This ends in a concrete initial state of the event system. Finally, this now prefixed instantiated sequence is an execution of the event system, that exercises the behaviours isolated by the predicate abstraction.

5.3 Test Execution

These executions can be seen as scenarios that exercise the system according to the purpose emphasized by the abstraction predicates. The way an execution can be turned into a test depends on the controllability and observability of the system under test (SUT). To control the SUT, we assume that it can be instrumented so as to react to the commands as appearing in the execution of the model. We also assume that the non-deterministic choices that the system could make can be controlled, so that we can reproduce the choices made by the SMT-solver. For the system's observability, we assume that we can interpret the outputs produced by the SUT in terms of model variables values. These assumptions hold in the framework that we have sketched in Section 1 of this paper, where the testing and development teams work side by side during the development stage of the system.

Let us make the strong assumption that: (i) the SUT is instrumented so that the state variables are totally observable (i.e. the internal state is known at any time), (ii) the events fireability as well as the non-deterministic choices of the events are totally controllable. Under these assumptions, the model executions can be directly used as a set of off-line tests. Consider a test of the illustrative example where the event e_1 is activated, and the values $a = 10$ and $b = 11$ have been chosen by the solver. In the real system, there is no guarantee that these values of a and b will actually be chosen. Playing the test thus requires that these values will be chosen by the instrumented SUT when this occurrence of e_1 is activated. The conformance of the SUT to the model can be evaluated by comparing the outputs produced by the system with the variable values predicted by the model for this execution. The test passes if they conform, otherwise it fails.

If the system cannot be completely controlled but that it is still observable, then the executions have to be played as on-line tests: once a non-deterministic choice has been operated by the SUT, the tests are adapted dynamically. At first, by checking that the choice conforms to a choice allowed by the model. If not, the test fails. Otherwise, the solver is forced to operate the same choice, and the end of the test sequence is re-calculated accordingly. In the end, the test passes if the outputs of the system have conformed to some variable values predicted by the model, otherwise it fails.

5.4 Soundness of the Method

The soundness of our method is based on the three following properties, used to prove Prop. 7:

(i) Every abstract transition sequence in the shape of $(must-)^* \cdot may \cdot (must+)^*$ is instantiable (proved in [Bal04] and explained in Sec. 2.5).

(ii) Every transition $(l \xrightarrow{e} l')$ in an RST_n is a $must-$ transition by definition of the strongest postcondition.

(iii) The node l_0 of an RST_n is the set of initial states of an ES by definition.

Property 7 *Let $A = \langle Q, q_{d_0}, \Delta, \Delta^+, \Delta^- \rangle$ be an n-D3MTS. Any abstract sequence $q_{d_0} \xrightarrow{e_0} q_1 \ldots \xrightarrow{e_{n-1}} q_n$ of A that is a Ball chain is instantiable from an initial state of the CTS that is the semantics of the ES from which the n-D3MTS is derived.*

Proof 8 *Any abstract sequence $q_{d_0} \xrightarrow{e_0} q_1 \ldots \xrightarrow{e_{n-1}} q_n$ that is a Ball chain can be instantiated as a concrete sequence $c_{d_0} \xrightarrow{e_0} c_1 \ldots \xrightarrow{e_{n-1}} c_n$ according to (i), but c_{d_0} is not necessarily an initial concrete state.*

The initial abstract state q_{d_0} is the union of all the nodes $l_i \in L$ of the RST_n used to construct the n-D3MTS. So there exists $l_i \in L$ such that $c_{d_0} \in l_i$.

By definition of an RST_n, for all $l_i \in L$, there exists a sequence $l_0 \xrightarrow{e_{l_1}} l_1 \ldots \xrightarrow{e_{l_i}} l_i$ that is a sequence in the shape of $(must-)^*$ according to (ii). According to (i), this sequence can also be instantiated as a concrete sequence $c_{l_0} \xrightarrow{e_{l_1}} c_{l_1} \ldots \xrightarrow{e_{l_i}} c_{d_0}$ where c_{l_0} is an initial concrete state. By concatenating both these concrete sequences, we obtain a concrete sequence starting from an initial concrete state of the ES according to (iii).

By property 7 we can define our under-approximation with respect to Ball's under-approximation noted L in [Bal04] by replacing the set of initial state Q_0 by the previously defined symbolic set of reachable states $\{q_{d_0}\}$. We obtain:

$L \stackrel{\text{def}}{=} \{q'' \mid \exists\, (q, e, q') \cdot (q \in \mathcal{R}[\Delta^-]\ (\{q_{d_0}\})\ \wedge (q' = q \vee q \xrightarrow{e} q' \in \Delta)\ \wedge q'' \in \mathcal{R}[\Delta^+](\{q'\})\}$.

6 Illustrative Application and Experimentation of the Method

We illustrate in this section our method by applying it to the small computational model of Fig. 5 in Sec. 3. We also give preliminary experimental results that were obtained by means of an experimental research prototype, that we have implemented in a proof-of-concept perspective.

6.1 Application to the Example

Suppose we have a TP that calls the events e_1 and e_2 of the ES of Fig. 5. By collecting the guards of these events as the set of abstraction predicates, we get the set $\mathcal{P}_0$ of Section 3.1: $\mathcal{P}_0 = \{p_0, p_1\}$ where $p_0 \stackrel{\text{def}}{=} z = 1$ and $p_1 \stackrel{\text{def}}{=} x > y$.

We first construct the RST_n for $n = 1$ that is graphically shown in Fig. 9. Then the 3MTS of Fig. 6 is computed. Due to the exponential complexity of the construction of RST_n, we have limited n to 1 for the sake of our illustration's readability. However, the instantiation step presented hereunder already shows the benefits of our method with such a small value of n.

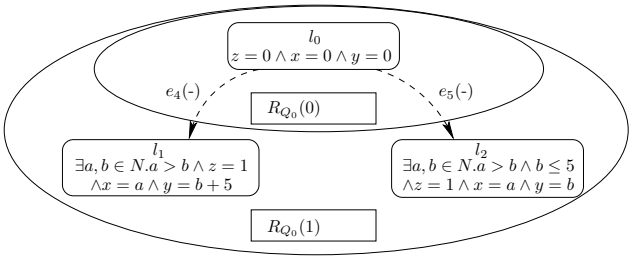

Figure 9: Graphical Representation of RST_1

Finally we construct the 1-D3MTS (see Fig. 10) from the 3MTS of Fig. 6 and the RST_1 of Fig. 9.

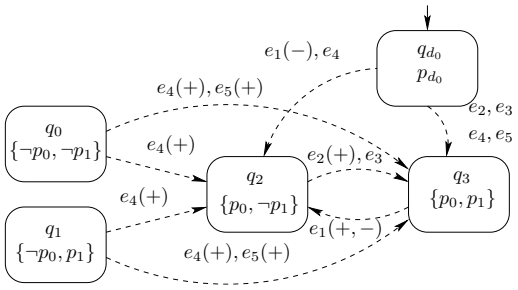

Figure 10: Graphical Representation of the 1-D3MTS Obtained from Fig. 6

Let us now compute the set of abstract transition sequences in the shape of $(must-)^* \cdot may \cdot (must+)^*$ that start by q_{d_0}. With at most two cycles (i.e. $m = 2$) these sequences are the following:

$$
\begin{aligned}
&q_{d_0} \xrightarrow{e_1-} q_2 \xrightarrow{e_3} q_3 \xrightarrow{e_1+} q_2 \xrightarrow{e_2+} q_3 \xrightarrow{e_1+} q_2 \xrightarrow{e_2+} q_3,\\
&q_{d_0} \xrightarrow{e_1-} q_2 \xrightarrow{e_2} q_3 \xrightarrow{e_1+} q_2 \xrightarrow{e_2+} q_3 \xrightarrow{e_1+} q_2 \xrightarrow{e_2+} q_3,\\
&q_{d_0} \xrightarrow{e_2} q_3 \xrightarrow{e_1+} q_2 \xrightarrow{e_2+} q_3 \xrightarrow{e_1+} q_2 \xrightarrow{e_2+} q_3,\\
&q_{d_0} \xrightarrow{e_3} q_3 \xrightarrow{e_1+} q_2 \xrightarrow{e_2+} q_3 \xrightarrow{e_1+} q_2 \xrightarrow{e_2+} q_3,\\
&q_{d_0} \xrightarrow{e_4} q_2 \xrightarrow{e_2+} q_3 \xrightarrow{e_1+} q_2 \xrightarrow{e_2+} q_3 \xrightarrow{e_1+} q_2,\\
&q_{d_0} \xrightarrow{e_4} q_3 \xrightarrow{e_1+} q_2 \xrightarrow{e_2+} q_3 \xrightarrow{e_1+} q_2 \xrightarrow{e_2+} q_3,\\
&q_{d_0} \xrightarrow{e_5} q_3 \xrightarrow{e_1+} q_2 \xrightarrow{e_2+} q_3 \xrightarrow{e_1+} q_2 \xrightarrow{e_2+} q_3.
\end{aligned}
$$

We observe, as expected by the predicates chosen (from the guards of e_1 and e_2), that each sequence applies mainly the events e_1 and e_2 in various contexts. The sequences are finite, and of the maximal length allowed by the limitation of the number of times a cycle is used.

Any of these sequences can be instantiated as described in Sec. 5.2. We illustrate it by instantiating (with the models as returned by the SMT solver) the following sequence, that is a reduction of the fourth sequence of the previous suite, obtained with $m = 1$: $q_{d_0} \xrightarrow{e_3} q_3 \xrightarrow{e_1+} q_2 \xrightarrow{e_2+} q_3$.

We first instantiate the *may* abstract transition (i.e. $q_{d_0} \xrightarrow{e_3} q_3$): $\{z{=}1,x{=}7,y{=}11\} \xrightarrow{e_3} \{z{=}1,x{=}17,y{=}11\} \xrightarrow{e_1+} q_2 \xrightarrow{e_2+} q_3$.

We then use the last instance previously obtained (i.e. $\{z = 1, x = 17, y = 11\}$) to instantiate forwardly the following *must*+ transitions (i.e. e_1 and e_2): $\{z{=}1,x{=}7,y{=}11\} \xrightarrow{e_3} \{z{=}1,x{=}17,y{=}11\} \xrightarrow{e_1+} \{z{=}1,x{=}28,y{=}52\} \xrightarrow{e_2+} \{z{=}1,x{=}53,y{=}52\}$.

N.B. The values given here in the application of the event e_1 result from the choice of the a and b values ($a = 28$ and $b = 52$) as performed by the solver during our experience.

Then the first instance in the sequence is used to instantiate backwardly the preceding *must*− transitions (none here). Finally we need to instantiate the concrete transitions leading from one of the initial states of the ES to the beginning of our previously obtained transition sequence. Thanks to the RST_1 showed in Fig. 9, we find that the concrete state $\{z = 1, x = 7, y = 11\}$ belongs to l_1 and can be reached through the following sequence: $\xrightarrow{init} l_0 \xrightarrow{e_4-} l_1$. This sequence is instantiated backwardly as (with the values $a = 7$ and $b = 6$ chosen by the solver):

$$\xrightarrow{init} \{z = 0, x = 0, y = 0\} \xrightarrow{e_4-} \{z = 1, x = 7, y = 11\}.$$

By concatenating the last sequence with the previous one, we have a concrete sequence applying four events that begins with the initial state of the ES.

6.2 Experimentation

We have questioned the feasibility of our approach by means of some preliminary experiments whose results we report here. The aim was to assess on some realistic event systems that the modalities of the events could indeed be evaluated and a 3MTS could be computed, despite the high complexity of the methods involved. Also, we were seeking answers to the following questions. What depth of symbolic exploration could be reached in practice? Would this exploration reach Ball chains, so as to produce tests? How would the method scale? And finally, how would the tests be related to the TP they originate from?

For this purpose we have developed a proof-of-concept prototype software. We have considered a set of six realistic event systems of increasing size, some of them issued from industrial collaborations. These event systems were taken back from various previous work without modification so as not to influence the experiment and threaten the validity of the results. Two intuitive TP have been expressed for each event system, from which off-line tests intended for a totally controllable and observable SUT have been computed. We first measured (see Table 3) how many *must* transitions were produced, and in how much time the 3MTS was computed. Then we measured (see Table 4) until what depth could the symbolic exploration be performed, how many tests and test steps were produced, how they cover the abstract model and how they cover the events called in the TP.

6.2.1 Proof-of-Concept Prototype Implementation

Our proof-of-concept prototype software, fully automating the proposed test generation approach, has been developed in Java. It requires however an external solver (e.g., Z3, CVC4) to carry out the main computations. Its block diagram is shown in Fig. 11.

The main inputs are: a test purpose (1) as well as the Event B model (2) from which the tests will be generated. First, according to the method chosen by the user, a set of abstraction predicates (4) is derived from the test purpose according to Sec. 4.2 (3). Second, the symbolic exploration (7) with depth n chosen by the user (6) and the Tri-Modal abstraction (5) based on the previously derived predicate set (4)

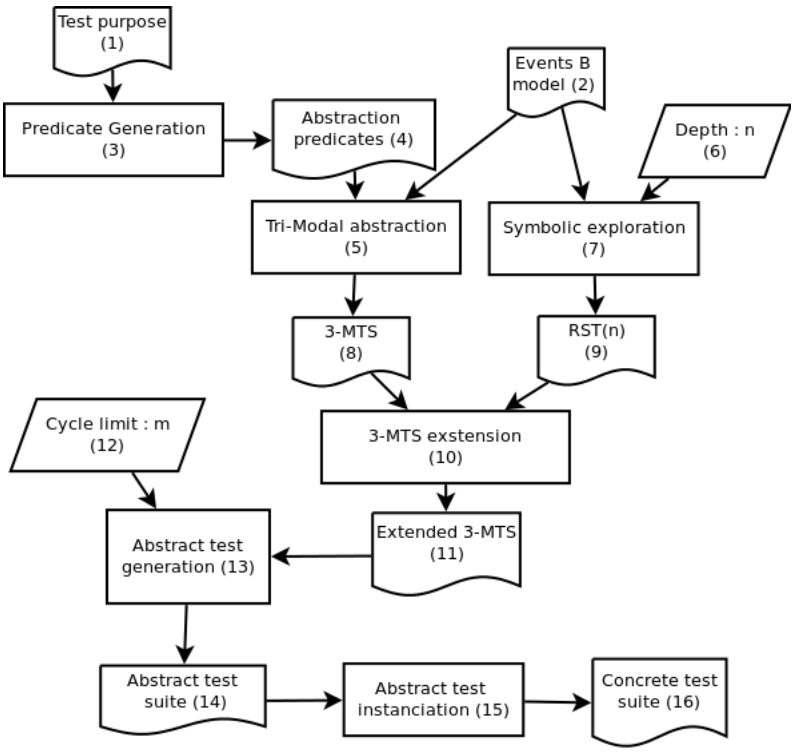

Figure 11: Prototype Software Description

are performed with the help of an SMT-solver to produce the corresponding Reachable State Tree (9) and 3MTS (8). Third, according to the proposed method an extended 3MTS (11) is constructed (10) from the 3MTS (8) and the Reachable State Tree (9). Fourth, an abstract test suite (14) with a maximum number m of cycles set by the user (12), is generated (13). Finally the abstract tests of this suite (14) are instantiated (15) using an SMT-solver in order to produce a concrete test suite (16).

In order to simplify its use, our tool has been integrated into a friendly user interface and all inputs and outputs are described by means of XML files with dedicated formats.

6.2.2 Experimental Results

All experimentations and time measurements have been realized on a personal laptop running under an Intel® Core i5-2410M CPU @ 2.3 GHz.

We indicate in Table 1 the characteristics of each system studied: its ID, its name, its number of variables (# Var.) and of events (# Ev.), its origin and its potential maximum number of states (MAX # States). Notice that this maximal number gives a better idea of the model's size than simply its number of variables or events, because the former depends on the domain sizes of the variables.

ID	Name	# Var.	# Ev.	Origin	MAX # States
ES	Electrical System	3	4	[CLE09, BJM11]	$< 2^6$
QD	Qui Donc	3	4	[UL06]	$< 2^6$
CM	Coffee Machine	4	8	Teaching example	$\sim 2^{16}$
CA	Car Alarm	12	20	[ABJK11]	$< 2^{15}$
CC	Credit Card	19	17	Industrial case study	$\sim 2^{120}$
FW	Front Wiper	15	25	Industrial case study	$\sim 2^{27}$

Table 1: System Characteristics

Table 2 gives for each system two test purposes, each one identified by an acronym in the first column. They express properties naturally relevant to the examples considered.

TP-ID	Test Purpose
ES_1	Commute **Responds to** Tic **Globally**
ES_2	**Eventually** Repair **After** Fail
QD_1	**Always** NbTries=0 **Between** HangUp **And** Start
QD_2	HangUp **Responds to** Timeout **Between** NbTries=1 **And** NbTries=0
CM_1	**Never** serveCoffee **Between** Balance = 0 **And** Balance < 60
CM_2	**Never** takeMago **Between** AutoOut **And** powerDown
CA_1	Bell_Activation **Responds to** Doors_Opening **Between** User_Unauthorized **and** User_Authorized
CA_2	**Always** (S=0) **Between** Lock_Doors **And** UnLock_Doors
CC_1	CARD_failed_pin **Responds to** CARD_process_pin **After** C_counter ≥ 2
CC_2	**Never** DB_ok **Between** CARD_success_pin **and** DB_operation_not_done
FW_1	**Never** power_down **Between** wash_action_begin **And** wash_action_end
FW_2	block_FW_engine **Responds to** action_commodo_down **Globally**

Table 2: Test Purposes Considered

According to the predicate generation method proposed in Sec. 4.2, Table 3 reports measurements on the resulting tri-modal abstractions. It indicates the method used to compute the abstraction predicates (Guard/Post: collection of the guards or of the postconditions), the number of resulting predicates (#Pr.), abstract states (#St.), and modal transitions (#may: may transitions that are neither *must*+ nor *must*−, #m+: *must*+ transitions, #m-: *must*− transitions) in the 3MTS. The %must column shows what proportion (in %) of the abstract transitions are *must* ones. Finally the Time column tells the time spent (in seconds) for computing the 3MTS.

ID	TP-ID	Method	#Pr.	#St.	#may	#m+	#m-	%must	Time (s)
ES	ES_1	Guard	2	3	12	1	2	20.0	0.4
		Post	2	2	8	1	1	20.0	0.2
	ES_2	Guard	2	3	9	4	4	47.1	0.9
QD	QD_1	Guard	3	3	12	3	8	47.8	0.5
		Post	3	5	19	15	16	62.0	0.8
	QD_2	Guard	4	7	33	19	30	59.8	1.6
		Post	6	8	62	44	58	62.2	3.2
CM	CM_1	Guard	3	4	30	7	10	36.2	1.3
		Post	3	6	52	12	12	31.6	2.2
	CM_2	Guard	3	7	33	19	30	59.8	1.6
		Post	3	6	24	18	10	53.9	2.4
CA	CA_1	Guard	9	27	68	23	14	35.2	66.3
		Post	6	17	63	8	6	18.2	31.4
	CA_2	Guard	3	4	34	3	2	12.8	2.4
		Post	2	3	23	1	0	4.2	1.7
CC	CC_1	Guard	3	6	52	2	12	21.2	5.6
		Post	4	7	102	3	24	20.9	8.7
	CC_2	Guard	2	4	24	3	4	22.6	2.7
		Post	3	4	28	2	8	26.3	2.9
FW	FW_1	Guard	3	8	99	22	0	18.2	42.5
	FW_2	Guard	3	8	139	6	4	6.7	33.5

Table 3: Tri-Modal Abstraction Results

Finally, Table 4 reports the results of the abstract test generation process. We have set the parameter m (maximum number of repetition of a transition) to 1 in each of our experiments. For each experimental case, Table 4 gives: the symbolic exploration depth (# Depth), the number of abstract tests generated (#

Tests), the number of test steps (event activations, including those of the symbolic exploration) achieved by these tests (# Steps), the abstract states and transitions coverage (State(%) and Trans(%)). The TP (%) column shows, for the transitions of the D3MTS that are occurrences of the events invoked in the TP, the ratio of the ones appearing in the test suite to their total number. Otherwise said, it is the proportion of event calls in the TP that are really activated by the test suite.

TP-ID	Method	# Depth	# Tests	# Steps	State(%)	Trans(%)	TP(%)
ES_1	Guard	3	8	21	100	100	100
	Post	3	7	18	100	100	100
ES_2	Guard	3	18	107	100	100	100
QD_1	Guard	2	213	1613	100	100	100
	Post	2	234780	3080456	100	100	100
QD_2	Guard	0	71680	837716	100	100	100
	Post	1	100000+	1000000+	100	100	100
CM_1	Guard	3	44	246	100	26	100
	Post	3	66	289	100	16	100
CM_2	Guard	3	228	1722	80	24	100
	Post	3	1098	13384	100	33	100
CA_1	Guard	2	30	162	37	17	100
	Post	3	22	72	47	24	100
CA_2	Guard	3	34	119	100	40	100
	Post	3	17	36	100	66	100
CC_1	Guard	6	7	32	50	15	50
	Post	3	8	21	42	12	100
CC_2	Guard	6	7	43	50	29	66
	Post	3	7	19	50	25	33
FW_1	Guard	2	3	7	25	3	66
FW_2	Guard	2	13	34	100	7	100

Table 4: Abstract Test Generation Results

Table 3 shows computation times that remain practicable on our realistic examples. This table also shows good proportions of *must* transitions, around 33% in average, which roughly confirms the ability of our method to make *must* transitions appear. However this indicator has to be refined to see if the *must* transitions are those that apply the events of the TP. This is what the % TP column in Table 4 is about.

We observe in Table 4 that the abstract states and transitions coverage tends to decrease as the examples grow in size. This coverage is low for the last three examples (Car Alarm System, Credit Card and Front Wiper). This is because our test intention is not to cover all the events of the specification, but only those that are concerned with the TP. Their proportion tends to decrease as the system's size grows. Indeed, a TP typically calls 3 or 4 events, which may be all of them for a small size example, but not for a larger one. As for the relevance of the tests to the TP, our method computes predicates likely to make the transitions applying the events of the TP be of the *must*+ or *must*− type, and thus appearing in the Ball chains. This is mainly confirmed by the empirical observation (see the TP (%) column), although the Credit Card and Front Wiper TP appear to be less well covered. The growing number of possible interleavings of the TP event calls in larger examples may increase the number of Ball chains. Possibly not all of them have been reached in our experimentation. Also, two of the TP are in the shape of *Never* e_1 *Between* e_2 *And* e_3, which means that the event e_1 should normally not occur between the occurrences of e_2 and e_3. Thus abstractions favouring the occurrences of e_2 and e_3 might, as a side effect, prevent e_1 to occur, or *vice versa*. Our preliminary results have not permitted to discriminate between these possible factors. This does not contradict the general observation that the events exercised by the tests are those used in the TP.

Also to be noted, the Qui Donc experiment showed a blow up of the number of tests generated, although we have purposely driven the test generation towards specific test purposes. A particularity of this model is that despite its apparent small size (3 variables and 4 events), it specifies a large number of behaviours, with multiple "if" constructions nested in each other. As a majority of the transitions are of the *must* type in

the abstraction (see the % *must* column in Table 3), our criterion consisting of enumerating all the possible combinations of *must*+ and *must*− transitions provokes that blow up, resulting in an under-approximation that becomes close to the concrete system. Otherwise said, as the abstraction is not that abstract, the abstract tests are badly abstracting the concrete paths. In such a case, other test generation criteria may be used to better select the most relevant tests (i.e. a subset of the Ball's under-approximation). Also the model could be re-written so as to split the events into elementary behaviours.

As a general conclusion of these first experiments, and in response to the experimentation questions, we see that our method remains applicable on realistic size examples: tri-modal transition systems have successfully been computed in a practicable time, from a TP and an event system. A small number of symbolic exploration steps have allowed to reach many Ball chains, and thus to compute abstract instantiable tests. These tests exercise the events that are called in the TP. Nevertheless, our experiments have revealed limits in the scalability of our approach. For example, exploring more Ball chains in a large size example could not surprisingly require a deepest symbolic exploration. So the method has to be refined to prove scalable, but it proves at least for now to be feasible. It can be used in the early steps of the development process, with relatively small size event systems that specify the system to be implemented.

7 Related Work

In [NK00] as well as in [PPV07], the set of abstraction predicates is iteratively refined in order to compute a bisimulation of the initial 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 at proposing an efficient MBT [UL06] method, our algorithm always terminates because it does not refine the approximation.

The method presented in [GGSV02] applies the same two steps: computation of a predicate abstraction and generation of tests. The abstraction predicates are the guards, reduced in DNF, of all the events, whereas we define them from the guards or postconditions only of the events invoked in a TP. Their abstraction generation algorithm computes a concrete instance of a *may* under-approximation instantiating the abstract transitions from the initial concrete state. By contrast, we propose an under-approximation that is computed at the abstract level, thanks to the *must*+ and *must*− modalities. Then the concrete test generation is different: in [GGSV02] a Chinese Postman algorithm is applied to cover the set of concrete transitions, whereas we concretize the reachable abstract Ball chains.

Other papers present work aiming at generating tests from abstractions. The tools Agatha [RGLG03], CUTE [SMA05], EXE [CGP$^+$06] and PEX [TdH08] also compute abstractions from models or from programs, but only by means of symbolic execution [PV09]. 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 DART [GKS05] tool generates test inputs to a program for driving it along targeted paths. It abstracts the branch statements by means of symbolic constraints, and generates inputs for activating a given program branch thanks to a constraint solver. The approach is reinforced in the SMART [God07] tool by computing function *summaries* that are re-used whenever the function is called. These summaries are predicates expressing input preconditions and output postconditions. Similarly to SMART our abstraction predicates are based on pre (the guards) and post-conditions. But in SMART they are intended to achieve structural coverage of the control flow of a program that calls the summarized function without enumerating all the paths of the control structure of this function, whereas our intention is to enforce particular event calls as formalized in a TP.

The method implemented in STG [JJRZ05] uses abstractions defined by the user and modelled by IOSTS (Input Output Symbolic Transition System). They use test purposes synchronized with abstractions, both defined as IOSTS. Then, the synchronized product allows generating tests after an optimization 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. But there are three differences. First, any abstraction is computed from a set of predicates defined from a test purpose and a behavioural model, whereas STG uses user-defined

abstractions. Second, an optimization is performed by the abstraction computation by using the invariant properties (that do not exist in an IOSTS) specified in the B models used implicitly in our method. It allows, for the weakest precondition computation, to minimize 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 of [SMA05], we use symbolic execution. But concolic tools do not use predicate abstraction. Concolic execution performs a concrete execution and at the same time collects the symbolic path constraints. Moreover, hybrid concolic execution [MS07] combines random generation of input values.

Finally, we have also used Ball's chain in [BJM11] for generating tests from a tri-modal abstraction of an event system. Although Ball's under-approximation was combined with an existential under-approximation in order to try to instantiate the abstract tests generated, there was no guarantee that these tests were possible to instantiate. On the contrary in this paper, all the tests generated are guaranteed to be instantiable.

8 Conclusion and Further Work

We have presented a method for generating model-based tests that deals with the state space explosion problem. Our approach is based on abstractions of infinite or very large behavioural models. A test is an execution of the model in this context. The abstraction is a tri-modal transition system for which the Ball chains are guaranteed to be concretizable as connected sequences of concrete (i.e. of the model) transitions. Our proposition is to select only those Ball chains that are reachable from concrete initial states. We turn them into model executions by concretizing them and computing a prefix that links them to a concrete initial state, adapting Ball's work for programs to event systems. This is performed by a symbolic exploration of the set of reachable abstract states. In this way we give solutions to both the problems expressed in Sec. 1: select some paths of the abstraction and concretize them.

We have defined a predicate that characterizes the set of states reachable in n steps, i.e. after n events have been applied. We have given a procedure to compute a symbolic execution tree, which records the successions of abstract states reached by the successive event applications. The predicate that characterizes the union of all such abstract states defines an abstract initial state that we add to the tri-modal transition system of the model. This allows us to select the Ball chains that originate from it, and prefix them with paths computed from the symbolic execution tree. We concretize each prefixed chain by SMT solving, which results in a set of model executions. They can be played as off-line or on-line model-based tests, according to the controllability and observability of the system under test.

We have also proposed a method for automatically computing a set of abstraction predicates with respect to a test purpose. The method relies on collecting either the guards or the postconditions of the events, in order to enhance the number of $must+$ or $must-$ transitions, and thus the number of Ball chains, related to the original test purpose. Our first experimental results on some realistic case studies confirm that our method allows to discover many Ball chains in the abstraction, and that even small values of n quickly enhance the number of Ball chains reached and finally concretized as tests. Thus they show the feasibility of our method on realistic examples, even though these examples are actually of a modest size. The scalability to huge industrial applications is still to be assessed, which will necessitate a robuster tool than the experimental prototype that we use at present.

The complexity of symbolic exploration limits our computation of the reachable state space to that reachable in a few steps. Using heuristics rather than exhaustive enumeration for applying successive events could lead us much faster towards targeted Ball chains starts. This could improve the coverage of the model by tests by discovering more reachable Ball chains for such targeted behaviours. Also, we intend to compute and combine several tri-modal systems rather than one as considered in this paper. This would allow more behaviours of the system to be tested, with one set of predicates per expected behaviour. The symbolic exploration of the reachable states would only have to be performed once, and could be added to all the tri-modal transition systems computed. Another research direction will be to enhance the link between the abstraction predicates and the test intention formalized in the test purpose TP. By now we measure the

relevance of our tests in terms of coverage of the events called in the TP. A finer criterion would be to measure the relevance w.r.t the semantics of the TP in terms of temporal succession of the events. That could be obtained by measuring the coverage of the paths of the Büchi automaton of the TP. Although our tests indeed call the events of TP as expected, we could filter the Ball chains that either meet or contradict the scenario expressed by the TP.

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.

[BBJM10] F. Bouquet, P.-C. Bué, J. Julliand, and P.-A. Masson. Test generation based on abstraction and test purposes to complement structural tests. In *A-MOST*, pages 54–61, Paris, 2010.

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

[BHT08] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Program analysis with dynamic precision adjustment. In *ASE*, pages 29–38, 2008.

[BJK+05] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*. Springer, 2005.

[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.

[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.

[CCDJT11] Kalou Cabrera Castillos, Frédéric Dadeau, Jacques Julliand, and Safouan Taha. Measuring test properties coverage for evaluating UML/OCL model-based tests. In B. Wolff and F. Zaidi, editors, *ICTSS'11, 23-th IFIP Int. Conf. on Testing Software and Systems*, volume 7019 of *LNCS*, pages 32–47, Paris, France, November 2011. Springer.

[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.

[CLE09] CLEARSY. ATELIER B 4.0, Langage B Manuel Utilisateur version 1.2, 2009.

[DAC98] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Property specification patterns for finite-state verification. In *FMSP 1998, Second Workshop on Formal Methods in Software Practice*, pages 7–15, 1998.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE'99, 21st Int. Conf. on Software Engineering*, pages 411–420, Los Angeles, California, USA, 1999. ACM.

[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 Discipline of Programming*. Prentice-Hall, 1976.

[FQ02] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *POPL*, pages 191–202, 2002.

[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.

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

[God07] Patrice Godefroid. Compositional dynamic test generation. In *POPL'07*, pages 47–54. ACM, 2007.

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

[JJRZ05] B. Jeannet, T. Jéron, V. Rusu, and E. Zinovieva. Symbolic test selection based on approximate analysis. In *TACAS*, volume 3440 of *LNCS*, pages 349–364, 2005.

[JMT08] J. Julliand, P.-A. Masson, and R. Tissot. Generating security tests in addition to functional tests. In *AST*, pages 41–44. ACM Press, 2008.

[Kin76] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

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

[MS07] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *ICSE*, pages 416–426, 2007.

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

[PMG04] Bruce Potter and Gary Mc Graw. Software security testing. *IEEE Security and Privacy*, 2(5):32–36, 2004.

[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.

[TJD+14] Safouan Taha, Jacques Julliand, Frédéric Dadeau, Kalou Cabrera Castillos, and Bilal Kanso. A compositional automata-based semantics and preserving transformation rules for testing property patterns. *FAC, Formal Aspects of Computing*, 27(4):641–664, July 2014.

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

A Car Alarm System

This appendix is intended to the reviewers to provide them with additional technical content regarding the case study car alarm system. It presents the whole model of the car alarm system. The model is composed of 12 variables and 19 events. Fig. 7 shows all of the event of the model.

$X \triangleq \{Be, AC, Do, Lo, Wa, Us, Gl, CS, Li, Mv, Tr, De\}$

$I \triangleq Be, AC, Do, Lo, Wa, Us, Gl, CS, Li, Mv, Tr \in 0..1 \wedge De \in 0..11 \wedge$
$(AC = 1 \wedge Us = 1) \Rightarrow Do = 0 \wedge$
/* if the alarm is engaged and the user authorized, the doors are closed */
$Be = 1 \Rightarrow AC = 1 \wedge$ /* if the bell noisy, the alarm controller is activated */
$AC = 0 \Rightarrow Be = 0 \wedge$ /* if the alarm controller is deactivated, the bell is silent */
$Wa = 1 \Leftrightarrow Be = 1 \wedge$ /* The warning light iff the bell is noisy */
$Wa = 0 \Leftrightarrow Be = 0 \wedge$ /* Warning is switched off iff the bell is silent */
$Lo = 1 \Rightarrow (Do = 0 \vee Us = 0) \wedge$ /* if the doors are locked, the doors are closed and the user is authorized */
$Us = 1 \Rightarrow Be = 0 \wedge$ /* if the user is authorized, the bell is silent */
$Lo = 1 \Rightarrow Gl = 0 \wedge$ /* doors locked, then glasses closed */
...

$Init \triangleq Be, AC, Do, Lo, Wa, Us, Gl, CS, Li, Mv, Tr, De := 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0$

$Bell\_Activation \triangleq Tr = 0 \wedge Mv = 0 \wedge AC = 1 \wedge Do = 1 \Rightarrow Wa, Be := 1, 1$
/* if the alarm is engaged and a door is opened the bell noisy */

$Bell\_Stop \triangleq Tr = 0 \wedge Mv = 0 \wedge Be = 1 \wedge AC = 0 \Rightarrow Wa, Be := 0, 0$
/* if the alarm is disengaged whereas the bell noisy the bell is stopped */

$Doors\_Opening \triangleq Tr = 0 \wedge Mv = 0 \wedge Do = 0 \wedge (Us = 0 \vee (Us = 1 \wedge Lo = 0 \wedge AC = 0)) \Rightarrow Do := 1$
/* if the doors are closed then the non authorized user can open the doors */
/* or the authorized user can open the doors when they are unlocked */

$Doors\_Closing \triangleq Tr = 0 \wedge Mv = 0 \wedge Do = 1 \wedge Us = 1 \Rightarrow Do := 0$
/* if the doors are opened then the authorized user can close it */

$User\_Authorized \triangleq Tr = 0 \wedge Mv = 0 \wedge Us = 0 \wedge Be = 1 \Rightarrow Us, Be, Wa, AC, Lo := 1, 0, 0, 0, 0$
/* if the user is non authorized and the bell noisy then the user becomes authorized */

$User\_Unauthorized \triangleq Tr = 0 \wedge Mv = 0 \wedge Us = 1 \wedge Do = 0 \wedge AC = 1 \wedge Lo = 1 \Rightarrow Us := 0$
/* if the user is authorized and the alarm is engaged then the user becomes non authorized */

$Alarm\_Deactivation \triangleq Tr = 0 \wedge Mv = 0 \wedge Us = 1 \wedge AC = 1 \Rightarrow AC := 0$
/* if the user is authorized and the alarm is engaged then alarm becomes disengaged */

$Alarm\_Activation \triangleq Tr = 1 \wedge Mv = 0 \wedge Lo = 1 \wedge Do = 0 \wedge AC = 0 \wedge De \geq 5 \wedge De \leq 10 \Rightarrow$
$AC, Tr, De := 1, 0, 0$
/* if the doors are locked, the alarm is engaged between five and ten ut */

$Doors\_Locking \triangleq Tr = 0 \wedge Mv = 0 \wedge Do = 0 \wedge Us = 1 \wedge Lo = 0 \wedge Gl = 0 \wedge AC = 0 \Rightarrow$
$Lo, Li, Tr, De := 1, 0, 1, 0$
/* if the doors are unlocked and the user authorized, they are locked and the chronometer is activated */

$Doors\_Unlocking \triangleq Tr = 0 \wedge Lo = 1 \wedge Do = 0 \wedge Us = 1 \Rightarrow Lo := 0$
/* if the doors are locked and the user authorized, they are unlocked */

$Glasses\_Opening \triangleq Tr = 0 \wedge Gl = 0 \wedge Lo = 0 \Rightarrow Gl := 1$
/* if the glasses are closed and the doors unlocked the glasses can be opened */

$Glasses\_Closing \triangleq Tr = 0 \wedge Gl = 1 \wedge Lo = 0 \Rightarrow Gl := 0$
/* if the glasses are opened and the doors unlocked the glasses can be closed */

$Ch\_Sec\_Activation \triangleq Tr = 0 \wedge Mv = 0 \wedge Lo = 0 \wedge CS = 0 \wedge Do = 1 \Rightarrow CS := 1$
/* if the doors are opened, the children security can be activated */

$Ch\_Sec\_Deactivation \triangleq Tr = 0 \wedge Mv = 0 \wedge Lo = 0 \wedge CS = 1 \wedge Do = 1 \Rightarrow CS := 0$
/* if the doors are opened, the children security can be deactivated */

$Car\_Moving \triangleq Tr = 0 \wedge Mv = 0 \wedge Do = 0 \wedge Lo = 0 \wedge AC = 0 \Rightarrow Mv := 1$
/* the car can be moved when it is unlocked and doors closed */

$Car\_Stopping \triangleq Tr = 0 \wedge Mv = 1 \wedge Do = 0 \wedge Lo = 0 \wedge AC = 0 \Rightarrow Mv := 0$
/* the car can be stopped when it moves */

$Light\_Activation \triangleq Tr = 0 \wedge Li = 0 \wedge Lo = 0 \wedge AC = 0 \Rightarrow Li := 1$
/* the lights can be activated when they are not and the car is unlocked */

$Light\_Deactivation \triangleq Tr = 0 \wedge Li = 1 \wedge Lo = 0 \wedge AC = 0 \Rightarrow Li := 0$
/* the lights can be deactivated when they are and the car is unlocked */

$Incr\_Chronometer \triangleq Tr = 1 \wedge De < 10 \Rightarrow De := De + 1$
/* the chronometer is incremented when it is started and when it did not reach ten ut */

Figure 12: Car Alarm System Model