Approximating Event System Abstractions by Covering their States and Transitions

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 event systems, contrarily to sequential ones, the control flow is implicit. Consequently, their abstraction may give rise to disconnected and unreachable paths. This paper presents an algorithmic method for computing a reachable and connected under-approximation of the abstraction of a system specified as an event system. We compute the under-approximation with concrete instances of the abstract transitions, that cover all the states and transitions of the predicate-based abstraction. To be of interest, these concrete transitions have to be reachable and connected to each other. We propose an algorithmic method that instantiates each of the abstract transitions, with heuristics to favour their connectivity. The idea is to prolong whenever possible the already reached sequences of concrete transitions, and to parameterize the order in which the states and actions occur. The paper also reports on an implementation, which permits to provide experimental results confirming the interest of the approach with related heuristics.

Keywords: Predicate abstraction; under-approximation; event systems

1 Introduction

Abstracting a program or its specification allows to control the size of its state space description, at the price of a loss in accuracy. That facilitates their algorithmic exploitation, otherwise limited by the huge if not infinite number of concrete states. The general idea of abstraction is to gather states that share common properties into super-states. In predicate abstraction [1] the concrete states are mapped onto a finite set of abstract ones, by means of a set of predicates that characterizes each abstract state. An abstract transition links two abstract states when it has at least one concrete instantiation. Such transitions are called *may* [2], meaning that they may be instantiated. Still there is no guarantee that two consecutive *may* transitions can necessarily be instantiated as two consecutive *connected* concrete transitions: their respective target and source concrete states may differ.

This paper aims at computing connected and reachable concrete paths from a predicate abstraction of a system formally specified as an event system [3], which is a special kind of action system [4,5]. We propose an algorithmic method for computing an under-approximation that covers all the states and transitions of this abstraction.

An event in an event system specifies state variable modifications by means of a guarded action. The actions are activated whenever their guard becomes true, so that there is no natural control flow as in a program. As a result, paths of the system may become disconnected and even unreachable in the abstraction. Still, we are interested in covering the reachable part of the abstraction as best as possible. This work has been motivated by a testing purpose: our aim is to cover by tests some selected execution paths of a system, and abstracting it avoids its state space to blow-up. But the method could also apply for example to the model-checking of safety properties.

We propose to under-approximate the abstraction by computing concrete instances of the abstract event sequences. The idea behind our method is to favour the connectivity and reachability of the successive concrete instances by prolonging whenever possible the already reached concrete transitions. Our proposal in this paper is as sketched:

– we instantiate each of the abstract transitions by enumerating all the possibilities of connecting two abstract states by any event,
– we use heuristics for controlling the order in which the events and states are enumerated, according to some know-how of the natural flow of the events succession,
– we use concrete state coloration, similarly to [6], for prolonging preferably the sequences known to be reachable and connected.

Our contributions allow then generating a concrete transition system from an event system. We also report on an implementation, which permits us to provide experimental results confirming the interest of the approach with the related heuristics.

The technical background of our paper regarding event systems, predicate abstraction and may transition systems is given in Sect. 2. Section 3 presents an electrical system as an illustrative example. The algorithm for computing both an abstraction and its approximation is presented in Sect. 4. The heuristics that we propose to enhance the coverage achieved by the algorithm are given in Sect. 5. Our experimental results are presented in Sect. 6. Section 7 describes related work, and Sect. 8 concludes the paper.

2 Background

In this paper systems are specified by event systems (ES) described in the B syntax [7,3][1]. Notice however that our proposals and results are generic since event system semantics is defined by concrete labelled transition systems.

In this section we first present the syntax and the semantics of the B event systems. Then we present the concept of predicate abstraction and formalize the abstraction of event systems by means of May Transition Systems (MTS).

[1] Our experimental models were written in B, but could alternatively be translated to a syntax with guarded commands [4], such as Abstract State Machines [8,9].

2.1 Model Syntax and Semantics

We start by introducing B event systems in Def. 1. The events are defined by means of guarded actions [4] by composition of five primitive actions where a, a_i are actions, E, F are arithmetic expressions and P, P' are predicates: *skip* an action with no effect, $x, y := E, F$ a multiple assignment, $P \Rightarrow a$ a guarded action, $a_1 [] a_2$ a bounded non-deterministic choice between a_1 and a_2, and $@z.a$ an unbounded non-deterministic choice $a_{z_1} [] a_{z_2} [] \ldots$ for all the values of z satisfying the guard of a denoted as $\mathsf{grd}(a)$. Here grd is defined on the primitive actions by: $\mathsf{grd}(skip) \stackrel{\text{def}}{=} true$, $\mathsf{grd}(x, y := E, F) \stackrel{\text{def}}{=} true$, $\mathsf{grd}(P' \Rightarrow a) \stackrel{\text{def}}{=} P' \wedge \mathsf{grd}(a)$, $\mathsf{grd}(a_1 [] a_2) \stackrel{\text{def}}{=} \mathsf{grd}(a_1) \vee \mathsf{grd}(a_2)$ and $\mathsf{grd}(@z.a) \stackrel{\text{def}}{=} \exists z \cdot \mathsf{grd}(a)$.

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 a state invariant, $Init$ is an* initialization *action such that I holds in any initial state, EvDef is a set of event definitions, each in the shape of $e \stackrel{\text{def}}{=} a$ for any $e \in$* Ev*, and such that every event preserves I.*

Following [10], we use labelled transition systems to define the semantics of event systems. An example of a B event system will be provided in Sect. 3.

Let $e \stackrel{\text{def}}{=} a$ be an event. It has a *weakest precondition* [5] and a *weakest conjugate precondition* [10] w.r.t. a set of target states Q' denoted respectively as $wp(a, Q')$ and $wcp(a, Q')$. $wp(a, Q')$ is the largest set of states from which applying a always leads to a state of Q' whereas $wcp(a, Q')$ is the largest set of states from which it is possible to reach a state of Q' by applying a. An event also defines a relation between the values of the state variables X before and after the application of the event. It is expressed by the before-after predicate of the event $e \stackrel{\text{def}}{=} a$ denoted as $prd_X(a)$.

Let us now formally define wp, wcp and prd_X following [11]. 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 wp w.r.t. the five primitive actions as:

– $wp(skip, Q') \stackrel{\text{def}}{=} Q'$,
– $wp(x := E, Q') \stackrel{\text{def}}{=} Q'[E/x]$ that is the usual substitution of x by E,
– $wp(P \Rightarrow a, Q') \stackrel{\text{def}}{=} P \Rightarrow wp(a, Q')$,
– $wp(a_1 [] a_2, Q') \stackrel{\text{def}}{=} wp(a_1, Q') \wedge wp(a_2, Q')$,
– $wp(@z.a, Q') \stackrel{\text{def}}{=} \forall z.wp(a, Q')$ where z is only bound by predicates in a.

We define the wcp and prd_X w.r.t. wp as:

– $wcp(a, Q') \stackrel{\text{def}}{=} \neg wp(a, \neg Q')$,
– $prd_X(a) \stackrel{\text{def}}{=} wcp(a, x_1' = x_1 \wedge \ldots \wedge x_n' = x_n)$ that is a predicate on 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 [1] is a special instance of the framework of abstract interpretation [12] that maps the potentially infinite state space C of a concrete transition system onto the finite state space A of an abstract transition system *via* a set of n predicates $\mathcal{P} \overset{\text{def}}{=} \{p_1, p_2, \ldots, p_n\}$ over the state variables. The set of abstract states A contains 2^n states. Each state is a tuple $q \overset{\text{def}}{=} (q_1, q_2, \ldots, q_n)$ with q_i being equal either to p_i or to $\neg p_i$, and we also consider q as the predicate $\bigwedge_{i=1}^{n} q_i$. We define a total abstraction function $\alpha : C \to \mathsf{A}$ such that $\alpha(c)$ is an abstract state q where c satisfies q_i for all $i \in 1..n$. By a misuse of language, we say that c is in q, or that c is a concrete 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 least one concrete transition $c \xrightarrow{e} c'$ where c and c' are concrete states with $\alpha(c) = q$ and $\alpha(c') = q'$.

We check predicate satisfiability thanks to SMT solvers. For a predicate P, we define the solver call $SAT_c(P)$ 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 (showing that P is satisfiable). Let $e \overset{\text{def}}{=} a$ be an event definition, $q \xrightarrow{e} q'$ is a *may* transition iff $SAT(wcp(a, q') \wedge q)$. 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 predicate q' in which each state variable x_i is substituted by x'_i.

2.3 May Transition Systems

Let us introduce *may* transition systems (MTS), which are transition systems with abstract states, and abstract *may* transitions. They are related to Modal Transition Systems [13,14,15], but with only the *may* modality.

Definition 2 (May Transition System). *Let* Ev *be a finite set of event names and $\mathcal{P} \overset{def}{=} \{p_1, p_2, \ldots, p_n\}$ 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 $\langle Q, Q_0, \Delta \rangle$ 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,*
- *$\Delta(\subseteq Q \times \mathrm{Ev} \times Q)$ is a may labelled transition relation.*

Now, Definition 3 associates an abstraction defined by an MTS with an event system.

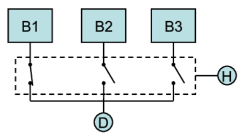
Definition 3 (MTS associated with an ES). *Let $ES \overset{def}{=} \langle X, I, Init, EvDef \rangle$ be an event system and $\mathcal{P} \overset{def}{=} \{p_1, p_2, ..., p_n\}$ be a set of n predicates over variables of X defining a set of 2^n abstract states $\mathsf{A} \overset{def}{=} \{p_1, \neg p_1\} \times \{p_2, \neg p_2\} \times ... \times \{p_n, \neg p_n\}$. A tuple $\langle Q, Q_0, \Delta \rangle$ is an MTS associated to ES and $\mathcal{P}$ if it satisfies the following conditions:*

- $Q \overset{def}{=} \{q \in \mathsf{A} | \exists (q', e).(q \overset{e}{\to} q' \in \Delta \vee q' \overset{e}{\to} q \in \Delta)\}$,
- $Q_0 \overset{def}{=} \{q | q \in \mathsf{A} \wedge (SAT(prd_X(Init) \wedge q[X'/X]))[X/X']\}$,
- $\Delta \overset{def}{=} \{q \overset{e}{\to} q' | q \in \mathsf{A} \wedge q' \in \mathsf{A} \wedge e \overset{def}{=} a \in EvDef \wedge SAT(wcp(a, q') \wedge q)\}$.

Further in this paper, in Fig. 2 of Sect. 4, the reader will find an MTS example, whose ES is described in Fig. 1(b). The MTS is the part shown in dashed lines, with the four abstract states named q_0 to q_3 appearing as rounded rectangular dashed boxes. The abstract transitions of Δ are represented as dashed arrows labelled by event names.

3 Illustrative Example: an Electrical System

To illustrate our approach, this section describes an electrical (**EL**) system example. It is a finite state control and command system that illustrates the MTS, as represented in Fig. 2.



(a) Physical representation

$$
\begin{array}{lll}
X & \overset{\mathsf{def}}{=} & \{H,\ Sw,\ Bat\} \\
I & \overset{\mathsf{def}}{=} & H \in \{tic, tac\} \wedge Sw \in 1..3 \wedge (Bat \in 1..3 \to \{ok, ko\}) \wedge \\
 & & Bat(Sw) = ok \\
Init & \overset{\mathsf{def}}{=} & H,\ Sw,\ Bat\ :=\ tac, 1,\ \{1 \mapsto ok,\ 2 \mapsto ok,\ 3 \mapsto ok\} \\
Tic & \overset{\mathsf{def}}{=} & H = tac \Rightarrow H := tic \\
Com & \overset{\mathsf{def}}{=} & \exists(i, j).(i \in 1..3 \wedge j \in 1..3 \wedge i \neq j \wedge Bat(i) = ok \wedge \\
 & & Bat(j) = ok) \wedge H = tic \Rightarrow \\
 & & \quad @ns.(ns \in 1..3 \wedge Bat(ns) = ok \wedge ns \neq Sw \Rightarrow H, Sw := tac, ns) \\
Fail & \overset{\mathsf{def}}{=} & \exists(i, j).(i \in 1..3 \wedge j \in 1..3 \wedge i \neq j \wedge Bat(i) = ok \wedge Bat(j) = ok) \Rightarrow \\
 & & \quad @nb.(nb \in 1..3 \wedge Bat(nb) = ok) \Rightarrow \\
 & & \qquad (nb = Sw \Rightarrow @ns.(ns \in 1..3 \wedge ns \neq Sw \wedge \\
 & & \qquad Bat(ns) = ok \Rightarrow Sw, Bat(nb) := ns, ko)) \\
 & & \qquad [](nb \neq Sw \Rightarrow Bat(nb) := ko)) \\
Rep & \overset{\mathsf{def}}{=} & @nb.(nb \in 1..3 \wedge Bat(nb) = ko) \Rightarrow Bat(nb) := ok)
\end{array}
$$

(b) Specification

Figure 1. Electrical System and its specification

Figure 1(a) shows a device D powered *via* a switch to one of three batteries B_1, B_2, and B_3. A clock H periodically sends a signal that causes a commutation of the closed switch. The system has to meet the following requirements: one and only one switch is closed at a time, and a clock signal changes the switch that is closed. The batteries may break down. If it happens to the one that is powering D, an exceptional commutation is triggered. We assume that there is always at least one battery working. When there is only one battery working, the clock signals are ignored.

The event system in Fig. 1(b) uses three variables. H models the clock and takes two values: *tic* to ask for a commutation, and *tac* when it has occurred. Sw models the switches by indicating which one is closed. Bat models the batteries breakdowns by a total function that associates *ok* or *ko* (for a broken battery) to each battery. The state changes occur by applying four events: *Tic* sends a commutation signal, *Com* changes the closed switch responding to a *Tic*, *Fail* breaks down at random a battery, and *Rep* repairs at random a broken battery.

The MTS (in dashed lines) of Fig. 2 in the next section abstracts the model of Fig. 1(b) w.r.t. the set of abstraction predicates $\mathcal{P}_0 \stackrel{\text{def}}{=} \{p_1, p_2\}$, where $p_1 \stackrel{\text{def}}{=} H = tic$ (meaning that a commutation is asked) and $p_2 \stackrel{\text{def}}{=} \exists(i,j).(i \in 1..3 \wedge j \in 1..3 \wedge i \neq j \wedge Bat(i) = ok \wedge Bat(j) = ok)$. p_2 means that at least two batteries are *ok*, so that a single battery is left working in the states where it is false.

4 Abstraction and Approximated Transition System Computation

This section presents an algorithm used to compute both an abstraction that is an MTS, and an under-approximation. The reunion of both is called an Approximated Transition System (ATS, defined in Def. 4), in which $\langle C, C_0, \Delta^c\rangle$ is an under-approximation of the labelled transition system that is the semantics of the event system from which the MTS is deduced.

Definition 4 (Approximated Transition System). *Let $\langle Q, Q_0, \Delta\rangle$ be an MTS. A tuple $\langle Q, Q_0, \Delta, C, C_0, \alpha, \Delta^c\rangle$ is an ATS whose $\langle C, C_0, \alpha, \Delta^c\rangle$ is a concretization of the MTS $\langle Q, Q_0, \Delta\rangle$ where:*

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

For example, Fig. 2 shows an ATS of the electrical system of Fig. 1(a). The MTS appears in dashed lines while the full lines represent its concretization. The concrete states are showed as big dots.

In the rest of the paper, for the abstract states, we distinguish between the *may-reachability* and the *reachability*. The former is the reachability by the abstract *may* transition relation Δ, and the latter is the reachability by the concrete transition relation Δ^c in the ATS. We say that an abstract state q is reachable if there exists at least one concrete instance of q that is reachable thanks to the transition relation Δ^c. By extension, an abstract transition is reachable if there exists at least one concrete instance in Δ^c whose source state is reachable.

The ATS computation algorithm that we present concretizes the *may* transitions on the fly during the MTS computation. It guarantees that every *may* transition between two abstract states is concretized. A total abstraction function α maps each concrete state of C to an abstract state of Q. The algorithm comes in two versions, both of them being presented in the same figure. The first version is the one presented in this section. It is referred to as Alg. 1. The second version, referred to as Alg. 2, is enhanced by heuristics that are explained in Sect. 5. The differences between Alg. 1 and Alg. 2 are highlighted and enclosed in square brackets. Read the left hand highlighted parts for Alg. 1, and replace them with the right hand ones for Alg. 2. Notice that since Alg. 2 computes more things than Alg. 1, some fictive empty parts (the empty highlighted square brackets [ ]) have been added in Alg. 1.

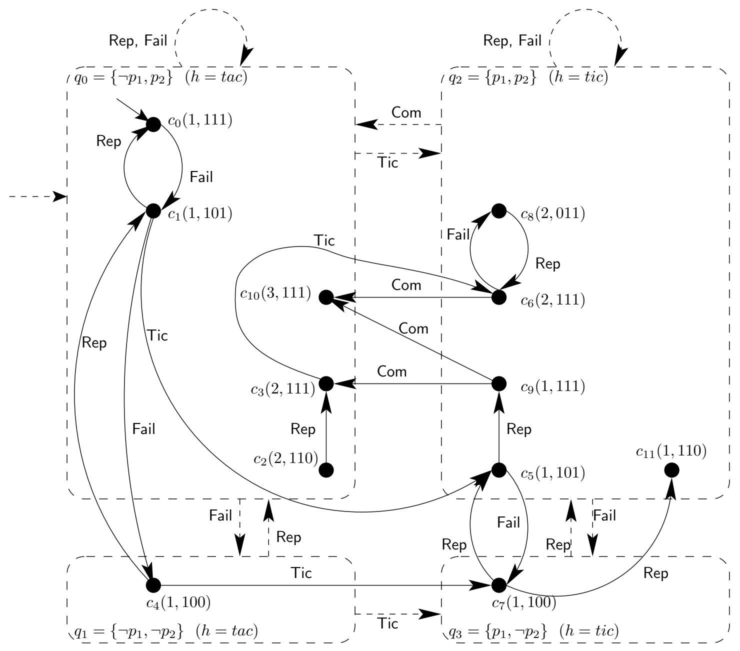


Figure 2. Example MTS and ATS of the Electrical System. The values of the concrete states are indicated in parentheses right by them. For example with state c_8, $(2, 011)$ means that battery 2 is used, and that battery 1 is *ko* while batteries 2 and 3 are *ok*. The value of h is given globally in the abstract states.

Lines 1 to 8 of Alg. 1 compute the set of initial abstract states Q_0, an instance of each being recorded as a concrete witness in C_0 with its association in α. Lines 9 to 35 compute the *may* transition relation Δ. Each abstract transition is concretized by a witness $\{c_w \xrightarrow{e} c'_w\}$, and the concrete states c_w and c'_w are recorded in C with their associations in α. For that it computes in the set RQ the set of *may-reachable* states. For each *may-reachable* source state, it checks for each potential abstract state (line 12) and for each event (line 13) if a *may* transition exists (line 14). When it is the case, the algorithm records the witness transition (see lines 16 and 28), but also possibly another concrete transition (see lines 17 to 27) whose source state is one of the existing concrete states of the current source state q when it exists. This last transition is computed first to improve the *reachability* of the concrete transition relation. Indeed, the existing concrete states are more likely to be connected to the initial states than the witness source state provided by the solver in line 14. Last, line 31 adds q' as a *may-reachable* state that has not been taken into account yet to compute the *may* transition relation.

The algorithm terminates because it iterates on a finite number of abstract states and events. It is sound because the transitions computed are concrete instances of the semantics of the event system.

Algorithm: ATS computation [**1**: without heuristics] [**2**: with heuristics]

```
Inputs    : $\langle X, I, Init, \mathrm{EvDef}\rangle$: an Event System where $\mathrm{EvDef} \overset{\text{def}}{=} \{e \overset{\text{def}}{=} a \mid e \in Ev\}$
            $A$: a finite set of abstract states
            [ ]          [$orderStates$: $2^A \times Q \rightarrow$ $list\ of\ Abstract\ States$
                              (ordering function of the abstract states)]
            [ ]          [$oEv$: ordered list of the events of $Ev$]
Output    : $\langle Q, Q_0, \Delta, C, C_0, \alpha, \Delta^c,$ [ ] $\rangle$:        [$\kappa$]
            an ATS [ ]      [provided with a coloration function $\kappa \in C \rightarrow \{green, blue\}$]
Variables : $RQ$: the set of abstract states remaining to be handled
            $q, q'$: the source and target abstract states of the current transition
            $c, c'$: the concrete source and target states of respectively $q$ and $q'$
            $c_w, c'_w$: the witness concrete source and target states of a $may$-transition
            $GC$: the set of [already known] concrete states of $q$        [$green$ $C$-$reachable$]
            [ ]           [$BC$: the set of $blue$ concrete states of $q$]

$Q := \emptyset$;  $Q_0 := \emptyset$;  $\Delta := \emptyset$;  $C_0 := \emptyset$;  $\alpha := \emptyset$;  $\Delta^c := \emptyset$; [ ]        [$\kappa := \emptyset$;]
foreach $q \in A$ do
    $c := (SAT_c(prd_X(Init) \wedge q[X'/X]))[X/X']$
    if $c \notin \{unsat, unknown\}$ then
        $Q_0 := Q_0 \cup \{q\}$; $C_0 := C_0 \cup \{c\}$; $\alpha(c) := q$
        [ ]                                    [$\kappa(c) := green$;]
    end
end
$C := C_0$;  $RQ := Q_0$;
while $RQ \neq \emptyset$ do
    choose $q \in RQ$;  $RQ := RQ - \{q\}$;  $Q := Q \cup \{q\}$
    foreach $q' \in$ [$A$] do                          [list $orderStates(A, q)$]
        foreach $(e \overset{def}{=} a) \in$ [$EvDef$] do              [list $oEv$]
            $(c_w, c'_w) := SAT_c(prd_X(a) \wedge q'[X'/X] \wedge q)$
            if $(c_w, c'_w) \notin \{unsat, unknown\}$ then
                $\Delta := \Delta \cup \{q \xrightarrow{e} q'\}$;
                $GC := \{c_q \mid \alpha(c_q) = q$ [ ]$\}$          [$\wedge\ \kappa(c_q) = green$]
                $(c, c') := SAT_c(prd_X(a) \wedge q'[X'/X] \wedge \bigvee_{c_q \in GC} c_q)$
                if $(c, c') \notin \{unsat, unknown\}$ then
                    $C := C \cup \{c'\}$; $\alpha(c') := q'$; $\Delta^c := \Delta^c \cup \{c \xrightarrow{e} c'\}$;
                    [ ]          [$\kappa(c') := green$; $BC := \{c'_q \mid \alpha(c'_q) = q' \wedge \kappa(c'_q) = blue\}$;]
                    [ ]          [$(c, c') := SAT_c(prd_X(a) \wedge (\bigvee_{c'_q \in BC} c'_q)[X'/X] \wedge \bigvee_{c_q \in GC} c_q)$;]
                    [ ]          [if $(c, c') \notin \{unsat, unknown\}$ then
                                         $\Delta^c := \Delta^c \cup \{c \xrightarrow{e} c'\}$;
                                         $recursively\ colour\ in\ green\ from\ c'$;
                                 end]
                end
                $C := C \cup \{c_w, c'_w\}$; $\Delta^c := \Delta^c \cup \{c_w \xrightarrow{e} c'_w\}$; $\alpha(c_w) := q$; $\alpha(c'_w) := q'$;
                [ ]          [if $c_w \notin domain(\kappa)$ then $\kappa(c_w) := blue$ end ]
                [ ]          [if $c'_w \notin domain(\kappa) \vee \kappa(c_w) = green$ then $\kappa(c'_w) := \kappa(c_w)$ end ]
                if $q' \notin Q$ then $RQ := RQ \cup \{q'\}$ end
            end
        end
    end
end
```

5 Heuristics for Better Abstraction Coverage

Using Alg. 1, the connectivity and the reachability of the computed ATS might be weak, depending on which witnesses are exhibited by the solver. This section provides two heuristics, integrated into Alg. 2, for improving both the connectivity and the reachability. The first heuristic addresses this problem by allowing the engineer to firstly define an order for the set of events of *Ev* in an ordered list *oEv* (line 13) and, secondly, a custom function ordering the set of abstract states A (line 12). The second heuristic, exposed in Sect. 5.2, adapts the partial computation of reachability proposed in [6] to our purpose for integrating it into Alg. 2. The resulting new algorithm's complexity is the same as the previous one. The ATS of Fig. 2 was obtained by applying Alg. 2 to the electrical system of Fig. 1(a), w.r.t. the set of predicates $\mathcal{P}_0$ (defined in Sect. 3). The concrete states are numbered according to their order of discovery by Alg. 2.

5.1 Events and States Ordering

Our first heuristic consists of providing means to control the order in which the events and abstract target states are handled by the algorithm.

Indeed, usually in reactive systems, some events can only be fired after other events have previously been executed. Let us consider the **EL** system where no battery repairing (modelled by the *Rep* event) can occur unless at least one battery has broken down first (modelled by the $Fail$ event). Since Alg. 1 currently uses an unordered set of events *EvDef*, it might attempt to concretize a *Rep* transition before trying to concretize any $Fail$ transition. In this case, the concrete source state of the *Rep* transition would not be a reachable one. To fix this, we introduce the ordered list of events *oEv* as an input in Alg. 2. To compute a complete abstraction, i.e. covering all events and all states, *oEv* must contain at least one occurrence of each event of the set *EvDef*. For example, for the **EL** system, in all judicious orders, $Fail$ must precede *Rep* for the aforementioned reason, and Tic must precede *Com* because *Com* is a response to the event Tic.

Similarly, the *orderStates* function parameterizes Alg. 2. Thanks to this function, the order in which the abstract target states are handled can be controlled. To compute a complete abstraction, the list returned by *orderStates* must contain at least all the abstract states of A. While being completely customizable by the engineer, the function used in our experiments presented in Sect. 6 gives better results for an order in which the first target abstract state handled is the source abstract state (state q in line 11) and the other states are ordered arbitrarily. Indeed, treating reflexive abstract transitions first tends to increase the number of reachable concrete states within the source abstract state. As a result, the chances that the next abstract transitions can be concretized from a reachable source state are increased.

When applying Alg. 1 to the **EL** system with a set of abstraction predicates (first **AP** in Table 1), 33.33% of the abstract states and 11.11% (see line 1) of the abstract transitions are covered by the ATS. Integrating the event and state ordering without coloration improved these ratios respectively to 66.67% and 44.44%.

These ordering heuristics are integrated into Alg. 2, along with the concrete states coloration discussed in Sect. 5.2. Even though our results did not focus on that point, an interesting perspective could be to consider the concretization of a same abstract transition multiple times. This could be useful for instance for systems requiring initialization steps that repetitively apply the same event, such as a credit card system for example. In fact, this behaviour can already be implemented using our algorithm by adding the same event to *oEv* several times, and by adding the target state of the abstract transition several times to the list returned by the *orderStates* function.

5.2 Concrete States Coloration

The reachability of the concrete states of the under-approximation is improved and computed on the fly in Alg. 2 at no additional cost w.r.t. Alg. 1. The principle is to associate a colour with each concrete state. A reachable state is coloured in green and a state whose reachability is unknown is coloured in blue. While Alg. 1 tried to concretize the abstract transitions from an already known concrete state, Alg. 2 uses the reachability information when concretizing the abstract transitions. It first tries to concretize an abstract transition from any known and *reachable* state (see lines 17 and 18). If it is indeed possible (line 19), the solver returns a first reachable concrete transition, added to the ATS, whose concrete target state becomes green (see lines 20 and 21). To improve the connectivity, the algorithm also tries to join a green source concrete state to a target blue one, whose *reachability* is thus currently unknown (line 22). If it is possible (line 23), its colour becomes green (line 24) since it is a target of a concrete transition starting from a *reachable* (green) concrete state. Even if the concretization from a known green concrete state is not possible, the abstract transition is still concretized. The corresponding concrete source and target states may already be known. In this case, their reachability remain the same. Otherwise, since we have no information about their reachability, they are coloured in blue (see lines 28 and 29).

When applying Alg. 2 with coloration and without events and states ordering to the **EL** system with the first set of abstraction predicates in Table 1, 100% of the abstract states and 77.78% of the abstract transitions are covered by the ATS. This is better than with Alg. 1 that gives respectively 33.33% and 11.11%. Moreover, integrating the two heuristics seen in Sect. 5.1 into Alg. 1 improved these two ratios to 100%.

These heuristics allow improving the reachability of the ATS for all the systems that we use in our experiments (see Sect. 6.2).

6 Implementation and Experimentation

This section introduces in Sect. 6.1 our proof-of-concept tool developed to evaluate the effects of the heuristics. The experimental results on four examples in Table 1 are presented in Sect. 6.2. Then, in Sect. 6.3 we analyse these results and conclude on the contributions of the heuristics presented in this paper.

6.1 About the Tool

The developed tool can be seen as a library for handling abstract and concrete transition systems as well as event systems. It embeds an event-B parser and allows to manipulate most event-B systems. The two algorithms are implemented and can be applied to them. The library also provides the user with many facilities for dealing with event systems. For instance, pre-implemented functions allow to easily compute an abstraction of a model from a set of abstraction predicates, as well as the wp, wcp and before-after predicates prd_X of events defined by guarded actions. The library also contains functions to check the modality of abstract transitions and to find a concretization of an abstract state or an abstract transition. It can also be seen as a simple API for multiple SMT-solvers since the tool automatically generates SMT-Lib2 code for checking the satisfiability of any first order boolean formula. The tool is constituted of more than 5000 lines of JAVA code (version 8) and uses Z3 [16] as default SMT-solver. The library can be downloaded at *https://github.com/stratosphr/stratest/wiki*. This website also gives information on how to use the tool.

6.2 Experimental Results

This section provides the results obtained when applying Alg. 1 and Alg. 2 to a set of four realistic event systems of increasing size. These event systems, available in the aforementioned GitHub repository, were taken back from various previous work without modification so as not to influence the experiment and threaten the validity of the results. The set of examples contains the electrical system (**EL**) presented in Sect. 3, a phone book service (**PH**), a coffee machine system (**CM**), and a car alarm system (**CA**). For each of them, two different sets of abstraction predicates have been used (see the **AP** column).

The following column names appear in Table 1: **Sys** for the system studied and an upper approximation of its size between parentheses, **#Ev** for the number of events in the event system, **AP** for an identification of the set of abstraction predicates used, **#AP** for the number of abstraction predicates in **AP**, **Alg.** for the algorithm applied, **#AS** for the number of *may-reachable* abstract states, $\mathbf{\#AS}_{reach}$ for the number of *reachable* abstract states computed, τ_{AS} for the abstract state coverage that is the ratio $\frac{\#AS_{reach}}{\#AS}$, **#AT** for the number of abstract transitions, $\mathbf{\#AT}_{reach}$ for the number of *reachable* abstract transitions computed. Note that we say that an abstract transition is *reachable* if there exists a concrete instance of it in the ATS that is reachable. Next, there are the following column names: τ_{AT} for the abstract transition coverage that is the ratio $\frac{\#AT_{reach}}{\#AT}$, **#CT** for the number of concrete transitions computed, ρ_{CT} for the ratio $\frac{\#CT}{\#AT_{reach}}$ which measures the efficiency of the method, by indicating in average how many concrete transitions have been computed for making an abstract transition reachable, and finally **Time** for the ATS computation runtime (in seconds). The connectivity between transitions is indirectly measured via the coverage and efficiency rates, since a reachable state or transition is necessarily *connected* to a concrete initial state.

The main results of our method are the coverage ratios of abstract states (τ_{AS}) and abstract transitions (τ_{AT}). For almost identical computation time(s), an improvement of these ratios indicates a better performance of the method.

For ρ_{CT}, a value between 3 and 1 indicates that the algorithm covers one abstract transition per iteration step. When this ratio decreases that indicates an improvement of the efficiency. Indeed, for each abstract transition, each iteration step computes one up to three transitions according to the conditions in lines 19 and 23. For the **EL** system, with the first set of abstraction predicates, ρ_{CT} decreases from 13 to 2, meaning that the heuristic allowed to compute more interesting concrete transitions, increasing the abstraction transition coverage from 11.11% to 100%.

Table 1. ATS computation results

Sys	**#Ev**	**AP**	**#AP**	**Alg.**	**#AS**	**#AS**$_{reach}$	τ_{AS}(%)	**#AT**	**#AT**$_{reach}$	τ_{AT}(%)	**#CT**	ρ_{CT}	**Time**
EL (24)	4	1	2	1	3	1	33.33	9	1	11.11	13	13	00.283
				2	3	3	100	9	9	100	18	2	00.297
		2	2	1	4	4	100	11	8	72.73	15	1.88	00.429
				2	4	4	100	11	11	100	17	1.55	00.449
PH (2^{10})	4	1	3	1	3	3	100	12	11	91.67	16	1.45	00.261
				2	3	3	100	12	12	100	22	1.83	00.287
		2	6	1	8	8	100	62	60	96.77	83	1.38	01.994
				2	8	8	100	62	62	100	88	1.42	02.204
CM (2^{16})	8	1	3	1	4	3	75	30	5	16.67	47	9.4	00.753
				2	4	4	100	30	24	80	54	2.25	00.854
		2	3	1	6	3	50	52	7	13.46	83	11.86	01.639
				2	6	6	100	52	25	48.08	77	3.08	01.629
CA (2^{15})	20	1	6	1	8	5	62.5	31	18	58.065	44	2.44	13.818
				2	8	8	100	31	25	80.65	50	2	13.283
		2	9	1	9	5	55.56	30	11	36.67	37	3.36	23.978
				2	9	9	100	30	28	93.33	52	1.86	25.381

6.3 Analysis of the Obtained Results

This section comments on the results exposed in Table 1.

As expected, the ATS computation times are nearly identical, no matter which version of the algorithm has been used. Note that for two cases out of eight the ATS computation time with Alg. 2 is on average slightly faster than with Alg. 1. Since the formulas whose satisfiability is checked are different between the two algorithms, the solver may be faster to provide an answer for the formulas in Alg. 2 than in Alg. 1.

We observe that Alg. 2 improves both the abstraction coverage rates and the efficiency ρ_{CT} compared to Alg. 1. In particular, we point out the **CM** case with the first **AP** where the transition coverage and the efficiency achieved by Alg. 2 is about respectively five and four times better than by Alg. 1.

For all systems and all sets of abstraction predicates, the abstraction coverage is improved by Alg. 2. Depending on the set of predicates used, the coverage for states and transitions can reach up to 100%. On most examples however, Alg. 1 covers less than half of the abstract states and transitions. Note for example the **CM** case with the second set of abstraction predicates where the abstract states and transitions coverage(s) are respectively twice and three times better using the heuristics. All these results empirically confirm the interest of the proposed heuristics to improve the abstraction coverage.

The heuristics also produced good results concerning the efficiency rate ρ_{CT}. For most cases, its value is decreased by Alg. 2 thanks to the heuristics, which means that they generally help concretizing abstract transitions by useful transitions improving the abstraction coverage. For the **CM** system with the second **AP** and Alg. 1 for example, an average of 11.86 concrete transitions need to be computed in order to cover one abstract transition. When the heuristics are used however, an average of only 3.08 concrete transitions computation is needed to cover one abstract transition.

The ordering heuristic alone does not necessarily improve the abstraction coverage w.r.t. Alg. 1, whereas the coloration heuristic alone always improves the results. Nevertheless, for four cases out of the eight presented in this paper, combining the ordering and coloration heuristics improves the abstraction coverage compared to coloration only. For the **CM** system with the first **AP** for example, the ordering heuristic alone covers two abstract states compared to three with Alg. 1, and six abstract transitions compared to five. The coloration heuristics alone covered all of the four abstract states, and twenty-two out of the thirty abstract transitions. When combining both heuristics, the four abstract states and twenty-four abstract transitions are covered.

7 Related Work

In [17] and in [18], the set of abstraction predicates is iteratively refined in order to compute a bisimulation of the semantics of the 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 [19] and DASH [20] combine under-approximation and over-approximation computations to check safety properties on programs. As we aim at proposing an efficient method to build a *reachable* under-approximation of a system that covers all abstract states and all abstract transitions w.r.t. a specification and a set of predicates, our algorithm does not refine the approximation and so always terminates.

The closest methods to ours are those that are proposed in [21] and in [6]. These approaches propose algorithms that compute an under-approximated concretization of a predicate abstraction covering its abstract states and transitions. Both these methods are exploited for generating tests. The algorithm in [21] does not traverse nor compute the *may* abstraction. It builds a partial concretization of the abstract states that are *reached* from an initial concrete state by a forward walk. To improve this method, the algorithm in [6] computes exhaustively the *may* abstraction by random abstract state generation. Therefore, some generated concrete states are not *reached*. Then Veanes and al. [6] propose to distinguish between four kinds of abstract transitions: green transitions when there exists an instance that is reached from an initial concrete state, blue transitions when there exists instances, but none known to be reachable from an initial state, red transitions when there does not exist any instance, and grey transitions for the transitions that have not been concretized yet. In our method, we compute and concretize only the part of the *may* abstraction that is *may-reachable* by an abstract transition from an initial abstract state. We do not record the red transitions that are non-existing transitions in the MTS, and we do not need the grey transitions that are the ones which remain to be treated. In contrast with [6], our method colours the concrete states instead of the abstract transitions. This allows us to distinguish between the *reached* states (green) and the states for which we do not know whether they are *reached* (blue) or not. So for improving the method, Alg. 2 connects in priority a green source state s to a blue target state t. That has a "domino effect" because all the blue reached states from t remain blue, but become *reachable*.

Some other work under-approximate an abstraction for generating tests. The tools Agatha [22], DART [23], CUTE [24], EXE [25] and PEX [26] also compute abstractions from models or programs, but only by means of symbolic executions [27]. This data abstraction approach computes an execution graph. Its set of abstract states is possibly infinite whereas it is finite with our method.

Our method can be applied to generate tests as the concolic execution in [24]. The concolic method allows to generate structural tests of systems covering partially the control flow that must be explicited. Our approach allows to generate tests covering the paths defined by the set of abstraction predicates for systems whose control flow is implicitly defined.

8 Conclusion and Further Work

This paper has presented an algorithmic method for computing a concrete approximation of the predicate abstraction of an event system. All of the abstract states and transitions are covered, but as the control flow is implicit in an event system, our method focuses on computing concrete sequences that are connected and reachable. We have presented two heuristics allowing us to better reach and connect these sequences. One heuristic colours the states that are known to be reachable, and the other takes a user defined order on the events and abstract states enumeration as parameters. Experimental results on four case studies are exhibited to confirm the practical interest of our approach.

As future work, we intend to define other means for guiding the sequences instantiation, in addition to the events ordering. We could introduce a relevance function on concrete states, as is done in [21], for targeting peculiar concrete states considered as more relevant. Also, our intention is to use the concrete sequences that we compute as model-based tests issued from a formal model of the specification. Abstracting this model would allow selection criteria such as paths selection to be used, when the size of the explicit model would prevent it.

References

1. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: CAV. Volume 1254 of LNCS., Springer (1997) 72–83

2. Godefroid, P., Jagadeesan, R.: On the expressiveness of 3-valued models. In: VMCAI. Volume 2575 of LNCS., Springer (2003) 206–222
3. Abrial, J.R.: Modeling in Event-B: System and Software Design. Cambridge Univ. Press (2010)
4. Dijkstra, E.: Guarded commands, nondeterminacy, and formal derivation of programs. Com. of the ACM **18**(8) (1975) 453–457
5. Dijkstra, E.: A Discipline of Programming. Prentice-Hall (1976)
6. Veanes, M., Yavorsky, R.: Combined algorithm for approximating a finite state abstraction of a large system. In: ICSE 2003/Scenarios Workshop. (2003) 86–91
7. Abrial, J.R.: The B Book. Cambridge Univ. Press (1996)
8. Gurevich, Y., Kutter, P.W., Odersky, M., Thiele, L.: Abstract State Machines, Theory and Applications. Volume 1912 of LNCS. Springer (2000)
9. Gurevich, Y.: Sequential abstract-state machines capture sequential algorithms. ACM Trans. Comput. Log. **1**(1) (2000) 77–111
10. Bert, D., Cave, F.: Construction of finite labelled transition systems from B abstract systems. In: IFM. (2000) 235–254
11. Bride, H., Julliand, J., Masson, P.A.: Tri-modal under-approximation for test generation. Science of Computer Programming **132**(P2) (2016) 190–208
12. Cousot, P., Cousot, R.: Abstract interpretation frameworks. J. Log. Comput. **2**(4) (1992) 511–547
13. Larsen, K.G., Thomsen, B.: A modal process logic. In: LICS. (1988) 203–210
14. Godefroid, P., Huth, M., Jagadeesan, R.: Abstraction-based model checking using modal transition systems. In: CONCUR. (2001) 426–440
15. Ball, T.: A theory of predicate-complete test coverage and generation. In: FMCO. Volume 3657 of LNCS. (2004) 1–22
16. de Moura, L., Bjorner, N.: An efficient SMT solver. In: TACAS. Volume 4963 of LNCS. (2008) 337–340
17. Namjoshi, K.S., Kurshan, R.P.: Syntactic program transformations for automatic abstraction. In: CAV. Volume 1855 of LNCS. (2000) 435–449
18. Păsăreanu, C.S., Pelánek, R., Visser, W.: Predicate abstraction with under-approximation refinement. LMCS **3**(1:5) (2007) 1–22
19. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Synergy: a new algorithm for property checking. In: SIGSOFT FSE. (2006) 117–127
20. Beckman, N.E., Nori, A.V., Rajamani, S.K., Simmons, R.J., Tetali, S., Thakur, A.V.: Proofs from tests. IEEE Trans. Software Eng. **36**(4) (2010) 495–508
21. Grieskamp, W., Gurevich, Y., Schulte, W., Veanes, M.: Generating finite state machines from abstract state machines. In: ISSTA. (2002) 112–122
22. Rapin, N., Gaston, C., Lapitre, A., Gallois, J.P.: Behavioral unfolding of formal specifications based on communicating extended automata. In: ATVA. (2003)
23. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: PLDI. (2005) 213–223
24. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: ESEC/SIGSOFT FSE. (2005) 263–272
25. 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
26. Tillmann, N., de Halleux, J.: Pex-white box test generation for .net. In: TAP. Volume 4966 of LNCS. (2008) 134–153
27. Păsăreanu, C.S., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. STTT **11**(4) (2009) 339–353