# Under-Approximation Generation Driven by Relevance Predicates and Variants

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

FEMTO-ST, UMR 6174 CNRS and Univ. Bourgogne Franche-Comté
16, route de Gray F-25030 Besançon Cedex France
{jjulliand, okouchna, pamasson, gvoiron}@femto-st.fr

**Abstract** In test generation, when computing a reachable concrete under-approximation of an event system's predicate abstraction, we aim at covering each reachable abstract transition with at least one reachable concrete instance. As this is in general undecidable, an algorithm must finitely instantiate the abstract transitions for it to terminate. The approach defended in this paper is to first concretely explore the abstract graph, while concretizing the abstract transitions met at most once. However, some abstract transitions would require that loops were taken previously for them to become reached. To this end, in a second phase, a test engineer guides the exploration by describing a relevance predicate able to travel such loops. We give hints on how to design and express a relevance predicate, and provide a method for automatically extracting a variant out of it. A relevance guided concretization algorithm is given, whose termination is ensured by using this variant. Experimental results are provided that show the interest of the approach.

**Keywords:** Predicate Abstraction, Under-Approximation Generation, Loop Variant, Relevance Predicate

## 1 Introduction

In model-based testing [1,2], the user wants to derive a test suite from a model, that achieves a given coverage (e.g. all states, all transitions, etc.) of it. Sometimes the infinite or very large size of the explicit state space of the model makes its coverage impossible, and an abstraction of the model can be used instead: the possibly infinitely many explicit states are grouped into finitely many abstract super-states. In predicate abstraction [3], explicit states are mapped onto abstract ones by means of a set of predicates that characterizes each abstract state. These predicates can for example automatically derive from a formalized test intention [4]. An abstract transition links two abstract states when it has at least one explicit instantiation. Such transitions are called *may* transitions [5], meaning that they may be instantiated.

The general framework of our work is to generate tests from predicate abstractions of event systems, that are a special kind of action systems. Contrarily to programs, event systems have no explicit control flow that could be preserved

in an abstraction for guaranteeing abstract paths to be explicitly instantiable as reachable and connected sequences. To this end, [6] introduces an algorithm that widens a frontier of reached states by systematically trying to prolong the existing concrete sequences with instantiating some yet unexplored abstract transition. The approach is called concrete exploration (CXP). It aims at covering each abstract transition, but only once for avoiding the concrete state space blow-up. Experiments in [6] have shown that, despite covering most of the abstract transitions, this approach fails at covering some of them whose enabling would require that previous transitions were taken repeatedly in loops.

In this paper we propose that selected loops are allowed to be traversed by designing an adequate *relevance predicate*. It is domain specific, and relies on the knowledge owned by a test engineer of the model that (s)he has written. We revisit the relevance function of Grieskamp et al. [7], that achieves a similar goal in the context of deriving a Finite State Machine from an Abstract State Machine. Our solution is to observe the coverage achieved by CXP, in order to drive the loop executions towards a *test goal*, i.e. reaching one or more concrete states in which the non-covered abstract transitions become enabled. Our relevance predicate expresses a condition over two consecutive concrete states, telling for the target concrete state if it is relevant or not to continue the exploration from it. It has to make the exploration go through cycles. To achieve termination of this process, we propose to deduce –from a relevance predicate exhibited by the test engineer– a variant that strictly decreases until it reaches a minimal value.

Summarized, our contributions are to: (1) propose a method for designing a relevance predicate, as well as a simple language for its expression, (2) automatically deduce a loop variant from a relevance predicate expressed in this language, (3) exhibit an algorithm that implements the approach by completing, with a relevance predicate as input, an existing under-approximation, (4) experimentally assess the method. The formal background required for reading the paper is given in Sec. 2. We illustrate our approach in Sec. 3 through the example of a simple coffee vending machine. Computing of a concrete under-approximation, designing a relevance predicate and deducing a loop variant from it are explained in Sec. 4. The algorithm that implements the method is given in Sec. 5. The experimental results of applying the method to five case studies are in Sec. 6. In Sec. 7 we position our approach w.r.t. related work, and we conclude the paper in Sec. 8.

## 2 Background

In this paper, systems are specified by event systems (ES) described in the B syntax [8,9][1]. Notice however that our proposals and results are general enough since event system semantics is given by labelled transition systems (LTS).

This section first provides the syntax and the semantics of B event systems. Then we present a predicate abstraction and formalize it for event systems by

[1] Our experimental models are written in B, but could alternatively be translated into a syntax with guarded commands [10], such as Abstract State Machines [11,12].

means of May Transition Systems (MTS). Finally, we recall the notion of variant, usually used to prove the termination of iterative programs and systems. It will serve as the support to terminate the exploration from a relevance predicate.

### 2.1 Model Syntax and Semantics

Let us first introduce B event systems. They are composed of events specified by means of guarded actions [10]. Once the system is in a state satisfying the guard of an event, the latter is spontaneously fired.

**Definition 1 (Event System).** *Let* EvName *be a set of event names. A B event system is a tuple* $\langle X, I, Init, \mathrm{Ev}\rangle$*, where* $X$ *is a set of state variables,* $I$ *is a state invariant,* $Init$ *is an* initialization *action such that* $I$ *holds in any initial state, and* Ev *is a set of event definitions, each of the form* $e \stackrel{def}{=} a$ *where* $e \in \mathrm{EvName}$ *is the name of the event and* $a$ *the action it performs. Note that every application of an event must preserve* $I$*.*

**Definition 2 ((Concrete) State of an Event System).** *A (concrete) state of an event system* $\langle X, I, Init, \mathrm{Ev}\rangle$ *is a predicate preserving* $I$ *and defined as a conjunction of valuations of all state variables in* $X$*.*

The events are defined by composing the following five primitive actions: *skip*, an action with no effect, $x := E$ an action assigning the value of the arithmetic expression $E$ to the state variable $x$, $P \Rightarrow a$, a guarded action requiring the event system to be in a state satisfying the predicate $P$ before the action $a$ can be applied, $a_1 [] a_2$, a bounded non-deterministic choice between the two actions $a_1$ and $a_2$ and finally $@z.a$ an action applying the action $a$ which depends on the bound variable $z$ whose value is chosen non-deterministically. The guard (noted $\mathsf{grd}$) is defined on the primitive actions by: $\mathsf{grd}(skip) \stackrel{\mathrm{def}}{=} true$ (the *skip* action can always be applied), $\mathsf{grd}(x := E) \stackrel{\mathrm{def}}{=} true$ (the single assignment action can always be applied), $\mathsf{grd}(P \Rightarrow a) \stackrel{\mathrm{def}}{=} P \wedge \mathsf{grd}(a)$ (as defined before, the guarded action only applies $a$ if the system is in a state satisfying $P$, and the guard of $a$ ($\mathsf{grd}(a)$) must also be satisfied for $a$ to be applied), $\mathsf{grd}(a_1 [] a_2) \stackrel{\mathrm{def}}{=} \mathsf{grd}(a_1) \vee \mathsf{grd}(a_2)$ (one of the actions $a_1$ or $a_2$ whose guard is satisfied is applied), $\mathsf{grd}(@z.a) \stackrel{\mathrm{def}}{=} \exists(z).\mathsf{grd}(a)$ (there must exist some bound variable $z$ satisfying the guard of $a$ (which depends on $z$) for $a$ to be applied).

See Fig. 1 in Sec. 3 for an example of a B event system. Following [13], we define the semantics of event systems by means of a labelled transition system (LTS). Let $e \stackrel{\mathrm{def}}{=} a$ be an event. It has a *weakest precondition* [14] w.r.t. a set $Q'$ of target states, denoted $wp(a, Q')$. It is the largest set of states from which applying $a$ always leads to a state in $Q'$. An event also defines a relation between the values of the state variables before ($X$) and after ($X'$) the application of the event. It is expressed by the before-after predicate of the event $e \stackrel{\mathrm{def}}{=} a$, denoted $prd_X(a)$.

Let us now formally define $wp$ and $prd_X$ following [8]. We associate the sets of states $Q$ and $Q'$ with predicates: a set of states $Q$ defines a predicate $Q$ that holds in any state of $Q$, but does not hold in any state not in $Q$.

We define the $wp$ w.r.t. the five primitive actions by: $wp(skip, Q') \stackrel{\text{def}}{=} Q'$, $wp(x := E, Q') \stackrel{\text{def}}{=} Q'[E/x]$ that is the usual substitution of $x$ by $E$, $wp(P \Rightarrow a, Q') \stackrel{\text{def}}{=} P \Rightarrow wp(a, Q')$, $wp(a_1 [] a_2, Q') \stackrel{\text{def}}{=} wp(a_1, Q') \vee wp(a_2, Q')$, $wp(@z.a, Q') \stackrel{\text{def}}{=} \forall z.wp(a, Q')$, where $z$ is not a free variable in $Q'$.

Then $prd_X$ is defined w.r.t. $wp$ by $prd_X(a) \stackrel{\text{def}}{=} \neg wp(a, x'_1 \neq x_1 \vee \ldots \vee x'_n \neq x_n)$. It is a predicate over the state variables $X = \{x_1, \ldots, x_n\}$ in the source state before $a$, and the target state variables $X' = \{x'_1, \ldots, x'_n\}$ after $a$.

### 2.2 Predicate Abstraction

Predicate abstraction [3] is a special instance of the framework of abstract interpretation [15] that maps the potentially infinite state space $C$ of an LTS onto the finite state space $\mathsf{A}$ of an abstract transition system *via* a set $\mathcal{P} \stackrel{\text{def}}{=} \{p_1, p_2, \ldots, p_n\}$ of $n$ predicates over the state variables. The set of abstract states $\mathsf{A}$ 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 $q$ is also considered as the predicate $\bigwedge_{i=1}^{n} q_i$. We define a total abstraction function $\alpha : C \rightarrow \mathsf{A}$ 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 that $c$ is a concrete state of $q$.

Let us now define abstract *may* transitions. Consider two abstract states $q$ and $q'$, and an event $e$. There exists a *may* transition $q \xrightarrow{e} q'$, if and only if there exists at least one concrete transition $c \xrightarrow{e} c'$ such that $\alpha(c) = q$ and $\alpha(c') = q'$. The *may* transition is *reachable* if and only if there is at least one such concrete transition $c \xrightarrow{e} c'$ whose source state $c$ is reachable from a concrete initial state.

We check predicate satisfiability thanks to SMT solvers. For a predicate $P$, we define the solver invocation $SAT_c(P)$ as returning either a model of $P$, or `unsat` if $P$ is unsatisfiable, or `unknown` if the solver failed to determine the satisfiability of $P$. We also define $SAT(P)$ as the predicate that is true iff $SAT_c(P)$ returns a model. Let $e \stackrel{\text{def}}{=} a$ be an event definition, $q \xrightarrow{e} q'$ is a *may* transition iff $SAT(\neg wp(a, \neg q') \wedge q)$. We compute a concrete witness $c \xrightarrow{e} c'$ by using the before-after predicate: $(c, c') := SAT_c(prd_X(a) \wedge q'[X'/X] \wedge q)$ where $q'[X'/X]$ is the $q'$ predicate in which each state variable $x_i$ is substituted by $x'_i$.

### 2.3 May Transition Systems

Definition 3 introduces *may* transition systems (MTS) having abstract states, and abstract *may* transitions. Definition 4 associates an abstraction defined by an MTS with an ES. The reader will be provided with an example in Sec. 3.

**Definition 3 (May Transition System).** *Let* EvName *be a finite set of event names, and* $\mathcal{P} \stackrel{\text{def}}{=} \{p_1, p_2, \ldots, p_n\}$ *be a set of predicates. Let* $\mathsf{A}$ *be a set of* $2^n$ *abstract states defined from* $\mathcal{P}$. *A tuple* $\langle Q, Q_0, \Delta \rangle$ *is an MTS 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, and* $\Delta(\subseteq Q \times \text{EvName} \times Q)$ *is a may transition relation with labels in* EvName.

**Definition 4 (MTS from an ES and abstraction predicates).** *Let* $ES \stackrel{def}{=} \langle X, I, Init, \text{Ev}\rangle$ *be an ES, and* $\mathcal{P} \stackrel{def}{=} \{p_1, p_2, ..., p_n\}$ *be a set of* $n$ *predicates over* $X$ *defining a set of* $2^n$ *abstract states. A tuple* $\langle Q, Q_0, \Delta\rangle$ *is an MTS from ES and* $\mathcal{P}$ *if it satisfies the following conditions:*

- $Q \stackrel{def}{=} \{q \in \mathsf{A} | \exists (q', e).(q \stackrel{e}{\rightarrow} q' \in \Delta \vee q' \stackrel{e}{\rightarrow} q \in \Delta)\}$,
- $Q_0 \stackrel{def}{=} \{q | q \in \mathsf{A} \wedge (SAT(prd_X(Init) \wedge q[X'/X]))[X/X']\}$,
- $\Delta \stackrel{def}{=} \{q \stackrel{e}{\rightarrow} q' | q \in \mathsf{A} \wedge q' \in \mathsf{A} \wedge e \stackrel{def}{=} a \in \text{Ev} \wedge SAT(\neg wp(a, \neg q') \wedge q)\}$.

*Reachable MTS.* The reachable MTS from ES and $\mathcal{P}$ is the MTS that contains all the reachable *may* transitions, and only those ones. The notion of reachable MTS is the same as that of *true* FSM in the context of abstract state machines [7]. As such, and as proved in [7], computing the reachable MTS from an ES and a set $\mathcal{P}$ of abstraction predicates is in general an undecidable problem.

### 2.4 Variant of Iterative Systems

The notion of variant is usually used to prove the termination of iterative programs and systems. In this paper variants are associated with relevance predicates, which can be seen as a kind of test goal.

For example in the deductive verification tools Frama-C [16] and KeY [17], in order to prove the termination of program loops, the engineer must provide for each loop a variant annotation that defines a natural integer arithmetic expression. This expression must be non-negative before each iteration of the loop, and must strictly decrease at each iteration. For example, in a binary search algorithm in the array interval $L..R$, the variant is the expression $R - L$ that defines the length of the search interval. It has to strictly decrease at each execution of the body of the following loop: `while` $L < R$ `do` $M := (L + R + 1)/2$; `if` $T[M] \leq X$ `then` $L := M$ `else` $R := M - 1$ `fi od`. So, the algorithm terminates when the interval is such that $L = R$.

The contributions of this paper provide means, for a test engineer, to cover by tests the transitions whose enabling require that a loop of events have previously been executed. We mainly propose that the test engineer provides a test goal described by means of a relevance predicate, from which we deduce a variant such that any event satisfying the relevance predicate strictly decreases this variant.

## 3 Running Example

Our illustrative example is a simplified coffee vending machine (see the ES in Fig. 1). It has a *Balance*, which can be augmented by putting coins of value either 50 or 100 (events `insert50` and `insert100` in Fig. 1). *Balance* may not

exceed an arbitrary fixed constant named *MAX_Bal*. There are arbitrary constants for the maximal number of coffees stored in the machine (*MAX_Cof*), and the maximal value (*MAX_Pot*+50) of the *Pot* (the money kept by the machine). Notice that *Balance* and *Pot* are multiples of 50 (specified in the invariant). 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 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 2 represents the MTS of the ES of Fig. 1 for the three following predicates: $p_0 \stackrel{\text{def}}{=} Status = 0 \wedge Pot \geq MAX_Pot - 50$, $p_1 \stackrel{\text{def}}{=} Status = 1$ and $p_2 \stackrel{\text{def}}{=} (Status = 1 \wedge AskChange = 0 \wedge AskCof = 0 \wedge Balance = 0) \vee Status = 2$ that are respectively the guards of the events takePot, autoOut and powerDown.

The test generation method presented in [6] generates a concrete LTS that is an under-approximation of the semantics of the specification in Fig. 1. The method concretizes all the *may* transitions but some instances are not connected to the initial state of the under-approximation, due to the choice of traversing each *may* transition only once. Sometimes previous transitions should have been taken in loop for reaching a connected concrete state in which the targeted transition is enabled. This is for example the case with the transition $q_2 \xrightarrow{\text{serveCof}} q_1$. It serves the machine's last coffee in stock. Its enabling requires to previously execute a loop which serves all coffees until emptying the stock. The idea presented in this paper for covering such transitions is to trigger a second step, for completing a posteriori the LTS computed at the first step by the algorithm of [6]. This second step is allowed to loop through some cycles of the abstract graph, by generating new concrete states from the existing ones as long as they are *relevant*. For guaranteeing this looping to terminate, we propose a state to be relevant as long as it decreases a variant of the loop.

*Relevance Predicate Example*: In the coffee machine, transition $q_2 \xrightarrow{\text{serveCof}} q_1$ can only be triggered once the coffee stock is empty ($CofLeft = 0$). This requires having previously looped between states $q_3$ and $q_2$ through the events insert50, insert100, cofReq and serveCof. The progress conditions along that loop are that either the variable *Balance* increases, or the variable *CofLeft* decreases, or the variable *AskCof* passes from zero to one. In terms of the before and after values of the variables, this is expressed as the following *relevance predicate* (RP): $Balance' > Balance \vee CofLeft' < CofLeft \vee (AskCof = 0 \wedge AskCof' = 1)$.

$$
\begin{array}{lll}
\mathsf{X} & \stackrel{\text{def}}{=} & \{Balance, Pot, Status, CofLeft, AskCof, AskChange\} \\
\mathsf{I} & \stackrel{\text{def}}{=} & Pot \in 0..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} & \stackrel{\text{def}}{=} & Balance := 0 \mid\mid Status := 0 \mid\mid Pot := 0 \mid\mid \\
 & & CofLeft := 10 \mid\mid AskCof := 0 \mid\mid AskChange := 0 \\
\mathsf{insert50} & \stackrel{\text{def}}{=} & Status = 1 \wedge AskChange = 0 \wedge AskCof = 0 \wedge \\
 & & Balance + 50 \leq MAX_Bal \Rightarrow Balance := Balance + 50 \\
\mathsf{insert100} & \stackrel{\text{def}}{=} & Status = 1 \wedge AskChange = 0 \wedge AskCof = 0 \wedge \\
 & & Balance + 100 \leq MAX_Bal \Rightarrow Balance := Balance + 100 \\
\mathsf{powerUp} & \stackrel{\text{def}}{=} & Status = 0 \wedge CofLeft > 0 \wedge Pot \leq MAX_Pot \Rightarrow \\
 & & Status := 1 \mid\mid Balance := 0 \mid\mid AskCof := 0 \mid\mid AskChange := 0 \\
\mathsf{powerDown} & \stackrel{\text{def}}{=} & (Status = 1 \wedge AskChange = 0 \wedge AskCof = 0 \wedge Balance = 0) \\
 & & \;\vee Status = 2 \Rightarrow Status := 0 \\
\mathsf{autoOut} & \stackrel{\text{def}}{=} & Status = 1 \Rightarrow Status := 2 \\
\mathsf{takePot} & \stackrel{\text{def}}{=} & Status = 0 \wedge Pot \geq MAX_Pot - 50 \Rightarrow Pot := 0 \\
\mathsf{cofReq} & \stackrel{\text{def}}{=} & Status = 1 \wedge Balance \geq 50 \wedge AskCof = 0 \wedge \\
 & & AskChange = 0 \Rightarrow AskCof := 1 \\
\mathsf{changeReq} & \stackrel{\text{def}}{=} & Status = 1 \wedge Balance > 0 \wedge AskCof = 0 \wedge \\
 & & AskChange = 0 \Rightarrow AskChange := 1 \\
\mathsf{addCof} & \stackrel{\text{def}}{=} & \exists x.(x \in 1..MAX_Cof \wedge CofLeft + x \leq MAX_Cof \\
 & & \wedge Status = 0 \Rightarrow CofLeft := CofLeft + x) \\
\mathsf{serveCof} & \stackrel{\text{def}}{=} & Status = 1 \wedge Balance \geq 50 \wedge AskCof = 1 \wedge CofLeft > 0 \wedge \\
 & & Pot \leq MAX_Pot \Rightarrow \\
 & & \quad AskCof := 0 \mid\mid Balance := Balance - 50 \\
 & & \;\mid\mid CofLeft := CofLeft - 1 \mid\mid Pot := Pot + 50 \\
 & & \;\mid\mid (Pot \geq MAX_Pot \vee CofLeft = 1 \Rightarrow Status := 2 \\
 & & \quad [] \; Pot + 50 \leq MAX_Pot \wedge CofLeft \neq 1 \Rightarrow skip) \\
 & & \;\mid\mid (Balance > 50 \Rightarrow AskChange := 1 \; [] \; Balance = 50 \Rightarrow skip) \\
\mathsf{backBalance} & \stackrel{\text{def}}{=} & Status = 1 \wedge Balance > 0 \wedge AskChange = 1 \Rightarrow \\
 & & Balance := 0 \mid\mid AskChange := 0
\end{array}
$$

**Figure 1.** ES Specification of a Coffee Machine

## 4 Test Generation Based On Relevance Predicates

We first define the concept of Approximated Transition System (ATS) which brings together an MTS and one of its under-approximations. Section 4.1 gives an overview of the process in two phases of under-approximation that we propose for computing an ATS. Then Sec. 4.2 explains how to design an RP on which the second phase (detailed in Sec. 5) is based. Finally Sec. 4.3 gives the relationship between an RP and a variant that guarantees the termination of the method.

We call Approximated Transition System (ATS, see Def. 5) the reunion of an abstraction with one of its under-approximations that is a concrete part of the LTS, which is the semantics of the event system from which the MTS is deduced.

**Definition 5 (Approximated Transition System).** *Let $\langle Q, Q_0, \Delta \rangle$ be an MTS. A tuple $\langle Q, Q_0, \Delta, C, C_0, \Delta^c, \alpha \rangle$ is an ATS whose $\langle C, C_0, \Delta^c, \alpha \rangle$ is a concretization of the MTS where $C, C_0$ are sets of respectively concrete states and concrete initial states, $\Delta^c (\subseteq C \times \text{EvName} \times C)$ is a concrete labelled transition relation, and $\alpha$ is a total abstraction function from $C$ to $Q$.*

### 4.1 Process Overview

We sketch our process in Fig. 3. We propose to compute an ATS in two steps. We get a first version (ATS (1) in Fig. 3) by an approach [6] called CXP for

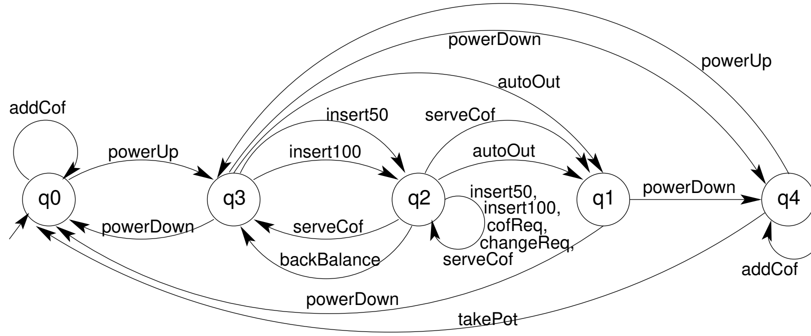


**Figure 2.** MTS of the Coffee Machine w.r.t. Predicates $p_0$, $p_1$ and $p_2$.

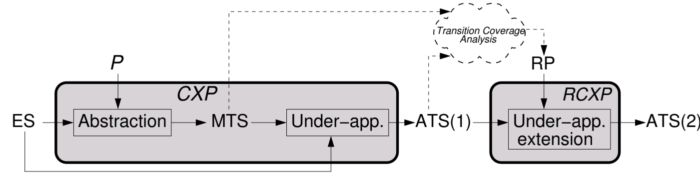


**Figure 3.** ATS Computation Process

*concrete exploration*, that traverses and concretizes each abstract transition only once. Then, thanks to a relevance predicate RP provided by the test engineer, selected loops of abstract transitions are additionally traversed and concretized by RCXP, in order to connect to new concrete transitions. As a result, ATS (1) is extended to ATS (2).

The CXP approach is fully described in [6]. The two operations **Abstraction** and **Under-app.** are summarized as follows.

1. **Abstraction.** The test engineer designs a set of abstraction predicates $P$ related to the behaviour of the system ES (s)he wishes to observe. It is proposed in [4] that these predicates are extracted from a *test purpose*, which is a test intention formalized by a pattern *a la* Dwyer et al. [18]. Using algorithm in [6] provides the test engineer with an MTS that over-approximates the reachable MTS.
2. **Under-app.** The under-approximation is computed by concretizing on the fly each *may* transition once, as it is discovered. The principle is to compute an instance that prolongs, whenever possible, some existing sequence connected to a concrete initial state. For CXP's efficiency, each abstract transition is concretized only once and thus cannot be applied repeatedly. The ATS obtained is called ATS (1) in Fig. 3.

In general, not all the instances of abstract transitions built by ATS (1) are reached, even though they are possibly reachable. It is always the case in particular, when repeating some transitions in a cycle would have been necessary

for enabling another transition. In case ATS (1) fails at building a connected instance of a *may* transition, the transition is concretized anyway but as a "hanging" instance, i.e. disconnected from the previously reached part of the under-approximation.

The second step, called RCXP for *relevant concrete exploration*, requires human interaction. The test engineer analyses, for each abstract transition of the MTS unreached in ATS (1), if (s)he thinks it could have been reached. If so (s)he identifies which transitions taken in loop it would require for reaching it. For that (s)he can observe in the MTS which cycles lead to enabling the target transitions. As this looping may not terminate, (s)he has to provide an RP telling whether or not it is relevant to pursue in the loop. The operation **Under-app. extension** consists of adding to ATS (1) the concrete transitions obtained by this RP guided exploration. It results in ATS (2), in which the transitions originally targeted by RP are possibly reached. An algorithmic implementation of RCXP is given in Sec. 5. Let us for now illustrate the process and RP design through the coffee machine example.

### 4.2 Design of a Relevance Predicate and Illustration of the Method

For illustrating the application of the method, we consider the coffee machine example, and a requirement stating that it must not break down after being powered off, so that collecting the pot remains possible. Using the temporal logic patterns of Dwyer et al. [18], this can be expressed as: *Never* autoOut *Between* powerDown *and* takePot. As proposed in [4], the tester can use the guards of the events invoked in this test purpose as abstraction predicates for computing the MTS. Here, this gives the predicates $p_0$, $p_1$ and $p_2$ defined in Sec. 3, from which the abstraction of Fig. 2 has been computed. The tester executes CXP [6] with these predicates as input, and observes the resulting MTS coverage. In the case of the coffee machine, the two transitions $q_2 \xrightarrow{\mathsf{serveCof}} q_1$ and $q_1 \xrightarrow{\mathsf{powerDown}} q_4$ and the state $q_4$ are not covered. Having designed the model, the tester is able to understand that the transition $q_2 \xrightarrow{\mathsf{serveCof}} q_1$ serves the last coffee in stock, so that its coverage would have required that previously all the coffees were served. By looking at the MTS, it is easy to see that covering this transition would require looping between states $q_3$ and $q_2$. (S)He identifies as illustrated in Sec. 3 the set of events to loop through in order to reach his (her) goal. In our case, the goal is serving the last coffee and the set of events to loop through is insert50, insert100, cofReq and serveCof. After this step (s)he has to express by means of a before-after predicate how it is relevant that the variables assigned in these events evolve. The RP is then the disjunction of these before-after predicates. For the coffee machine example, this gives the RP described in Sec. 3, paragraph *Relevance Predicate Example*. The variables have to decrease a variant for the looping to terminate. Let us now explain in Sec. 4.3 how to deduce this variant from the RP.

### 4.3 Variant Deduced from a Relevance Predicate

Given an RP, this section shows how to derive a variant that guarantees the termination of the computation of relevant concrete states. We assume that the test engineer uses a simple language defined as follows to express the RP, where $x$, $x'$ denote the state variable $x$ respectively before and after an event application:

$$\begin{aligned} rel_p &::= rp_1 \vee \ldots \vee rp_n \\ rp &::= ap \mid cp \\ ap &::= x' < x \mid x' > x \mid x = v \ \wedge \ x' = v' \\ cp &::= b_1 \Rightarrow ap_1 \wedge \ldots \wedge b_m \Rightarrow ap_m \end{aligned}$$

An RP is a disjunction of before-after predicates, each of which being either an atomic predicate $ap$ or a conditional predicate $cp$. We consider these predicates to only use the following three variable types: intervals of integer $MIN_x..MAX_x$, booleans, and finite enumerated sets of labels. We assume that the atomic predicates $ap$ over a state variable $x$ expresses that an integer variable either strictly decreases ($x' < x$) or increases ($x' > x$), or the value of an enumerated variable (including the boolean type) passes from $v$ to $v'$. We assume that $v \neq v'$. We also consider that in the conditional predicate pattern $cp$, each $b_i$ is a boolean condition on the source state. For any concrete state $c$, we finally assume that there exists one and only one $i$ such that the predicate $b_i$ is satisfied in the concrete state $c$, denoted as: $c \models b_i$.

Let $c$, $c'$ denote respectively the state $c$ before and after an event execution. The first following three rules associate an initial variant, denoted $V_{init}(rp, c)$, with the concrete state $c$ depending on the RP $rp$. The next five rules associate the next value of the variant, denoted $V(rp, c')$, with an RP $rp$ in a target state $c'$ reached from a source state $c$. Notice that in rules 4, 5, 6 and 8, $V(rp, c) \stackrel{\text{def}}{=} V_{init}(rp, c)$ when $c$ is an initial state:

1. $V_{init}(ap, c) \stackrel{\text{def}}{=} Card(Type(x))$,
2. $V_{init}(b_1 \Rightarrow ap_1 \wedge \ldots \wedge b_m \Rightarrow ap_m, c) \stackrel{\text{def}}{=}$
   `if` $c \models b_1$ `then` $V_{init}(ap_1, c)$
   `else if` $\ldots$ `else if` $c \models b_m$ `then` $V_{init}(ap_m, c)$,
3. $V_{init}(rp_1 \vee \ldots \vee rp_n, c) \stackrel{\text{def}}{=} V_{init}(rp_1, c) + \ldots + V_{init}(rp_n, c)$,
4. $V(x' < x, c') \stackrel{\text{def}}{=} V(x' < x, c) - (x - x')$,
5. $V(x' > x, c') \stackrel{\text{def}}{=} V(x' > x, c) - (x' - x)$,
6. $V(x = v \wedge x' = v', c') \stackrel{\text{def}}{=} V(x = v \wedge x' = v', c) - 1$,
7. $V(b_1 \Rightarrow ap_1 \wedge \ldots b_m \Rightarrow ap_m, c') \stackrel{\text{def}}{=}$
   `if` $c' \models b_1$ `then` $V(ap_1, c')$
   `else if` $\ldots$ `else if` $c' \models b_m$ `then` $V(ap_m, c')$.
8. $V(rp_1 \vee \ldots \vee rp_n, c') \stackrel{\text{def}}{=} \sum_{\{i \mid i \in 1..n \wedge (c,c') \models \neg rp_i\}} V(rp_i, c) + \sum_{\{i \mid i \in 1..n \wedge (c,c') \models rp_i\}} V(rp_i, c')$.

Rule 1 allows applying as many operations as the size of the enumerated sets or of the integer intervals ($MAX_x - MIN_x + 1$). Rule 2 applies the previous rule according to the condition that holds in the initial state. Rule 3 defines the initial variant value as the sum of the variant values of each of the disjunction members. Rules 4 and 5 define that the next variant value decreases of the difference between the two successive values of $x$. Rule 6 defines that the next variant value for a modification of an enumerated variable decreases of one. Rule 7 defines that the next variant value for a conditional predicate decreases as much as the atomic predicate that is satisfied in the state $c'$. Last, with rule 8, the variant in the target state of a relevant predicate is unchanged for the disjunction member that are not satisfied, and varies according to the rules that apply for the ones that are satisfied.

*Property 1.* If an RP is satisfied, then the associated variant decreases.

*Proof.* For a transition $c \xrightarrow{\mathsf{e}} c'$ in an ATS and for an RP $rp$, the variant decreases if $V(rp, c') < V(rp, c)$. We prove that for the three predicate cases: $ap, cp$ and $rel_p$. For an atomic predicate, the variant decreases respectively of $|x - x'|$ and one respectively according to rules 4, 5 and 6. For a conditional predicate, the variant decreases as much as the atomic predicate that is true according to rule 7. For a disjunctive predicate $rel_p$ the variant decreases, according to rule 8. Indeed, (1) the variant is not modified for the disjunction members that are not satisfied in the consecutive states $c, c'$, and (2) the variant decreases for the disjunction members that are satisfied in the states $c, c'$ because they are either $ap$ or $cp$, for which the decrease has already been shown.

## 5 RCXP Algorithm

In this section we present the main contribution of this paper. The RCXP (for relevant concrete exploration) algorithm implements the second step of the process presented in Sec. 4.1.

### 5.1 Under-Approximation Extension Using Relevance Predicates

To extend the ATS computed in the first place by CXP, we propose an algorithm called RCXP which aims at covering the non-covered transitions. It is driven by an RP designed as explained in Sec. 4.2. RCXP is designed from the concepts of relevant state and goal state. A goal state is a state in which an non-covered abstract transition is triggerable. Informally a state is relevant when it gets closer to a goal state. Formally we say that a target state $c'$ by a transition $t$ is relevant w.r.t. the source state $c$ of $t$ when $(c, c')$ satisfies an RP (see Sec. 4.3).

Algorithm RCXP launches its execution from each state $c$ built by CXP that is evaluated as relevant, assuming that the variant expression $V_{init}(rp, c)$ is non-negative (line 1). The algorithm tries to reach a new relevant target concrete state (line 10) for each target abstract state (line 7) and for each event (line 8) such that the corresponding transition is *may* (line 9). If such a state $c'$ is found

**Algorithm RCXP:** Concretization Algorithm using Relevance Predicate

**Inputs** : $\langle Q, Q_0, \Delta, C, C_0, \alpha, \Delta^c\rangle$: an ATS; $\mathsf{A}$: the set of all abstract states $relevance_pred_X$: a relevance predicate
**Output** : $\langle Q, Q_0, \Delta, C, C_0, \alpha, \Delta^c\rangle$: the ATS enriched
**Variables** : $RCS(resp.\ PRCS)$: the set of relevant concrete states to process (resp. processed)

```
1  RCS := {c | c ∈ C ∧ V_init(relevance_pred_X, c) ≥ 0}; PRCS := ∅;
2  /* the reachable concrete states computed by CXP */
3  while RCS ≠ ∅ do
4    choose c ∈ RCS;
5    RCS := RCS − {c}; PRCS := PRCS ∪ {c};
6    q := α(c);
7    foreach q' ∈ A do
8      foreach e =def a ∈ Ev do
9        if q -e-> q' ∈ Δ then
10         (c, c') := SAT_c(c ∧ prd_X(a) ∧ q'[X'/X] ∧ relevant_pred_X);
11         if (c, c') ∉ {unknown, unsat} then
12           if V(relevance_pred_X, c') ≥ 0 ∧ c' ∉ PRCS then
13             RCS := RCS ∪ {c'};
14           end
15           α(c') := q'; C := C ∪ {c'}; Δ^c := Δ^c ∪ {c -e-> c'};
16         else
17           /* a goal state is reached, we try to apply the transition from it */
18           (c, c') := SAT_c(c ∧ prd_X(a) ∧ q'[X'/X]);
19           if (c, c') ∉ {unknown, unsat} then
20             α(c') := q'; C := C ∪ {c'}; Δ^c := Δ^c ∪ {c -e-> c'};
21           end
22         end
23       end
24     end
25   end
26 end
```

(lines 12-15), it is added (line 15) to the under-approximation. Additionally in case $c'$ is new and has a non-negative variant value (line 12), it is added to the set of relevant states to be processed (line 13). When no more relevant state is found (*else* statement in line 16), a goal state has been reached. The algorithm tries to finally apply the event $e$ from it (lines 18-21) because it might correspond to a non-covered transition. The algorithm's result is the input ATS enriched with new concrete states and transitions.

### 5.2 Soundness, Complexity and Termination

This section discusses the soundness, complexity and gives the termination proof for the RCXP algorithm.

*Soundness.* RCXP computes an under-approximation of the reachable MTS. Indeed, as our method only keeps transition instances that are connected to an initial concrete state, all the *may* transitions that we cover are reachable, and thus are part of the reachable MTS.

*Complexity.* Let us denote by $C_{in}$ the set $C$ of concrete states of the input ATS. For each state $c \in C_{in}$, RCXP computes a concrete instance of each *may* transition whose source state is $\alpha(c)$ (there are at most $|Ev| \times |\mathsf{A}|$ of them). From every state reached from these concrete transitions (at most $|C_{in}| \times |Ev| \times |\mathsf{A}|$ states), RCXP launches a search for relevant successors. The number of computed successors is bounded by the maximum number of steps allowed by the variant, which equals $\max_{c \in C_{in}}(V_{init}(rp, c))$. Thus, our algorithm runs in $O(|C_{in}| \times |Ev| \times |\mathsf{A}| \times \max_{c \in C_{in}}(V_{init}(rp, c)))$. Notice that $|C_{in}|$, $|Ev|$ and $|\mathsf{A}|$ depend on the size of the abstract graph and that in practice the number of "relevant events" is likely to be lower than $|Ev|$. This means that for an abstract graph of reasonable size, the complexity is dominated by the number of steps of the variant.

*Termination.* Our algorithm computes new concrete states only from concrete states for which the variant on the one hand is non-negative, and on the other hand strictly decreases (see property 1). Thus our algorithm terminates. In addition, as the number of relevant states may explode, RCXP has been implemented with a timeout option modifiable by the tester.

## 6 Implementation and Experiments

The tool used to generate the results presented in this section, as well as the complete set of examples, along with their corresponding sets of abstraction and relevance predicates, can be downloaded, compiled and used by following the instructions at https://github.com/stratosphr/stratestx/wiki.

### 6.1 Experimental Results

We have experimented with five different case studies: a multiple battery-powered electrical system (EL [19]), the coffee machine CM presented in this paper, two explorations of an automatic subway line (L14, as yet unpublished), an elevator (ELV, as yet unpublished) and a subpart of the GSM 11.11 standard (GSM [20]). The subway modelled by L14 in our experimentation has three stations and three trains that circulate in ring around them. The test goal is to observe half a revolution of a train around the ring. Two different relevance predicates have been experimented with: in L14-1 the half-lap can be that of any train, whereas it is for a fixed train in L14-2. The GSM system considered corresponds to the exploration and reading of files on a SIM (*Subscriber Identity Module*) card with different access rights. Some files can only be read when a correct PIN (*Personal Identification Number*) is entered by the user. After three unsuccessful attempts with the wrong PIN, the card is locked and can only be unlocked if the user enters the correct PUK (*PIN Unlock Key*). After ten unsuccessful attempts with the wrong PUK, the protected files can never be read again.

**Table 1.** ATS computation results

| Sys. | #Ev | #AP | $\#AS_{rchbl}$ | $\#AT_{rchbl}$ | Alg. | $\#AS_{rchd}$ | %AS | $\#AS^{rel}$ | $\%AS^{rel}$ | $\#AT_{rchd}$ | %AT | $\#AT^{rel}$ | $\%AT^{rel}$ | #CS | $\#CS_{rchd}$ | #CT | $\#CT_{rchd}$ | Time |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **EL** | 4 | 2 | 4 | 11 | CXP | 2 | 50 | - | - | 6 | 54.55 | - | - | 28 | 6 | 17 | 6 | 00:00:01 |
| | | | | | RCXP | 4 | 100 | 2 | 100 | 11 | 100 | 2 | 100 | 58 | 42 | 53 | 45 | 00:00:05 |
| | | | | | FULL | 4 | 100 | - | - | 11 | 100 | - | - | 896 | 896 | 9856 | 9856 | 00:02:10 |
| **CM** | 11 | 3 | 5 | 21 | CXP | 4 | 80 | - | - | 12 | 57.14 | - | - | 46 | 11 | 33 | 12 | 00:00:01 |
| | | | | | RCXP | 5 | 100 | 1 | 100 | 19 | 90.48 | 1 | 100 | 149 | 117 | 179 | 158 | 00:00:04 |
| | | | | | FULL | 5 | 100 | - | - | 21 | 100 | - | - | 1742 | 1742 | 4503 | 4503 | 00:00:49 |
| **L14-1** | 15 | 3 | 4 | 49 | CXP | 2 | 50 | - | - | 4 | 8.16 | - | - | 99 | 5 | 57 | 4 | 00:00:04 |
| | | | | | RCXP | 4 | 100 | 2 | 100 | 49 | 100 | 45 | 100 | 2897 | 2840 | 7533 | 7498 | 00:59:08 |
| | | | | | FULL | 4 | 100 | - | - | 49 | 100 | - | - | 2982 | 2982 | 7950 | 7950 | 00:14:39 |
| **L14-2** | 15 | 3 | 4 | 49 | CXP | 2 | 50 | - | - | 4 | 8.16 | - | - | 103 | 5 | 57 | 4 | 00:00:05 |
| | | | | | RCXP | 4 | 100 | 2 | 100 | 11 | 22.45 | 5 | 100 | 136 | 40 | 93 | 41 | 00:00:16 |
| | | | | | FULL | 4 | 100 | - | - | 49 | 100 | - | - | 2982 | 2982 | 7950 | 7950 | 00:13:47 |
| **ELV** | 11 | 3 | 4 | 27 | CXP | 3 | 75 | - | - | 9 | 33.33 | - | - | 49 | 10 | 36 | 9 | 00:00:02 |
| | | | | | RCXP | 4 | 100 | 1 | 100 | 23 | 85.19 | 3 | 100 | 255 | 240 | 401 | 390 | 00:01:38 |
| | | | | | FULL | 4 | 100 | - | - | 27 | 100 | - | - | 1656 | 1656 | 6556 | 6556 | 00:05:27 |
| **GSM** | 5 | 2 | 4 | 25 | CXP | 1 | 25 | - | - | 5 | 20 | - | - | 45 | 5 | 30 | 5 | 00:00:01 |
| | | | | | RCXP | 4 | 100 | 3 | 100 | 16 | 64 | 8 | 100 | 155 | 115 | 208 | 183 | 00:00:16 |
| | | | | | FULL | 4 | 100 | - | - | 25 | 100 | - | - | $> 100000$ | $> 100000$ | $> 1000000$ | $> 1000000$ | $> 48:00:00$ |

The results are given in Table 1. Columns #Ev, #AP, $\#AS_{rchbl}$ and $\#AT_{rchbl}$ give respectively per model the numbers of: events, abstraction predicates, reachable abstract states and reachable abstract transitions in the MTS. Then the results per model are spread over three lines for comparing: the CXP approach (1st line), the RCXP approach (2nd line) and a full exploration of the reachable concrete space (FULL, on 3rd line). Notice that we have chosen the problem sizes in this table for making the full exploration possible.

Table 1 gives the number of: abstract states and transitions reached ($\#AS_{rchd}$ and $\#AT_{rchd}$), target abstract states and transitions of RCXP ($\#AS^{rel}$ and $\#AT^{rel}$) and concrete states and transitions either built (#CS and #CT) or reached from an initial state ($\#CS_{rchd}$ and $\#CT_{rchd}$). The other columns indicate the percentage of abstract states and transitions reached ($\%AS = \frac{\#AS_{rchd}}{\#AS}$ and $\%AT = \frac{\#AT_{rchd}}{\#AT}$) and target abstract states and transitions of RCXP reached ($\%AS^{rel} = \frac{\#AS^{rel}_{rchd}}{\#AS^{rel}}$ and $\%AT^{rel} = \frac{\#AT^{rel}_{rchd}}{\#AT^{rel}}$ where $\#AS^{rel}_{rchd}$ and $\#AT^{rel}_{rchd}$ are respectively the number of target abstract states and transitions of RCXP reached). The Time column gives the computation times in *hours*, *minutes* and *seconds*.

### 6.2 Results Analysis

Table 1 shows that RCXP succeeds at reaching all the RP targeted transitions in all of the five case studies (see that all the percentages equal 100 in the $\%AT^{rel}$ column). Moreover RCXP even succeeds in two out of the five case studies at reaching all the reachable abstract transitions (see the three percentages that equal 100 in the %AT column). RCXP generates far less concrete transitions than FULL. The most spectacular example is EL where RCXP only builds 53 concrete transitions among the 9836 built by FULL. The ratio between RCXP and FULL depends on RP. In EL, only one action is allowed by RP so that the number of concrete transitions (#CT) explored by RCXP is very small w.r.t that explored by FULL. By contrast in L14-1, all actions are allowed by RP, which makes #CT for RCXP and FULL very close. In L14-2, as RP restricts the actions allowed to that of a single chosen train, RCXP reduces #CT in large proportion:

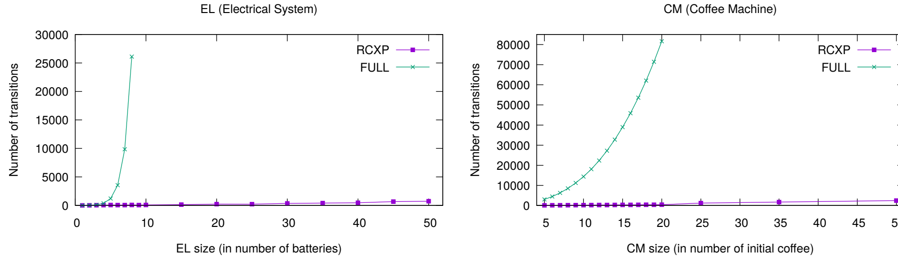


**Figure 4. #CT** growth for FULL and RCXP against the system's size

93 out of the 7950 of FULL. But only 22.45% coverage of the abstract transitions is achieved, due to the other trains' actions being unexplored. The ELV case is similar, with an RP allowing only the actions on the inside lift buttons. RCXP builds 401 concrete transitions out of the 6556 of FULL, but covers 85.19% of the abstract transitions. Note that the ELV case has necessitated several RP design attempts before covering 100% of the targeted transitions. As many smartcard like systems, the GSM allows everything to happen but returns error status words in case of unauthorized events occuring (this is called defensive programming). Therefore, its events are in practice very weakly guarded. For this reason, and because the GSM system has a lot of state variables, the FULL exploration is not feasible in reasonable time due to the huge state space (more than 100,000 states and more than a million transitions). However, the height targeted transitions (leading to states where the SIM card was definitely locked) were all successfully covered in less than 20 seconds by RCXP by only instanciating 155 states and 208 transitions. The design of the relevance predicate was also easy since it only had to decrease the number of attempts remaining for the PIN and the PUK. This shows that the method can be applied to systems of industrial size and that designing relevance predicates for such systems is not necessarily harder.

RCXP succeeds at reaching as many targeted abstract transitions as FULL with, except for L14-1, a much smaller number of concrete transitions generated, which results in smaller RCXP generation times w.r.t. FULL. Table 1 shows that in four cases (EL, CM, L14-2, GSM), the time taken by RCXP is much closer to CXP than it is to FULL. This is less spectacular with ELV, but RCXP remains faster than FULL by roughly 70%. L14-1 is the exception where RCXP lasts four times as long as FULL. This is an extreme case because in that experiment any action on any train is considered as relevant if it helps moving a train in a privileged direction. Here computing the relevant states amounts to enumerate about half of them, which is more costly with RCXP than a full enumeration with FULL, due to the RP evaluation at each step. This example shows that when RP allows too many actions to occur, the RCXP exploration gets close to the FULL one.

We have measured the concrete graph's growth w.r.t. to the problem size of our case studies. For the two of them (EL and CM) for which the FULL

exploration time took less than two hours despite its growth, we have drawn curves in Fig. 4 to compare RCXP and FULL's respective growth. As expected the curves show that FULL grows exponentially. These curves show by contrast a linear growth of RCXP. In EL the RP only allows battery fail actions, thus the sequences explored by RCXP grow linearly with the number of batteries. In CM the RP aims at serving all the coffees, thus RCXP's linear growth with the number of coffees. For the other cases, we have observed that L14-2 RCXP's exploration doesn't grow with the number of trains as only one of them is observed, but grows linearly with the number of stations, for the reason that the observed train has to move as many times as there are stations. The ELV case is similar with the number of lift moves growing linearly with the number of storeys to serve. Finally the L14-1 case showed not a linear but exponential growth of RCXP. Indeed all the trains are observed in this experiment. Thus adding one train leads to explore all of its actions, as well as their interleavings with the actions of all the other trains.

As a general conclusion of these experiments we observe that RCXP provides a means for covering the reachable part of the abstraction, and that it behaves efficiently provided that the growth of the ATS resulting of CXP is controlled, and that the number of events involved in the relevance predicate is small.

## 7 Related Work

The closest methods to ours are those proposed in [7], where a relevance function is introduced, in [21] that extends [7] and in [22] that implements the process of [7] in Spec Explorer (SE). These methods are for generating tests. They generate a Finite State Machine (FSM) that is an under-approximated concretization of an Abstract State Machine (ASM) *may* predicate abstraction. Ideally, the methods seek for building the *true* FSM of the ASM, which contains only reachable links, but all of them. SE [22] proposes five techniques to implement the defined relevance function and to prune the search space: state grouping, directed search, parameter selection, state filtering and action restriction that are closely related to our approach, though with many differences. Our method begins by computing an abstraction in which the abstract states group the concrete states defined from a set of state predicates automatically extracted from a test purpose. In SE, the tester must give a state-based grouping expression. Then our method computes a concrete under approximation by directed search (CXP). As in SE, the tester selects for that the values of many parameters, e.g. the initial number of coffees in the CM. In SE, the directed search is applied after all the pruning parameters have been given. For us, the covering of each abstract state and transition only once by CXP leads to a very strong state filtering. To relax this filtering, the tester designs a relevance predicate by observing which abstract states and transitions are not covered, in order to fix a new coverage goal. Then (s)he executes a new directed search (RCXP), that filters the states that satisfy the relevance predicate. The tester does not provide an execution's maximum length as in SE, but termination is ensured thanks to a variant automatically

computed from the relevance predicate. Last, our method allows action filtering by defining the RP, whereas in SE the tester strengthens some actions' guards.

In [23] and [24], the set of abstraction predicates is iteratively refined in order to compute a bisimulation of the model's semantics when it exists. Except by arbitrary limiting the number of refinement step (as suggested in [23]), none of these two methods is guaranteed to terminate, because the refinement step may would be repeated infinitely if no bisimulation quotient exists for the system. SYNERGY [25] and DASH [26] combine under-approximation and over-approximation for checking safety properties on programs. As we aim at proposing an efficient method for building a *reachable* under-approximation that covers all the abstract states and transitions w.r.t. a specification and a set of predicates, our algorithm does not refine the approximation but refines the under-approximation thanks to a relevance predicate associated to a variant, which guarantees the refinement process to terminate.

Some other work under-approximate an abstraction for generating tests. The tools Agatha [27], DART [28], CUTE [29], EXE [30] and PEX [31] also compute abstractions from models or programs, but only by means of symbolic executions [32]. This data abstraction approach computes an execution graph. Its set of abstract states is possibly infinite whereas it is finite with our method.

Another approach [33] for computing an under-approximation of a predicate abstraction is to characterize the abstract transitions not only as *may* ones, but also as $must+$ and $must-$. Indeed, abstract sequences in the shape of $must-^{*} \cdot may \cdot must+^{*}$ can necessarily be instantiated as connected concrete sequences. An attempt to prolong an existing under-approximation thanks to these additional modalities is experimentally tested in [34].

## 8 Conclusion And Further Work

We have proposed a method for computing (or rather completing) a concrete under-approximation of a *may* abstraction. Building a new concrete transition is conditioned by the fact that it prolongs an existing reachable concrete sequence while satisfying a user defined relevance predicate. To ensure termination this predicate has to decrease a variant, and we propose a method for automatically extracting a variant out of the relevance predicate. We have experimented with five case studies, for which we have achieved 100% coverage of the abstract transitions targeted by the relevance predicate, but with far less transitions than with the full exploration (except for one case as discussed in Sec. 6.2).

This work shows that the efficiency and success of RCXP depends on how the system is abstracted and the relevance predicate are chosen. Our results suggest that targeting a few transitions at a time with RCXP is preferable even if repeating the process is necessary. This corresponds to performing several test campaigns with different test objectives. We intend to experiment with various ways of abstracting and writing relevance predicates so as to investigate these methodological aspects. Also a more expressive relevance predicate language and a more semantic estimation of the variant's initial value are to be proposed.

## References

1. Broy, M., Jonsson, B., Katoen, J.P., Leucker, M., Pretschner, A., eds.: Model-Based Testing of Reactive Systems. Volume 3472 of LNCS. Springer (2005)
2. Utting, M., Legeard, B.: Practical Model-Based Testing. Morgan Kaufmann (2006)
3. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: CAV. Volume 1254 of LNCS., Springer (1997) 72–83
4. Bride, H., Julliand, J., Masson, P.A.: Tri-modal under-approximation for test generation. Science of Computer Programming **132**(P2) (2016) 190–208
5. Godefroid, P., Jagadeesan, R.: On the expressiveness of 3-valued models. In: VMCAI. Volume 2575 of LNCS., Springer (2003) 206–222
6. Julliand, J., Kouchnarenko, O., Masson, P.A., Voiron, G.: Approximating event system abstractions by covering their states and transitions. In: PSI'17, A.P. Ershov Informatics Conference. Volume 10742 of LNCS. (June 2017) 211–226
7. Grieskamp, W., Gurevich, Y., Schulte, W., Veanes, M.: Generating finite state machines from abstract state machines. In: ISSTA. (2002) 112–122
8. Abrial, J.R.: The B Book. Cambridge Univ. Press (1996)
9. Abrial, J.R.: Modeling in Event-B: System and Software Design. Cambridge Univ. Press (2010)
10. Dijkstra, E.: Guarded commands, nondeterminacy, and formal derivation of programs. Com. of the ACM **18**(8) (1975) 453–457
11. Gurevich, Y., Kutter, P.W., Odersky, M., Thiele, L.: Abstract State Machines, Theory and Applications. Volume 1912 of LNCS. Springer (2000)
12. Gurevich, Y.: Sequential abstract-state machines capture sequential algorithms. ACM Trans. Comput. Log. **1**(1) (2000) 77–111
13. Bert, D., Cave, F.: Construction of finite labelled transition systems from B abstract systems. In: IFM. (2000) 235–254
14. Dijkstra, E.: A Discipline of Programming. Prentice-Hall (1976)
15. Cousot, P., Cousot, R.: Abstract interpretation frameworks. J. Log. Comput. **2**(4) (1992) 511–547
16. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Asp. Comput. **27**(3) (2015) 573–609
17. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M., eds.: Deductive Software Verification - The KeY Book - From Theory to Practice. Volume 10001 of Lecture Notes in Computer Science. Springer (2016)
18. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: ICSE'99, 21st Int. Conf. on Software Engineering, Los Angeles, California, USA, ACM (1999) 411–420
19. Bué, P.C., Julliand, J., Masson, P.A.: Association of under-approximation techniques for generating tests from models. In: TAP. Volume 6706 of LNCS., Springer (2011) 51–68
20. Bernard, E., Legeard, B., Luck, X., Peureux, F.: Generation of test sequences from formal specifications: Gsm 11-11 standard case study. Software: Practice and Experience **34**(10) 915–948
21. Veanes, M., Yavorsky, R.: Combined algorithm for approximating a finite state abstraction of a large system. In: ICSE 2003/Scenarios Workshop. (2003) 86–91
22. Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., Nachmanson, L.: Model-based testing of object-oriented reactive systems with spec explorer. In: Formal Methods and Testing. (2008) 39–76

23. Namjoshi, K.S., Kurshan, R.P.: Syntactic program transformations for automatic abstraction. In: CAV. Volume 1855 of LNCS. (2000) 435–449
24. Păsăreanu, C.S., Pelánek, R., Visser, W.: Predicate abstraction with under-approximation refinement. LMCS **3**(1:5) (2007) 1–22
25. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Synergy: a new algorithm for property checking. In: SIGSOFT FSE. (2006) 117–127
26. Beckman, N.E., Nori, A.V., Rajamani, S.K., Simmons, R.J., Tetali, S., Thakur, A.V.: Proofs from tests. IEEE Trans. Software Eng. **36**(4) (2010) 495–508
27. Rapin, N., Gaston, C., Lapitre, A., Gallois, J.P.: Behavioral unfolding of formal specifications based on communicating extended automata. In: ATVA. (2003)
28. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: PLDI. (2005) 213–223
29. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: ESEC/SIGSOFT FSE. (2005) 263–272
30. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: ACM CCS. (2006) 322–335
31. Tillmann, N., de Halleux, J.: Pex-white box test generation for .net. In: TAP. Volume 4966 of LNCS. (2008) 134–153
32. Păsăreanu, C.S., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. STTT **11**(4) (2009) 339–353
33. Ball, T.: A theory of predicate-complete test coverage and generation. In: FMCO. Volume 3657 of LNCS. (2004) 1–22
34. Julliand, J., Kouchnarenko, O., Masson, P.A., Voiron, G.: Two under-approximation techniques for 3-modal abstraction coverage of event systems: Joint effort? In: TASE 2017, Nice, France (September 2017) To appear in IEEE.