Software Quality Journal manuscript No.
(will be inserted by the editor)

B Model Slicing and Predicate Abstraction to Generate Tests

J. Julliand · N. Stouls · P.-C. Bué · P.-A. Masson

the date of receipt and acceptance should be inserted later

Abstract In a model-based testing approach as well as for the verification of properties, B models provide an interesting modelling solution. However, for industrial applications, the size of their state space often makes them hard to handle. To reduce the amount of states, an abstraction function can be used. The abstraction is often a domain abstraction of the state variables that requires many proof obligations to be discharged, which can be very time consuming for real applications.

This paper presents a contribution to this problem that complements an approach based on domain abstraction for test generation, by adding a preliminary syntactic abstraction phase, based on variable elimination. We define a syntactic transformation that suppresses some variables from a B event model, in addition to three methods that choose relevant variables according to a test purpose. In this way, we propose a method that computes an abstraction of a source model M according to a set of selected relevant variables. Depending on the method used, the abstraction can be computed as a simulation or as a bisimulation of M. With this approach, the abstraction process produces a finite state system. We apply this abstraction computation to a Model Based Testing process. We evaluate experimentally the impact of the model simplification by variables elimination on the size of the models, on the number of proof obligations to discharge, on the precision of the abstraction and on the coverage achieved by the test generation.

Keywords Abstraction · Test Generation · (Bi)Simulation · Slicing

1 Introduction

B models are well suited for producing tests of an implementation by means of a *model-based testing* approach [BJK+05, UL06] as well as to verify dynamic properties by model-

Julliand, Bue, Masson
LIFC, Université de Franche-Comté
16, route de Gray F-25030 Besançon Cedex, France
E-mail: julliand, bue, masson@lifc.univ-fcomte.fr

Stouls
Université de Lyon, INRIA
INSA-Lyon, CITI, F-69621, Villeurbanne, France
E-mail: nicolas.stouls@insa-lyon.fr

checking [LB08]. But both model-checking and test generation require models to be finite, and of tractable size. This is not usually the case with industrial applications, for which the exploration of the model executions frequently comes up against combinatorial explosion problems. Abstraction techniques allow for projecting the (possibly infinite or very large) state space of a system onto a small finite set of symbolic states. Abstract models make test generation or model-checking possible in practice [BCDG07]. In [BBJM10], we have proposed and experimented with an approach of test generation from abstract models. It appeared that the computation of the abstraction could be very time expensive, as evidenced by the Demoney [MM02] case study. We had replaced a problem of time for searching in a state graph with a problem of time for discharging proof obligations, as the abstractions were computed by proving enabledness and reachability conditions on symbolic states [BPS05].

In this paper, we contribute to solving this proving time problem by defining a syntactic abstraction function by model slicing that requires no proof. Inspired from program slicing techniques [Wei84], the function works by suppressing some state variables from a model. The variables to keep are chosen according to the tester's intention. In order to produce a state space that is both finite and sufficiently small, we still have to perform a semantic abstraction which is defined as a predicate abstraction. This requires that some proof obligations are discharged, but fewer than with the initial model, because it has been syntactically reduced. This approach results in a semantic pruning of the generated proof obligations as proposed in [CGS09].

Our process for generating tests using successively syntactic and semantic abstractions is sketched in Fig. 1. Given a source model and a set of abstract variables (the ones to be kept), the model is first reduced by syntactic abstraction. Then it is abstracted again, semantically, which gives the abstract model. Symbolic tests are extracted from it according to some selection criteria. For the tests to have the same abstraction level as the source model, they finally are instantiated on it.

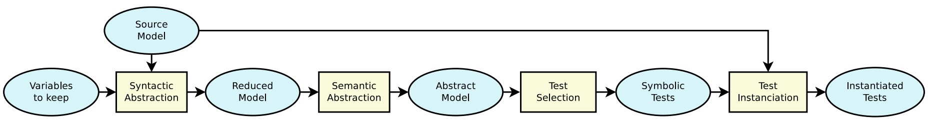

Fig. 1 Overview of the Process for Generating Tests by Abstraction

In Sec. 2, we introduce the notion of B event system, some of the main properties of the substitution computation and the predicate abstraction method. Section 3 presents two small examples that illustrate our approach, an electrical system and an elevator. In Sec. 4, we define the set of variables to be preserved by the abstraction function. The abstraction function itself is defined in Sec. 5. We prove that with this function the generated abstract model A simulates or bisimulates the initial model M. Consequently, the abstraction can be used to verify safety properties and to generate tests. In Sec. 6, we present an end to end process that computes test cases according to a set of observed variables, by using both the syntactic and semantic abstractions. In Sec. 7, we compare this process to a completely semantic one on several examples, and we evaluate the practical interest for the generation of test cases. Section 8 compares our approach to other syntactic and semantic abstraction methods. Section 9 concludes the paper and gives some future research directions.

2 Background

2.1 B Event Systems and Refinement

We use the B notation [Abr96b] to describe our models: this section gives the background required for reading the paper. Let us first define the following B notions: primitive forms of substitution, substitution properties and refinement. Then we will summarize the principles of before-after predicates, and conjunctive form (CF) of B predicates.

First introduced by J.-R. ABRIAL [Abr96a, Abr10], a B event system defines a closed specification of a system by a set of events. In the sequel, we use the following notations: x, y, z are variables and X, Y, Z are sets of variables. Pred is the set of B predicates. $I \in \mathsf{Pred}$ is an invariant and P, P_1 and P_2 ($\in \mathsf{Pred}$) denote other predicates. The modifications of the variables, i.e. the instructions, are called *substitutions* in B, following [Hoa69] where the semantics of an assignment is defined as a substitution. In B, substitutions are *generalized*: they are the semantics of every kind of atomic action. We use S, S_1 and S_2 to denote B generalized substitutions, and E and F to denote B expressions. The B events are defined as generalized substitutions. All the substitutions allowed in B event systems can be rewritten by means of the five B primitive forms of substitutions of Def. 1. The multiple assignment can be generalized to n variables. It is commutative, i.e. $x, y := E, F \;\equiv\; y, x := F, E$.

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

- single and multiple assignments, denoted by $x := E$ and $x, y := E, F$,
- substitution with no effect, denoted by SKIP,
- guarded substitution, denoted by $P \implies S$,
- bounded nondeterministic choice, denoted by $S_1 \;[]\; S_2$,
- substitution with a local variable z, denoted by $@z \cdot S$.

The substitution with a local variable is mainly used for expressing the unbounded nondeterministic choice denoted by $@z \cdot (P \implies S)$. With these primitive substitutions, some usual structures of specification languages can be defined. For instance, the conditional substitution IF P THEN S_1 ELSE S_2 END is denoted by $(P \implies S_1) \;[]\; (\neg P \implies S_2)$ with the primitive forms. Moreover, the parallel composition denoted by $||$ can be used to make the B models easier to read by human readers. This substitution is not primitive, since it can be defined through the following simplification rules from [Abr96b]:

$$\begin{aligned}
x := E \;||\; y := F &\;\Leftrightarrow\; x, y := E, F && (1)\\
\mathrm{SKIP} \;||\; S &\;\Leftrightarrow\; S && (2)\\
(P \implies S_1) \;||\; S_2 &\;\Leftrightarrow\; P \implies (S_1 \;||\; S_2) && (3)\\
(S_1 \;[]\; S_2) \;||\; S_3 &\;\Leftrightarrow\; (S_1 \;||\; S_3) \;[]\; (S_2 \;||\; S_3) && (4)\\
(@z \cdot S_1) \;||\; S_2 &\;\Leftrightarrow\; @z \cdot (S_1 \;||\; S_2) \qquad \text{if } z \text{ is not free in } S_2 && (5)\\
S_1 \;||\; S_2 &\;\Leftrightarrow\; S_2 \;||\; S_1 && (6)
\end{aligned}$$

Given a substitution S and a post-condition P, it is possible to compute the weakest precondition such that if it is satisfied, then P is satisfied after the execution of S. The weakest precondition is denoted by $[S]P$. $[x := E]P$ is the usual substitution of all the free occurrences of x in P by E. For the five other primitive forms, the weakest precondition is computed as indicated by Formulas (7) to (11) below, proved in [Abr96b].

$$
\begin{aligned}
[\text{SKIP}]P &\Leftrightarrow P && (7)\\
[P_1 \Longrightarrow S]P_2 &\Leftrightarrow (P_1 \Rightarrow [S]P_2) && (8)\\
[S_1 \; [] \; S_2]P &\Leftrightarrow [S_1]P \wedge [S_2]P && (9)\\
[@z \cdot S]P &\Leftrightarrow \forall z \cdot [S]P \qquad \text{if } z \text{ is not free in } P && (10)\\
[S](P_1 \wedge P_2) &\Leftrightarrow [S]P_1 \wedge [S]P_2 && (11)
\end{aligned}
$$

Definition 2 defines correct B event systems.

Definition 2 (Correct B Event System) It is a tuple $\langle D, C, PC, X, I, Init, Ev\rangle$ where:

- D is a list of sets (with enumerated or deferred[1] domains),
- C is a set of constants,
- $PC \in \mathsf{Pred}$ is a predicate defining the constants C,
- X is a set of state variables,
- $I \in \mathsf{Pred}$ is an invariant predicate over X,
- $Init$ is a substitution called *initialization*, such that the invariant holds in any initial state: $PC \Rightarrow [Init]I$,
- Ev is a set of event definitions in the shape of $ev_i \mathrel{\widehat{=}} S_i$ such that every event preserves the invariant: $PC \wedge I \Rightarrow [S_i]I$.

To refer to a part of an explicitly given model, we add the name of that model as a subscript to the associated symbol. I_{M} is for example the invariant of a model M.

Def. 3 is the definition of a B event system refinement. It describes the conditions under which a refinement is correct. A B refinement R is such that the user defines a new data model and its relationship with the data model of A by means of a gluing invariant. In R, the user redefines the events of A and possibly introduces new ones. The refinement proof demonstrates on the one hand that the effects on the variables of R produced by the events already existing in A are in conformance to their effect in A, and on the other hand that the events that are new in R refine SKIP, which means that they had no effect on the variables of A. Intuitively, the events of the refined system R may be triggerable less often than in the abstract system A.

Notice that in our context the refinement relation is used in the opposite direction: what the user gives is the refined model, from which we compute the abstract one automatically. The gluing invariant (later called I_G) is always a conjunction of equalities between the preserved variables. In this context, the events that could be considered as "new" in R are the ones that have been reduced either to SKIP or to $P \Longrightarrow$ SKIP in A. In other words, no event is new in R w.r.t. A since it appears explicitly in A.

Definition 3 (B Event System Refinement) Let A and R be two correct B event systems. Let I_G be their gluing invariant, i.e. a predicate that indicates how the values of the variables in R and A relate to each other. R refines A if:

- any initialization of R is associated to an initialization of A according to I_G: $PC_{\mathsf{A}} \wedge PC_{\mathsf{R}} \Rightarrow [Init_{\mathsf{R}}]\neg[Init_{\mathsf{A}}]\neg I_G$,
- any event $ev \mathrel{\widehat{=}} S_{\mathsf{R}}$ of R is either an event of A defined by $ev \mathrel{\widehat{=}} S_{\mathsf{A}}$ in Ev_{A} or a new event associated to $S_{\mathsf{A}} \mathrel{\widehat{=}}$ SKIP in A, that satisfies I_G: $PC_{\mathsf{A}} \wedge PC_{\mathsf{R}} \wedge I_{\mathsf{A}} \wedge I_G \Rightarrow [S_{\mathsf{R}}]\neg[S_{\mathsf{A}}]\neg I_G$.

[1] A *deferred* set is defined only by its name. Such a set is assumed to be finite and nonempty.

This paper also relies on two more definitions: the before-after predicate and the conjunctive form (CF) of a B predicate. We denote by $Prd_X(S)$ the before-after predicate of a substitution S. It defines the relation between the values of the variables of the set X before and after the substitution S. A primed variable denotes its after value. From [Abr96b], the before-after predicate is defined by:

$$Prd_X(S) \;\widehat{=}\; \neg[S]\neg(\bigwedge_{x\in X}(x=x')). \tag{12}$$

For a convenient reading of this paper, we give the induction definition of Prd_X on the primitive forms of substitutions:

$$\begin{aligned}
Prd_X(x:=E) &\;\widehat{=}\; x'=E \wedge (\bigwedge_{y\in X-\{x\}}(y=y')) && \text{if } x\in X && (13)\\
Prd_X(y:=E) &\;\widehat{=}\; \bigwedge_{x\in X}(x=x') && \text{if } y\notin X && (14)\\
Prd_X(P \Longrightarrow S) &\;\widehat{=}\; P \wedge Prd_X(S) && && (15)\\
Prd_X(S_1 \,[]\, S_2) &\;\widehat{=}\; Prd_X(S_1) \vee Prd_X(S_2) && && (16)\\
Prd_X(@z \cdot S) &\;\widehat{=}\; \exists(z,z') \cdot Prd_{X\cup\{z\}}(S) && \text{if } z\notin X && (17)
\end{aligned}$$

Definition 4 (Conjunctive Form) A B predicate $P \in \mathsf{Pred}$ is in CF when it is a conjunction $p_1 \wedge p_2 \wedge \ldots \wedge p_n$ where every p_i is a disjunction $p_i^1 \vee p_i^2 \vee \ldots \vee p_i^m$ such that any p_i^j is an elementary predicate in one of the following two forms:

– $E(Y)\ r\ F(Z)$, where $E(Y)$ and $F(Z)$ are B expressions on the sets of variables Y and Z and r is a relational operator,
– $\forall z \cdot P$ or $\exists z \cdot P$, where P is a B predicate in CF.

We will define a set of predicate transformation rules in Sec. 5. They apply to predicates that are put in CF according to Def. 4 before their transformation.

2.2 Predicate Abstraction

Predicate abstraction [GS97] is a special instance of the framework of abstract interpretation [CC92] that maps a potentially infinite state space R of a transition system onto a finite state space of a *symbolic transition system via* a set of atomic predicates $AP = \{a_1, a_2, \ldots, a_n\}$ over model (or program) variables. A state of R is a valuation of the state variables of the model. The symbolic transition system has a set of abstract states Q that contains at most 2^n states. Each state is a tuple $q = (p_1, p_2, \ldots, p_n)$ with p_i being either a_i or $\neg a_i$. We define an abstraction function $\alpha_{AP} : R \to Q$ such that $\alpha_{AP}(r)$ is an abstract state q with $r \models p_i$ for all $i \in 1..n$.

Let us now define the abstract transitions as *may-transitions*. Although this is not required for our formal presentation, this will clarify the forthcoming comparison with related work. A may-transition is such that for two abstract states q and q' and for an event ev, there exists a transition from q to q' by ev, denoted by $q \xrightarrow{ev} q'$, if and only if there exists a concrete transition $r \xrightarrow{ev} r'$ where r and r' are concrete states such that $\alpha_{AP}(r) = q$ and $\alpha_{AP}(r') = q'$. Such a transition $q \xrightarrow{ev} q'$ is computed by means of a predicate satisfiability problem. If we assume that an abstract state q is the predicate $\bigwedge_{i=1}^{n}\ p_i$ and that the event ev is defined by the substitution S, there is a transition $q \xrightarrow{ev} q'$ iff $SAT(\neg[S]\neg q' \wedge q)$.

Some algorithms, based on predicate abstraction and that compute abstractions that are over-approximations, can be found e.g. in [GS97,BMMR01]. They compute *may* abstract transitions automatically by means of a theorem prover. Predicate abstraction is used by Ball in [Bal05] to compute program abstraction for generating tests.

2.3 Syntactical Abstraction

Our work is mainly based on the initial work described in [BW05], that introduces an extension of the program slicing techniques to models. Program slicing is a technique introduced in [Wei84] which proceeds by removing parts of a program in order to focus on behaviors of specific parts of the program. The slicing method introduced in [BW05] is based on the CSP-ObjectZ integrated method and is established as a syntactical abstraction method. In order to slice a model, the technique proceeds in four steps:

1. computing the *program dependence graph*, which represents the *control flow* and *data flow* dependencies of each part of the program,
2. choosing some nodes of this graph as a *slicing criterion*,
3. backtracing the graph from the nodes of the slicing criterion in order to compute the set of relevant nodes,
4. removing all the parts of the program (graph) that have no effect on the slicing criterion (i.e. that are not relevant).

If the slicing criterion is defined as keeping only some variables of a model M, then this method will produce a model A which is an abstraction of M. In the current paper, we propose an extension of this method.

2.4 Refinement and Simulation

We now discuss about the preservation of properties through the refinement process, as it is of importance in the context of test cases generation. We need for that to briefly introduce the notion of simulation and its relationship with refinement, as we will refer to it in the forthcoming sections.

With two additional clauses: no deadlock introduction and no livelock introduction by the new events, the B refinement relation of event systems (see Def. 3) is proven in [BJK00] to be a simulation and, in [DJK03], to preserve propositional linear temporal logic properties.

In [CGP00], simulation is formally defined on transition systems whose transition relation is total, i.e. whose executions are infinite. We intuitively say that A simulates R if there is a relation $\mathscr{S}$ between the set of states of A and of R that satisfies the following two conditions:

- two states a and r related ($\mathscr{S}(a,r)$) have the same values for the variables of A,
- if $\mathscr{S}(a,r)$, for every state r' such that r' is a successor of r by an event e, there is a state a' that is a successor of a by e and $\mathscr{S}(a',r')$.

By extension, there is a bisimulation relation between A and R if A simulates R and if for all the states a, r and a' such that $\mathscr{S}(a,r)$ holds and a' is a successor of a by an event e, there is a state r' that is a successor of r by e and such that $\mathscr{S}(a',r')$.

In [CGP00], it is proven that the relation "A simulates R" is a preorder and that every ACTL* formula satisfied by A is satisfied by R. ACTL* defines temporal logic formulas that hold on all the executions (quantifier A). Intuitively, as the executions of both systems perform the same actions and that there are more executions in A than in R, it is obvious that a property that holds on A also holds on R. For a bisimulation, it is proven in [CGP00] that every CTL* formula holds in A if and only if it holds in R.

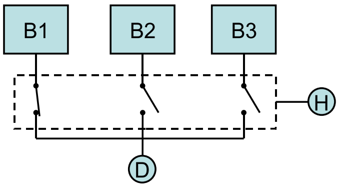

Fig. 2 Electrical System

But a B event system may be blocking, i.e. define executions that are finite, and in Def. 3, we have defined the B refinement without the two aforementioned clauses. Thus the refinement can introduce new deadlocks or new livelocks in the refined system. In such cases, the simulation conditions still hold, but the preservation theorems of [CGP00] do not apply anymore. It follows that the ACTL* properties of A are not preserved on R, but it is proven that safety properties do. Indeed, if nothing bad happens on a set of executions, then nothing bad happens either on a subset of it. In contrast, liveness and fairness properties are not preserved when some deadlocks or livelocks are introduced.

The reason why we have not added in this paper the no deadlock and no livelock clauses to Def. 3, is because our problem is not a verification one but a test generation one. Also notice that in our context, since we compute the abstraction A from the initial system R and not the contrary, there is no new livelock in R w.r.t. A since no event is new in R. In contrast, some deadlocks of R can be removed in A.

3 Examples

We introduce in this section two B event systems that we use as running examples to illustrate our propositions in this paper. The first one describes a simple electrical system by means of a small model. The second one describes an elevator by modelling its calls, its position, its direction, its doors and its light.

The electrical system generalizes the example from [JSBM10] to an infinite state space. It is simple to read and well suited for short illustrations. But we also want to exhibit some differences between three methods that we present in Sec. 4, and that requires the model to be slightly more complicated. This is the reason why we introduce the second example.

3.1 Electrical System Example

A device D is powered by $NBat$ batteries $B_1, B_2, \ldots, B_{NBat}$ as shown in Fig. 2 with $NBat = 3$. A switch connects (or not) a battery B_i to the device D. A clock H periodically sends a signal that causes a commutation of the switches, i.e. a change of the battery in charge of powering the device D. The system has to satisfy the three following requirements:

- Req_1: no short-circuit, i.e. there is only one switch closed at a time,
- Req_2: continuous power supply, i.e. there is always one switch closed,
- Req_3: a signal from the clock always changes the switch that is closed.

The batteries are subject to electrical failures. If a failure occurs on the battery that is powering D, the system triggers an exceptional commutation to satisfy the requirement Req_2. The broken batteries are replaced by a maintenance service. We assume that it works fast enough for not having more than $NBat-1$ batteries down at the same time. When $NBat-1$ batteries are down, the requirement Req_3 is relaxed and the clock signal leaves unchanged the switch that is closed.

This system is modeled in Fig. 3 by means of three variables. H models the clock and takes two values: *tic* when it asks for a commutation and *tac* when this commutation has occurred. Sw models the state of the switches by an integer between 1 and $NBat$: $Sw=i$ indicates that the switch i is closed while the others are opened. This modelling makes that requirements Req_1 and Req_2 necessarily hold. Bat models the electrical failures by a total function. The *ko* value for a battery indicates that it is down. In addition to the typing of the variables, the invariant I expresses the assumption that at least one battery is not down by stating that $Bat(Sw)=ok$. Notice that the requirement Req_3 is a dynamic property, not formalized in I. The initial state is defined by *Init* in Fig. 3. The behavior of the system is described by means of four events:

- Tic sends a commutation request,
- *Com* performs a commutation (i.e. changes the closed switch),
- *Fail* simulates an electrical failure on one of the batteries,
- *Rep* simulates a maintenance intervention replacing a down battery.

In this model, we use the expression $r \rhd E$ which denotes a relation where the range is restricted by the set E. For example: $\{1 \mapsto ok,\ 2 \mapsto ko,\ 3 \mapsto ok\} \rhd \{ok\} = \{1 \mapsto ok,\ 3 \mapsto ok\}$.

$$
\begin{array}{lcl}
C & \widehat{=} & \{NBat\} \\
PC & \widehat{=} & NBat \in \mathbb{N}_1 \\
X & \widehat{=} & \{H,\ Sw,\ Bat\} \\
I & \widehat{=} & H \in \{tic,\ tac\} \ \wedge\ Sw \in 1..NBat \ \wedge\ (Bat \in 1..NBat \to \{ok,\ ko\}) \ \wedge\ Bat(Sw) = ok \\
Init & \widehat{=} & H := tac \ \|\ Sw := 1 \ \|\ Bat := (1..NBat) \times \{ok\} \\
Tic & \widehat{=} & H = tac \implies H := tic \\
Com & \widehat{=} & H = tic \implies @ns.((ns \in 1..NBat \ \wedge\ Bat(ns) = ok \ \wedge\ ns \neq Sw) \implies H := tac \ \|\ Sw := ns) \\
Fail & \widehat{=} & \mathsf{card}(Bat \rhd \{ok\}) > 1 \implies \\
 & & \quad @nb.((nb \in 1..NBat \ \wedge\ Bat(nb) = ok) \implies \\
 & & \qquad Bat(nb) := ko \ \| \\
 & & \qquad \mathsf{IF}\ nb = Sw\ \mathsf{THEN}\ @ns.((ns \in 1..NBat \ \wedge\ ns \neq Sw \ \wedge\ Bat(ns) = ok) \implies Sw := ns)\ \mathsf{END}) \\
Rep & \widehat{=} & @nb.((nb \in 1..NBat \ \wedge\ Bat(nb) = ko) \implies Bat(nb) := ok)
\end{array}
$$

Fig. 3 B Specification of the Electrical System

3.2 Elevator Case Study

The event B model in Fig. 4 describes an elevator w.r.t. five parameters: its position (*position*), the set of floors from which it can be called (*Calls*), its movement (*status* and *direction*), the floor, if any, where its doors are open (*Doors*) and the state of the light in the lift cage (*light*).

The elevator serves the floors between *minFloor* and *maxFloor*, as modelled by FLOORS, its set of floors. Thus its current *position* is restricted to FLOORS. Its *direction* is either *up* or *down*, and its *status* can be: *movement*, *stop* or *standby*. When the elevator is in standby, the *light* is *off*. When it is stopped, the doors (*Doors*) are either closed ($Doors=\varnothing$) or open ($Doors=\{position\}$).

Four types of events can occur in this model:

– the elevator can be called from another floor (*call*),
– the doors can be opened or closed (*open*, *close*),
– the elevator can move (*move*),
– the elevator can go into standby or be woken up (*sleepdown*, *wakeup*).

$D \;\widehat{=}\; \mathsf{MODE} = \{movement, stop, standby\}\ ;\ \mathsf{MOVEMENT} = \{up, down\}\ ;\ \mathsf{ONOFF} = \{on, off\}$

$C \;\widehat{=}\; \{minFloor, maxFloor, \mathsf{FLOORS}\}$
$P \;\widehat{=}\; maxFloor \in \mathbb{Z} \wedge minFloor \in \mathbb{Z} \wedge minFloor < maxFloor \wedge \mathsf{FLOORS} = minFloor..maxFloor$

$X \;\widehat{=}\; \{position, Calls, status, Doors, direction, light\}$
$I \;\widehat{=}\; position \in \mathsf{FLOORS} \wedge Calls \subseteq \mathsf{FLOORS} \wedge status \in \mathsf{MODE} \wedge Doors \subseteq \mathsf{FLOORS} \wedge$
$\quad direction \in \mathsf{MOVEMENT} \wedge light \in \mathsf{ONOFF} \wedge$
$\quad ((Doors \neq \varnothing) \Rightarrow (Doors = \{position\} \wedge status = stop)) \wedge$
$\quad (status = stop \Rightarrow position \notin Calls) \wedge$
$\quad ((light = off) \Leftrightarrow (status = standby)) \wedge$
$\quad (status = standby \Rightarrow Doors = \varnothing)$

$Init \;\widehat{=}\; position := minFloor \parallel Calls := \varnothing \parallel status := standby \parallel Doors := \varnothing \parallel direction := up \parallel light := off$

$call \;\widehat{=}\; @fl \cdot (fl \in \mathsf{FLOORS} \wedge fl \neq position \implies Calls := Calls \cup \{fl\})$

$open \;\widehat{=}\; Doors = \varnothing \wedge status = stop \implies Doors := \{position\}$

$close \;\widehat{=}\; Doors \neq \varnothing \implies Doors := \varnothing$

$move \;\widehat{=}\; Doors = \varnothing \wedge Calls \neq \varnothing \wedge status \neq standby \implies$

```
    IF position ∈ Calls THEN
        status := stop || Calls := Calls − {position}
    ELSE
        status := movement ||
        IF direction = up THEN
            IF (Calls ∩ (position..maxFloor)) = ∅ THEN
                position := position − 1 || direction := down
            ELSE
                position := position + 1
            END
        ELSE
            IF (Calls ∩ (minFloor..position)) = ∅ THEN
                position := position + 1 || direction := up
            ELSE
                position := position − 1
            END
        END
    END
```

$sleepdown \;\widehat{=}\; Doors = \varnothing \wedge status = stop \wedge Calls = \varnothing \implies status := standby \parallel light := off$

$wakeup \;\widehat{=}\; status = standby \wedge Calls \neq \varnothing \implies$

```
    status := stop || light := on ||
    IF position ∈ Calls THEN Calls := Calls − {position} END
```

Fig. 4 B Specification of the Elevator

4 Choice of the Variables for the Syntactical Abstraction

Our aim is to produce an abstract model A of a model M by observing only a subset X_{A} of the state variables X_{M} of M. For instance, to test the electrical system in the particular case where there is only one battery left working, it is sufficient to observe only the variable *Bat*. However, for preserving the behaviors of M related to the variables of X_{A}, the variables used either to assign the observed variables or to define the conditions under which they are assigned also have to be kept in A.

The slicing technique that we present in this paper uses as a slicing criterion a set of variables that we denote as *observed variables*. We use a two steps method: (i) computing the set of variables to be kept according to the slicing criterion, (ii) slicing the model according to this computed set of variables. We present the first step in the current section, while the second step will be described in Sec. 5.

We first describe in this section the principle of choosing a set of variables to be kept in an abstraction, then we propose three methods that compute a set of *abstract variables* according to a set of *observed variables*, and we finally compare these three methods.

4.1 Principle

As proposed in [BW05], we make a distinction between the observed variables and the abstract ones. A set X_{A} of *abstract variables* is the union of a set of *observed variables* with a set of *relevant variables*. In the context of test generation, the observed variables are the ones used to describe a test purpose, while the relevant variables are the ones used to describe the evolutions of the observed variables. More precisely, the possible relevant variables are the ones used to assign an observed variable (*data-flow dependence*), augmented with the variables used to express when such an assignment occurs (*control-flow dependence*).

A naive method to compute X_{A} is to syntactically collect all the variables that are either on the right side or in the guard of the assignments of an observed variable. But this method will in most cases collect a very large amount of variables, mainly because of the guard. For instance, in $(y \Longrightarrow x,z := E_1,E_2)[](\neg y \Longrightarrow x := E_3)$, if x is the observed variable, then y is not relevant if y occurs neither in E_1 nor in E_3. A similar weakness goes for the unbounded non-deterministic choice $@z \cdot (P \Longrightarrow S)$. Moreover, since we want to facilitate the computation and minimize its time, we must keep all the variables assigned to an observed variable. We cannot abstract such assignments with non deterministic choices as it would require to perform a complex type induction in order to characterize the definition domain of the abstracted expressions. Consequently, we need to achieve the computation of each set of abstract variables by means of a fix-point calculus.

Hence our contribution consists of three methods for identifying the relevant variables. The first one only considers the data-flow (DF) dependence. It is efficient but may select a set too small of relevant variables, resulting in a model with too many behaviors in the abstracted model. The second one uses both data and control flow (CF) dependencies, and produces abstract models that have the same set of behaviors as the original model w.r.t. the abstract variables. But this second method may compute a set with too many relevant variables, because a predicate simplification would be required to restrict the size of X_{A}, and predicate simplification is not a decidable problem. Hence we propose a third method that is a mix between the first two ones, and provides an interesting trade-off.

4.2 Proposition 1: Data-Flow Dependence Only

The first method considers as relevant only the variables that appear on the right side of an assignment symbol to an abstract variable. Starting from the set of observed variables, the set of all abstract variables is computed as the least fix-point when adding the relevant variables. For instance, the set of relevant variables of the electrical system is empty if the set of observed variables is $\{Bat\}$ (see Fig. 3). Hence if a test purpose is only based on *Bat*, then $X_{\mathsf{A}} = \{Bat\}$. A drawback of this method is that it can introduce in A new execution traces w.r.t. M. Indeed, it may weaken the guards of some of the events, that would thus become enabled more often.

4.3 Proposition 2: Data-Flow and Control-Flow Dependencies

The second method first computes a predicate that characterizes a condition under which an abstract variable is modified, then simplifies it, and finally considers all its free variables as relevant. We express by means of Formula (18) the modifications really performed by a substitution S on a set X_{A}:

$$Mod_{X_{\mathsf{A}}}(S) \;\widehat{=}\; Prd_{X_{\mathsf{A}}}(S) \wedge (\bigvee_{x \in X_{\mathsf{A}}} x \neq x'). \tag{18}$$

Our intention is that the predicate, that defines the condition under which an abstract variable is modified, only involves the variables really required to modify it. Hence primed variables are not quantified, but are allowed to be free. For instance, consider $X_{\mathsf{A}} = \{x\}$ and the substitution $x := y [] (z > 0 \Longrightarrow x := w) [] v := 3$. The predicate has to be in the shape of $(x' = y \vee (z > 0 \wedge x' = w)) \wedge x \neq x'$, where the variables y, w and z are relevant whereas v is not (see Fig. 5).

$$\begin{array}{lll}
& Mod_{\{x\}}(x := y \;[]\; (z > 0 \implies x := w) \;[]\; v := 3) & \\
\Leftrightarrow & Prd_{\{x\}}(x := y \;[]\; (z > 0 \implies x := w) \;[]\; v := 3) \wedge x \neq x' & \text{– applying (18)} \\
\Leftrightarrow & (Prd_{\{x\}}(x := y) \vee Prd_{\{x\}}((z > 0 \implies x := w)) \vee Prd_{\{x\}}(v := 3)) \wedge x \neq x' & \text{– applying (16)} \\
\Leftrightarrow & (x' = y \vee (z > 0 \wedge Prd_{\{x\}}(x := w)) \vee (x = x')) \wedge x \neq x' & \text{– applying (13, 14, 15)} \\
\Leftrightarrow & (x' = y \vee (z > 0 \wedge x' = w) \vee (x = x')) \wedge x \neq x' & \text{– applying (13)} \\
\Leftrightarrow & (x' = y \vee (z > 0 \wedge x' = w)) \wedge x \neq x' & \text{– by simplification}
\end{array}$$

Since v is not free in this predicate, v is not relevant for x in $x := y \;[]\; (z > 0 \implies x := w) \;[]\; v := 3$.

Fig. 5 Example of a Mod_X Computation

The Mod_X predicate can also be defined by induction through primitive substitutions, as proposed in Table 1. This second formalization is more suited to an automated simplification. Intuitively, an assignment $x := E$ is associated to $false$ if and only if either x is not in X or x already has the same value as E. The other assignment cases are just generalizations. This implements the data-flow dependence. For the control-flow dependence, a non-deterministic choice is a union between control-flow branches, thus a disjunction between predicates. A guarded substitution $P \Longrightarrow S$ is associated to the whole condition P augmented with the result of the analysis of S. Once expressed, this predicate needs to be logically simplified.

Substitution		Modification Predicate	Condition
$Mod_X(x := E)$	$\widehat{=}$	$false$	$x \notin X$
$Mod_X(x := E)$	$\widehat{=}$	$x' = E \wedge \bigwedge_{z \in X - \{x\}}(z' = z) \wedge x \neq x'$	$x \in X$
$Mod_X(x, y := E, F)$	$\widehat{=}$	$false$	$x \notin X \wedge y \notin X$
$Mod_X(x, y := E, F)$	$\widehat{=}$	$x' = E \wedge \bigwedge_{z \in X - \{x\}}(z' = z) \wedge x \neq x'$	$x \in X \wedge y \notin X$
$Mod_X(x, y := E, F)$	$\widehat{=}$	$x' = E \wedge y' = F \wedge \bigwedge_{z \in X - \{x,\, y\}}(z' = z) \wedge \bigvee_{z \in \{x,\, y\}}(z \neq z')$	$x \in X \wedge y \in X$
$Mod_X(\text{SKIP})$	$\widehat{=}$	$false$	
$Mod_X(P \implies S)$	$\widehat{=}$	$P \wedge Mod_X(S)$	
$Mod_X(S_1 \;[]\; S_2)$	$\widehat{=}$	$Mod_X(S_1) \vee Mod_X(S_2)$	
$Mod_X(@z \cdot S)$	$\widehat{=}$	$\exists(z,\, z') \cdot Mod_{X \cup \{z\}}(S)$	

Table 1 $Mod_X(S)$ Predicate Defined through Primitive Substitutions

Property 1 $Mod_X(S)$ as defined in Table 1 satisfies the definition of Formula (18).

Proof (of property 1) For any case of primitive substitution S, we prove that $Mod_X(S)$ as defined by Formula (18) is equal to its value in Table 1. We replace for that $Prd_X(S)$ by its definition given in Formula (12) and we transform it according to the Formulas (7) to (10). □

Finally, X_A is computed as a least fix-point, by iteratively incrementing for each event the initial set of observed variables with the relevant variables. This process necessarily terminates since the set of variables is finite and growing. For instance, $Mod_{\{Bat\}}$ gives an empty set of relevant variables when applied to the electrical system example, as shown in Fig. 6, while $Mod_{\{H\}}$ gives $X_\mathsf{A} = \{Bat, H\}$.

$$
\begin{array}{lcl}
Mod_{\{Bat\}}(Init) & \Leftrightarrow & Bat' = (1..NBat) \times \{ok\} \\
Mod_{\{Bat\}}(Tic) & \Leftrightarrow & false \text{ (no assignment of } Bat) \\
Mod_{\{Bat\}}(Com) & \Leftrightarrow & false \text{ (no assignment of } Bat) \\
Mod_{\{Bat\}}(Fail) & \Leftrightarrow & \mathrm{card}(Bat \rhd \{ok\}) > 1 \Longrightarrow \exists nb \cdot (nb \in 1..NBat \wedge Bat(nb) = ok \wedge Bat'(nb) = ko) \\
Mod_{\{Bat\}}(Rep) & \Leftrightarrow & \exists nb \cdot (nb \in 1..NBat \wedge Bat(nb) = ko \wedge Bat'(nb) = ok)
\end{array}
$$

Fig. 6 $Mod_{\{Bat\}}$ Computation Applied to the Power System Example

The $Mod_X(S)$ predicate aims at computing a set of abstract variables for syntactically abstracting a model. But applying the rules of Table 1 to compute Mod_X will in most cases require the use of a constraint solver. Since the computation time of such a tool is comparable to the one of an automatic theorem prover, the gain w.r.t. the computation of Formula (18) is not obvious. However, some "easy simplifications" can be performed, that require no computation. This is the case for example for the first and third rules of Table 1. Additionally, it is possible to make use of information that appear in constructions such as the IF or the SWITCH structures, that the B language offer as syntactic sugar. See for example the IF rules that we propose in Fig. 7. Hence, at least in the the first case, IF structures can be syntactically simplified. This is why we claim that the computation of Mod_X can be performed syntactically, which makes it light to use in practice.

Substitution		**Modification Predicate**	**Condition**
$Mod_X(\text{IF } C \text{ THEN } S_1 \text{ ELSE } S_2 \text{ THEN})$	$\widehat{=}$	$Mod_X(S_1) \vee Mod_X(S_2)$	$free(Mod_X(S_1)) = free(Mod_X(S_2))$
$Mod_X(\text{IF } C \text{ THEN } S_1 \text{ ELSE } S_2 \text{ THEN})$	$\widehat{=}$	$Mod_X((C \Longrightarrow S_1) \, [] \, (\neg C \Longrightarrow S_2))$	$free(Mod_X(S_1)) \neq free(Mod_X(S_2))$

Fig. 7 $Mod_{\{Bat\}}$ Computation Applied to IF Substitution

4.4 Proposition 3: Data-Flow and Partial Control-Flow Dependencies

Intuitively, the most relevant variables to describe the evolutions of the observed variables are the ones on which the observed variables directly depend, through control flow or data flow. They are computed by the first iteration of the fix-point calculus. The variables computed by the second iteration are less relevant. So are the variables added by further iterations, that become less and less relevant. Hence we propose to mix the first two propositions, in order to have as much as possible strongly relevant variables and as less as possible weakly relevant variables. Our third proposition is:

– first, use Mod_X to characterize the set R_1 of variables directly relevant to the observed variables,
– then compute X_A as a fix-point w.r.t. DF dependence only, starting with $X_\mathsf{A} \cup R_1$.

4.5 Comparison Between the Three Propositions on the Elevator Example

Figure 8 illustrates the differences between the three propositions by describing all the variables dependencies on the Elevator example, according to each of the propositions. Table A in Fig. 8 gives, for each event and each variable, the set of relevant variables by using either Proposition 1 or Proposition 2. It has been computed by means of a single pass on the B model, i.e. without fix-point. The rows where both the propositions returned no relevant variable have been removed. Table B shows the results of the fix-point computations, according to each of the different propositions. The results are given for the system as a whole, i.e. not event by event, since the fix-point computation involves all the events.

A. For each event, without fix-point

Event	**Observed variables**	**Relevant var. w.r.t. Prop. 1**	**Relevant var. w.r.t. Prop. 2**
call	*Calls*	∅	$\{position\}$
open	*Doors*	$\{position\}$	$\{position, status\}$
move	*position*	∅	$\{Calls, status, Doors\}$
	Calls	$\{position\}$	$\{position, status, Doors\}$
	status	∅	$\{position, Calls, Doors\}$
	direction	∅	$\{position, Calls, status, Doors\}$
sleepdown	*status*	∅	$\{Calls, Doors\}$
	light	∅	$\{Calls, Doors, status\}$
wakeup	*Calls*	$\{position\}$	$\{position, status\}$
	status	∅	$\{Calls\}$
	light	∅	$\{Calls, status\}$

B. For the whole system, with fix-point

	Observed variables	**Relevant var. w.r.t. Prop. 1**	**Relevant var. w.r.t. Prop. 2**	**Relevant var. w.r.t. Prop. 3**
	position	∅	$\{Calls, status, Doors\}$	$\{Calls, status, Doors\}$
	Calls	$\{position\}$	$\{position, status, Doors\}$	$\{position, status, Doors\}$
	status	∅	$\{position, Calls, Doors\}$	$\{position, Calls, Doors\}$
	Doors	$\{position\}$	$\{position, Calls, status\}$	$\{position, status\}$
	direction	∅	$\{position, Calls, status, Doors\}$	$\{position, Calls, status, Doors\}$
	light	∅	$\{position, Calls, status, Doors\}$	$\{position, Calls, status, Doors\}$

Fig. 8 Variables Dependencies in the Elevator System

Let $\{Doors\}$ be for example the set of observed variables. Table B in Fig. 8 indicates that the set of abstract variables is:

– $X_\mathsf{A} = \{position\}$ with Proposition 1,
– $X_\mathsf{A} = \{position, Calls, status\}$ with Proposition 2,
– $X_\mathsf{A} = \{position, status\}$ with Proposition 3.

Hence the set of abstract variables, on which depends the size and the precision of the abstraction, can be finely controlled by the choice of the method to compute the abstract variables.

$$
\begin{array}{rcll}
T_X(E(Y)\; r\; E(Z)) & \widehat{=} & E(Y)\; r\; E(Z) & \text{if } Y \subseteq X \text{ and } Z \subseteq X \quad (R_1)\\
T_X(E(Y)\; r\; E(Z)) & \widehat{=} & \mathit{true} & \text{if } Y \not\subseteq X \text{ or } Z \not\subseteq X \quad (R_2)\\
T_X(P_1 \vee P_2) & \widehat{=} & T_X(P_1) \vee T_X(P_2) & (R_3)\\
T_X(P_1 \wedge P_2) & \widehat{=} & T_X(P_1) \wedge T_X(P_2) & (R_4)\\
T_X(\alpha z \cdot P) & \widehat{=} & \alpha z \cdot T_{X \cup \{z\}}(P) & (R_5)
\end{array}
$$

Fig. 9 CF Predicate Slicing Rules

$$
\begin{array}{ll}
T_{\{Bat\}}(H \in \{tic,\ tac\} \wedge Sw \in 1..NBat \wedge Bat \in 1..NBat \rightarrow \{ok,\ ko\} \wedge Bat(Sw) = ok) & \\
= \begin{array}{l} T_{\{Bat\}}(H \in \{tic,\ tac\}) \wedge T_{\{Bat\}}(Sw \in 1..NBat) \\ \wedge\ T_{\{Bat\}}(Bat \in 1..NBat \rightarrow \{ok,\ ko\}) \wedge T_{\{Bat\}}(Bat(Sw) = ok) \end{array} & \text{–applying } (R_4)\\
= Bat \in 1..NBat \rightarrow \{ok,\ ko\} & \text{–applying } (R_1) \text{ and } (R_2)
\end{array}
$$

Fig. 10 Example of Predicate Slicing

5 B Event Model Slicing

This section introduces an abstraction method of B models using a set of abstract variables as slicing criterion. Similar rules could be adapted for more generic formalisms such as pre-post models or symbolic transition systems. We first define slicing functions for the predicates and the substitutions w.r.t. a set of abstract variables. We then define the abstraction of a B event model M as the abstraction of its clauses, and we establish some properties of simulation and bisimulation between the computed abstract model and M, according to the method used to select the abstract variables (see Sec. 4).

5.1 Predicate Slicing

Once the set of abstract variables $X_{\mathsf{A}} (\subseteq X_{\mathsf{M}})$ is defined, we have to describe how to abstract a model according to X_{A}. We first define the slicing function $T_{X_{\mathsf{A}}}(P)$ that abstracts a predicate P according to X_{A}. We define T_X on predicates in the conjunctive form (see Def. 4) by induction with the rules given in Fig. 9.

An elementary predicate is left unchanged when all the variables used in the predicate are considered in the abstraction (see the rule R_1). Otherwise, when an expression depends on some variables not kept in the abstraction, the truth value of an elementary predicate is undetermined (see the rule R_2). As we want to weaken the predicate, we replace an undetermined elementary predicate by *true*. Consequently, a predicate $P_1 \wedge P_2$ is transformed into P_1 when P_2 is undetermined, and a predicate $P_1 \vee P_2$ is transformed into *true* when P_1 or P_2 is undetermined (see the rules R_3 and R_4). Finally, the slicing of a quantified predicate is the slicing of its body w.r.t. the abstract variables, augmented with the quantified variable (see the rule R_5).

For example the invariant I of the electrical system is transformed, according to the single variable *Bat*, into $T_{\{Bat\}}(I) = Bat \in 1..NBat \rightarrow \{ok,\ ko\}$ as in Fig. 10.

Property 2 Let P be a CF predicate in Pred and let X be a set of variables. $P \Rightarrow T_X(P)$ is valid.

Proof (of property 2) As aforementioned, $T_X(P)$ is weaker than P. Indeed, for any predicate P in CF there exist p_1 and p_2 such that $P = p_1 \wedge p_2$ and such that it is transformed either into $p_1 \wedge p_2$, or into p_1, or into p_2, or into *true*, by application of the slicing rules R_i. For any disjunctive predicate P there exist p_1 and p_2 such that $P = p_1 \vee p_2$ and $p_1 \vee p_2$ is transformed either into $p_1 \vee p_2$ or into *true*. □

$$
\begin{array}{rlrr}
T_X(x := E) \mathrel{\widehat{=}} & \text{SKIP} & \text{if } x \notin X & (R_6)\\
T_X(x := E) \mathrel{\widehat{=}} & x := E & \text{if } x \in X & (R_7)\\
T_X(\text{SKIP}) \mathrel{\widehat{=}} & \text{SKIP} & & (R_8)\\
T_X(x, y := E, F) \mathrel{\widehat{=}} & \text{SKIP} & \text{if } x \notin X \text{ and } y \notin X & (R_9)\\
T_X(x, y := E, F) \mathrel{\widehat{=}} & x := E & \text{if } x \in X \text{ and } y \notin X & (R_{10})\\
T_X(x, y := E, F) \mathrel{\widehat{=}} & x, y := E, F & \text{if } x \in X \text{ and } y \in X & (R_{11})\\
T_X(P \implies S) \mathrel{\widehat{=}} & T_X(P) \implies T_X(S) & & (R_{12})\\
T_X(S_1 \mathbin{[]} S_2) \mathrel{\widehat{=}} & T_X(S_1) \mathbin{[]} T_X(S_2) & & (R_{13})\\
T_X(@z \cdot S) \mathrel{\widehat{=}} & @z \cdot T_{X \cup \{z\}}(S) & & (R_{14})
\end{array}
$$

Fig. 11 Primitive Substitution Slicing Rules

5.2 Substitution Slicing

The abstraction of substitutions is defined through cases in Fig. 11 on the primitive forms of substitutions. Intuitively, any assignment $x := E$ is preserved into the sliced model if and only if x is an abstract variable. According to any of the three methods described in sec. 4.1, if x is an abstract variable, then so are all the variables in E. Therefore, in rules R_6 to R_{11}, we do not transform the expressions E and F.

A substitution is abstracted by SKIP when it does not modify any variable from X (see rules R_6, R_8, R_9 and R_{10} in which $y := F$ is abstracted by SKIP). The assignment of a variable x is left unchanged if x is an abstract variable (see rules R_7, R_{10}, R_{11}). The slicing of a guarded substitution S is such that $T_X(S)$ is enabled at least as often as S, since $T_X(P)$ is weaker than P from Prop. 2 (see rule R_{12}). The bounded non deterministic choice $S_1 \mathbin{[]} S_2$ becomes a bounded non deterministic choice between the abstraction of S_1 and the one of S_2 (see rule R_{13}). The quantified substitution is sliced by inserting the bound variable into the set of abstract variables (see rule R_{14}).

Notice that a conditional substitution defined by a non deterministic choice between two exclusive guarded substitutions ($P \Longrightarrow S_1 \mathbin{[]} \neg P \Longrightarrow S_2$) can be transformed into an actual non deterministic choice, since $T_X(P)$ and $T_X(\neg P)$ can respectively become weaker than P and $\neg P$. For example, $T_{\{x,y\}}(x = y \wedge z > x \Longrightarrow x := 3 \mathbin{[]} x \neq y \vee z \leq x \Longrightarrow x := 4)$ is equal to $(x = y \Longrightarrow x := 3 \mathbin{[]} TRUE \Longrightarrow x := 4)$.

5.3 Model Slicing

According to the predicate and substitution slicing functions (see Fig. 9 and Fig. 11), we define the slicing of a B event model according to a set of abstract variables (see Sec. 4.1) in Def. 5. It translates a correct model M into a model A that simulates M (see Sec. 5.4).

Definition 5 (B Event System Slicing) Let X_{A} be a set of abstract variables, defined as in Sec. 4.1 from a set of observed variables X with $X \subseteq X_{\mathsf{M}}$. A correct B event system $\mathsf{M} = \langle D_{\mathsf{M}}, C_{\mathsf{M}}, PC_{\mathsf{M}}, X_{\mathsf{M}}, I_{\mathsf{M}}, Init_{\mathsf{M}}, Ev_{\mathsf{M}} \rangle$ is abstracted as the B event system
$\mathsf{A} = \langle D_{\mathsf{M}}, C_{\mathsf{M}}, PC_{\mathsf{M}}, X_{\mathsf{A}}, I_{\mathsf{A}}, Init_{\mathsf{A}}, Ev_{\mathsf{A}} \rangle$ as follows:

- $X_{\mathsf{A}} \subseteq X_{\mathsf{M}}$, the set of abstract variables is a subset of the state variables,
- $I_{\mathsf{A}} = T_{X_{\mathsf{A}}}(I_{\mathsf{M}})$, the invariant is sliced,
- $Init_{\mathsf{A}} = T_{X_{\mathsf{A}}}(Init_{\mathsf{M}})$, the initialization is sliced,
- to each event $ev \mathrel{\widehat{=}} S_{\mathsf{M}}$ in Ev_{M} is associated the sliced event $ev \mathrel{\widehat{=}} T_{X_{\mathsf{A}}}(S_{\mathsf{M}})$ in Ev_{A}.

$$
\begin{array}{lll}
C & \widehat{=} & \{\mathsf{NBat}\} \\
PC & \widehat{=} & \mathsf{NBat} \in \mathbb{N}_1 \\
X & \widehat{=} & \{\mathsf{Bat}\} \\
I & \widehat{=} & \mathsf{Bat} \in 1..\mathsf{NBat} \to \{ok, ko\} \\
Init & \widehat{=} & \mathsf{Bat} := (1..\mathsf{NBat}) \times \{ok\} \\
Tic & \widehat{=} & \textsc{skip} \\
Com & \widehat{=} & @ns.(ns \in 1..\mathsf{NBat} \ \wedge \ \mathsf{Bat}(ns) = ok \implies \textsc{skip}) \\
Fail & \widehat{=} & \mathrm{card}(\mathsf{Bat} \rhd \{ok\}) > 1 \implies \\
 & & \quad @nb.(nb \in 1..\mathsf{NBat} \ \wedge \ \mathsf{Bat}(nb) = ok \implies \mathsf{Bat}(nb) := ko) \\
Rep & \widehat{=} & @nb.(nb \in 1..\mathsf{NBat} \ \wedge \ \mathsf{Bat}(nb) = ko \implies \mathsf{Bat}(nb) := ok)
\end{array}
$$

Fig. 12 B Variable Slicing of the Electrical System

In Def. 5, the sets of sets (D), constants (C) and properties (PC) are kept unchanged in the abstraction. Indeed these clauses are not in the right part of proof obligations of formulas from Def. 2. Hence, slicing these clauses reduces neither the number, nor the complexity of the generated proof obligations.

By applying Def. 5, the electrical system is transformed as shown in Fig. 12 for the set of abstract variables $\{Bat\}$.

5.4 Properties of the Generated Abstractions

In this section, we discuss the preservation of properties by the various abstractions that we produce, as well as the instanciability of the tests generated from them. We distinguish between Proposition 2 and Propositions 1 and 3.

5.4.1 Proposition 2

When the set of abstract variables X_{A} preserves both the data and control flows as defined in Sec. 4.3 (Proposition 2), the transition relation, projected on X_{A}, is preserved, as established by Theorem 1. In other words, A and M are bisimilar, since they have an equivalent before-after relation modulo X_{A} ($Prd_{X_{\mathsf{A}}}$). Hence when a CTL* property is verified on A it holds on M and the test cases generated from A can always be instantiated on M.

Theorem 1 *Let S be a substitution. Let X be a set of abstract variables composed of any free variable of $Mod_X(S)$. We have $Prd_X(S) \Leftrightarrow Prd_X(T_X(S))$.*

Proof (of theorem 1) We are in the case of Proposition 2 as defined in Sec. 4.1. We prove that the following formula holds: $Prd_X(S) \Leftrightarrow Prd_X(T_X(S))$.

Since $Prd_X(S) \widehat{=} \neg[S]\neg \bigwedge_{x\in X} x = x'$ and $Prd_X(T_X(S)) \widehat{=} \neg[T_X(S)]\neg \bigwedge_{x\in X} x = x'$ (see Formula (12) in Sec. 2), we verify it by induction through primitive substitutions by proving that $[S]P \Leftrightarrow [T_X(S)]P$ holds when P is defined only in terms of abstract variables in X (as in Prd_X definition). Let $[T_X(S)]P \ \Leftrightarrow \ [S]P$ be the induction hypothesis. A proof by induction on primitive substitutions that $[T_X(S)]P \ \Leftrightarrow \ [S]P$ holds is the following:

$[\mathbf{T_X(S)}]\mathbf{P} \Leftrightarrow [\mathbf{S}]\mathbf{P}$	**Condition or justification**
$[\text{SKIP}]P \Leftrightarrow [y := E]P \Leftrightarrow P$	if $y \notin X$
$[x := E]P \Leftrightarrow [x := E]P$	if $x \in X$
$[\text{SKIP}]P \Leftrightarrow [\text{SKIP}]P \Leftrightarrow P$	
$[\text{SKIP}]P \Leftrightarrow [z, y := E, F]P \Leftrightarrow P$	if $z \notin X$ and $y \notin X$
$[x := E]P \Leftrightarrow [x, y := E, F]P$	if $x \in X$ and $y \notin X$
$[x := F]P \Leftrightarrow [y, x := E, F]P$	if $y \notin X$ and $x \in X$
$[x_1, x_2 := E, F]P \Leftrightarrow [x_1, x_2 := E, F]P$	if $x_1 \in X$ and $x_2 \in X$
$T_X(P_1) \Rightarrow [T_X(S)]P \Leftrightarrow [P_1 \Longrightarrow S]P$	by Formula (8), induction hypothesis and since $T_X(P_1) = P_1$ according to $Mod_X(P_1 \Longrightarrow S)$ definition.
$[T_X(S_1) \mathbin{[]} T_X(S_2)]P \Leftrightarrow [S_1 \mathbin{[]} S_2]P$	by Formula (9) and by induction hypothesis
$[@z \cdot T_{X\cup\{z\}}(S)]P \Leftrightarrow [@z \cdot S]P$	by Formula (10) and $[T_{X\cup\{z\}}(S)]P \Leftrightarrow [S]P$ according to $Mod_X(@z \cdot S)$ definition.

Notice that the hypothesis when P is defined only in terms of abstract variables X induces that $[y := E]P = P$ when $y \notin X$ because there is no occurrence of y in P.

We can then conclude that the set of behaviors on the set of abstract variables X of an event ev is unchanged when we simplify it by T_X. $\square$

5.4.2 Propositions 1 and 3

When the set of abstract variables X_{A} is computed by using either Proposition 1 (see Sec. 4.2) or Proposition 3 (see Sec. 4.4), some new behaviors may potentially be introduced in the transition relation projected on X_{A}.

As a consequence of theorems 2 and 3, with the methods defined in Sec. 4.2 (Proposition 1) and Sec. 4.4 (Proposition 3), M refines A. Consequently and according to Sec. 2.4, when A does not remove the deadlocks of M, the ACTL* properties established on A are preserved on M. Otherwise, only the safety properties established on A are preserved on M. However, some tests generated from A might be impossible to instantiate on M since A is an over-approximation, which means that some of its executions may not exist in M.

The refinement theory as defined in B [Abr96b] requires that the variable sets of the abstraction and of the refinement are disjoint. Consequently, when a variable x is preserved through the refinement process, it has to be renamed, e.g. by $x_{renamed}$, and the values of both versions of the variable have to be associated by means of a gluing invariant, such for example as $x = x_{renamed}$. In order to formally express and prove the correctness of the refinement, we introduce the $\mathsf{Ren}()$ function, which renames every variable of a substitution or a predicate. Hence, the substitution S_{A} abstracted from a substitution S_{M}, and the gluing invariant I_{G} are defined as follows:

$$S_{\mathsf{A}} \mathrel{\hat{=}} \mathsf{Ren}(T_X(S_{\mathsf{M}})) \qquad\qquad I_{\mathsf{G}} \mathrel{\hat{=}} \bigwedge_{x_i \in X}(x_i = \mathsf{Ren}(x_i))$$

Theorem 2 *Let I_M be an invariant in CF of a correct B event system M, let S_M be a substitution of M and let X be a set of abstract variables computed by one of the three methods proposed in section 4.1. The slicing rules R_6 to R_{14} are such that S_M refines S_A according to the invariant I_G.*

Proof (of theorem 2)

To prove that S_{M} is a correct refinement of S_{A}, we need to prove (Def. 3):

$$PC_{\mathsf{A}} \wedge PC_{\mathsf{M}} \wedge I_{\mathsf{A}} \wedge I_{\mathsf{M}} \wedge I_{\mathsf{G}} \Rightarrow [S_{\mathsf{M}}]\neg[S_{\mathsf{A}}]\neg(I_{\mathsf{M}} \wedge I_{\mathsf{G}}) \tag{19}$$

where the invariant I_{A} abstracted from I_{M} is defined by $I_{\mathsf{A}} \mathrel{\widehat{=}} \mathsf{Ren}(T_X(I_{\mathsf{M}}))$. In order to prove formula (19), it is sufficient to establish that the following two formulas hold:

$$PC_{\mathsf{A}} \wedge PC_{\mathsf{M}} \;\wedge\; I_{\mathsf{A}} \;\wedge\; I_{\mathsf{M}} \;\wedge\; I_{\mathsf{G}} \;\Rightarrow\; [S_{\mathsf{M}}]\neg[S_{\mathsf{A}}]\neg I_{\mathsf{M}} \tag{20}$$

$$PC_{\mathsf{A}} \wedge PC_{\mathsf{M}} \;\wedge\; I_{\mathsf{A}} \;\wedge\; I_{\mathsf{M}} \;\wedge\; I_{\mathsf{G}} \;\Rightarrow\; [S_{\mathsf{M}}]\neg[S_{\mathsf{A}}]\neg I_{\mathsf{G}} \tag{21}$$

Since the sets of free variables from I_{A} and I_{M} are strictly disjoint, (20) can be rewritten as: $PC_{\mathsf{A}} \wedge PC_{\mathsf{M}} \wedge I_{\mathsf{A}} \wedge I_{\mathsf{M}} \wedge I_{\mathsf{G}} \Rightarrow [S_{\mathsf{M}}]I_{\mathsf{M}}$, that holds, since the initial model M is correct. Hence, we only have to establish (21) to prove Theorem 2. The proof is by induction on the five primitive forms of substitutions. We make a case analysis for each rule of Fig. 11. We use Prop. 2 of Sec. 5.1 and axioms (7 to 11) defined in Sec. 2.

We denote by *Hyps* the repetitive predicate $Hyps \mathrel{\widehat{=}} PC_{\mathsf{A}} \wedge PC_{\mathsf{M}} \wedge I_{\mathsf{A}} \wedge I_{\mathsf{M}} \wedge I_{\mathsf{G}}$.

Case $S_{\mathsf{M}} \mathrel{\widehat{=}} x := E$

Rule R_6 $S_{\mathsf{A}} \mathrel{\widehat{=}} \text{SKIP}$ *when* $x \notin X$

is $Hyps \;\Rightarrow\; [x := E]\neg[\text{SKIP}]\neg I_{\mathsf{G}}$ valid ?

It is valid, according to (7), since x is not free in I_{G}.

Rule R_7 $S_{\mathsf{A}} \mathrel{\widehat{=}} \mathsf{Ren}(x) := \mathsf{Ren}(E)$ *when* $x \in X$

is $Hyps \;\Rightarrow\; [x := E]\neg[\mathsf{Ren}(x) := \mathsf{Ren}(E)]\neg I_{\mathsf{G}}$ valid ?

It is valid since Rule R_7 is the identity.

Case $S_{\mathsf{M}} \mathrel{\widehat{=}} \text{SKIP}$

Rule R_8 $S_{\mathsf{A}} \mathrel{\widehat{=}} \text{SKIP}$

$Hyps \;\Rightarrow\; [\text{SKIP}]\neg[\text{SKIP}]\neg I_{\mathsf{G}}$ is obviously valid according to (7).

Case $S_{\mathsf{M}} \mathrel{\widehat{=}} x, y := E, F$

Rules R_9 **to** R_{11} proofs are similar to the first case.

Case $S_{\mathsf{M}} \mathrel{\widehat{=}} P \implies S$

Rule R_{12} $S_{\mathsf{A}} \mathrel{\widehat{=}} \mathsf{Ren}(T_X(P)) \implies \mathsf{Ren}(T_X(S))$

is $Hyps \;\Rightarrow\; [P \implies S]\neg[\mathsf{Ren}(T_X(P)) \implies \mathsf{Ren}(T_X(S))]\neg I_{\mathsf{G}}$ valid ?

$\equiv Hyps \;\Rightarrow\; (P \implies [S](\mathsf{Ren}(T_X(P)) \;\wedge\; \neg[\mathsf{Ren}(T_X(S))]\neg I_{\mathsf{G}}))$ – applying (8)

$\equiv \begin{cases} \quad (\mathbf{A.})\; (Hyps \;\wedge\; P \;\Rightarrow\; [S]\mathsf{Ren}(T_X(P))) \\ \wedge\; (\mathbf{B.})\; (Hyps \;\wedge\; P \;\Rightarrow\; [S]\neg[\mathsf{Ren}(T_X(S))]\neg I_{\mathsf{G}}) \end{cases}$ – applying (11)

According to Prop 2, (**A**) holds since S variables are not free in $\mathsf{Ren}(T_X(P))$ and since I_{G} is in *Hyps*. (**B**) is valid w.r.t. the induction hypothesis: $Hyps \;\Rightarrow\; [S]\neg[\mathsf{Ren}(T_X(S))]\neg I_{\mathsf{G}}$.

Case $S_{\mathsf{M}} \mathrel{\widehat{=}} S_1 \;[]\; S_2$

Rule R_{13} $S_{\mathsf{A}} \mathrel{\widehat{=}} \mathsf{Ren}(T_X(S_1)) \;[]\; \mathsf{Ren}(T_X(S_2))$

is $Hyps \;\Rightarrow\; [S_1 \;[]\; S_2]\neg[\mathsf{Ren}(T_X(S_1)) \;[]\; \mathsf{Ren}(T_X(S_2))]\neg I_{\mathsf{G}}$ valid ?

$\equiv Hyps \;\Rightarrow\; [S \;[]\; S_2](\neg[\mathsf{Ren}(T_X(S))]\neg I_{\mathsf{G}} \;\vee\; \neg[\mathsf{Ren}(T_X(S_2))]\neg I_{\mathsf{G}})$ – applying (9)

$\equiv \begin{cases} \quad (Hyps \;\Rightarrow\; [S_1](\neg[\mathsf{Ren}(T_X(S_1))]\neg I_{\mathsf{G}} \;\vee\; \neg[\mathsf{Ren}(T_X(S_2))]\neg I_{\mathsf{G}})) \\ \wedge (Hyps \;\Rightarrow\; [S_2](\neg[\mathsf{Ren}(T_X(S_1))]\neg I_{\mathsf{G}} \;\vee\; \neg[\mathsf{Ren}(T_X(S_2))]\neg I_{\mathsf{G}})) \end{cases}$ – applying (9)

This formula is valid because the two induction hypotheses are valid:

1. $Hyps \;\Rightarrow\; [S_1]\neg[\mathsf{Ren}(T_X(S_1))]\neg I_{\mathsf{G}}$,
2. $Hyps \;\Rightarrow\; [S_2]\neg[\mathsf{Ren}(T_X(S_2))]\neg I_{\mathsf{G}}$.

Case $S_{\mathsf{M}} \mathrel{\widehat{=}} @z \cdot S$

Rule R_{14} $S_{\mathsf{A}} \mathrel{\widehat{=}} \mathsf{Ren}(@z \cdot T_{X \cup \{z\}}(S))$

is $Hyps \;\Rightarrow\; [@z \cdot S]\neg[\mathsf{Ren}(@z \cdot T_{X \cup \{z\}}(S))]\neg I_{\mathsf{G}}$ valid ?

$\equiv Hyps \;\Rightarrow\; \forall z \cdot [S]\neg\forall \mathsf{Ren}(z) \cdot [\mathsf{Ren}(T_{X \cup \{z\}}(S))]\neg I_{\mathsf{G}}$ – applying (10)

It is valid since the following formula is implied by the induction hypothesis:

$Hyps \;\Rightarrow\; \forall z \cdot \exists \mathsf{Ren}(z) \cdot (z = \mathsf{Ren}(z) \;\wedge\; [S]\neg[\mathsf{Ren}(T_{X \cup \{z\}}(S))]\neg(I_{\mathsf{G}} \;\wedge\; z = \mathsf{Ren}(z)))$

Hence, Theorem 2 holds. □

Theorem 2 establishes that any substitution S refines its slicing $T_X(S)$ for a set of abstract variables X computed by one of the propositions described in sec 4.1. Theorem 3 establishes that a B event system M refines the B abstract system obtained according to Def. 5 by applying to M the slicing rules of Fig. 9 and Fig. 11.

Theorem 3 *Let X be a set of abstract variables defined as in Proposition 1 or in Proposition 3. Let T_X be the slicing defined in Fig. 11, and let A be an abstraction of an event system M defined according to Def. 5. A is refined by M in the sense of Def. 3.*

Proof (of theorem 3) This is a direct consequence of theorem 2 and Def. 5 since the substitution $Init_{\mathsf{A}} \widehat{=} T_X(Init_{\mathsf{M}})$ is refined by $Init_{\mathsf{M}}$, and that for any event $ev \widehat{=} S_{\mathsf{M}}$, the substitution $S_{\mathsf{A}} \widehat{=} T_X(S_{\mathsf{M}})$ is refined by S_{M}. $\square$

Notice that the set of abstract variables obtained when applying Proposition 3 is bounded between the sets of Propositions 1 and 2. This means that the abstraction A obtained is either a bisimulation of M when X_{A} of Proposition 3 is equal to X_{A} of Proposition 2, or a simulation when A does not remove deadlocks of M and that X_{A} of Proposition 3 is strictly included into X_{A} of Proposition 2 .

6 Application of the Method to a Testing Process

We show in this section how to use the variable abstraction in a model-based testing approach.

6.1 Test Generation from an Abstraction

We have described in [BBJM10] a model-based testing process using an abstraction as input. It can be summarized as follows. A validation engineer describes by means of a handwritten test purpose TP how he intends to test the system, according to his know-how. We have proposed in [JMT08] a language based on regular expressions to describe a TP as a sequence of actions to fire and states to reach (targeted by these actions). The actions can be explicitly called in the shape of event names, or left unspecified by the use of a generic name. The unspecified calls then have to be replaced with explicit event names. However, a combinatorial explosion problem occurs, when searching in a concrete model for the possible replacements that lead to the target states. This led us to use abstractions instead of concrete models. Figure 13 shows our approach.

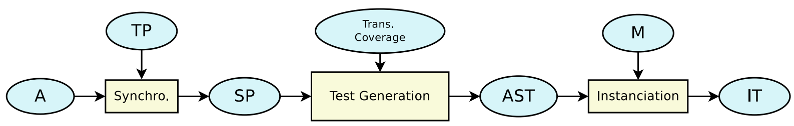

Fig. 13 Generating Tests from Test Purpose by Abstraction

We compute the symbolic abstract tests as selected executions of the abstraction, by covering all the transitions of the synchronized product SP between the abstraction A and the TP (see Fig. 13). This provides a set of paths such that every transition of SP is covered at least once. Every path is a symbolic abstract test that terminates in a final state of SP. It is a sequence of non parameterized action calls. We still have to instantiate the tests, i.e. to find parameter values that make these sequencings of actions possible according to the behavioral model M.

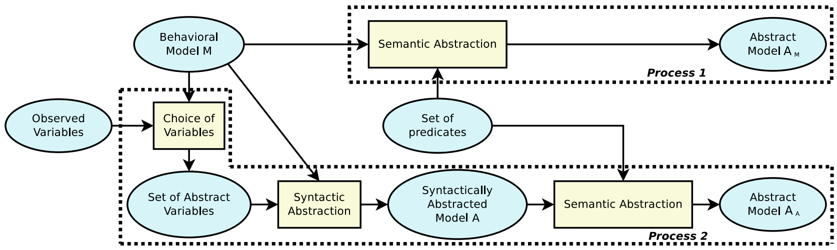

Fig. 14 Comparison of Two Abstraction Processes

6.2 Abstraction Computation

We show in this section a process that compares two ways of producing an abstraction A that can be used as an input of the process of Fig. 13. One of these two ways relies on the variable abstraction presented in Sec. 4.

Before we compute the synchronized product of an abstraction A with the automaton of a TP, we first compute the semantics of A as a labelled transition system. This is obtained by means of an algorithm that performs a semantic abstraction by predicate abstraction, and results in a symbolic labelled transition system as explained in Sec. 2.2. The algorithm proceeds by removing from all the potential transitions the ones whose unfeasibility is proved. This is achieved by computing a set of proof obligations (POs), that are tried to be discharged automatically. It results in transitions being proved not to exist when the proof terminates. When a PO fails to be discharged automatically, the existence or not of the corresponding transition remains uncertain.

The two main drawbacks of this semantic abstraction process are its time cost and the proportion of POs not automatically discharged. Indeed, each failed PO results in a transition that is kept in the symbolic labelled transition system, although it is possibly unfeasible. An abstract symbolic test going through such a transition may not be possible to instantiate from the concrete model M. Our intention is to reduce the impact of that problem by reducing the number and the size of the POs. For this, we apply a preliminary phase of syntactic abstraction, for the (semantic) predicate abstraction to operate on an already abstracted model. For example, no proof obligation is generated for an event reduced to SKIP, that becomes a reflexive transition on any symbolic state.

In Fig. 14 we confront two processes for computing an abstraction. In Fig. 14/Process 1, an abstraction A_M is computed by a completely semantic process, i.e. by applying directly the predicate abstraction to the source model. In Fig. 14/Process 2, an abstraction A_A is computed in two steps. First, a static variable slicing is applied to the source model, and then the semantic abstraction is applied to the resulting model. Notice that the observed variables are the free variables of the abstraction predicates that are issued from a test purpose.

We have compared these two processes experimentally. The results appear in Sec. 7.

7 Experimentations

We have applied our method to six case studies, that are various cases of reactive systems: an automatic conveying system (Robot [BBJM09]), a reverse phone book service (QuiDonc [UL06]), the electrical system (Electr., see Sec. 3.1), an electronic purse (De-

Money [MM02]), the elevator specification (see Sec. 3.2) and a laptop daemons management specification [2].

In our experiments, we compute and compare tests issued from four abstractions of each source model. The first abstraction is obtained by applying directly a semantic abstraction to the source model (see Process 1 of Fig. 14). The three other ones are obtained by preliminarily reducing the model by means of variable slicing, before the semantic abstraction is applied (see Process 2 of Fig. 14). This gives three abstractions according to the three propositions to compute the abstract variables (see Sec. 4). We evaluate the results by computing the ratio of the number of instantiated steps of test on the total number of steps of test, and by measuring the state and transition coverage of the abstract models by the tests. All our abstraction predicates are issued from a very small set of observed variables. In Process 2, each set of observed variables gives three sets of abstract variables, according to Propositions 1, 2 and 3 defined in Sec. 4.1.

We present in Sec. 7.1 the tools that we have used for the experimentations and in Sec. 7.2 the experimental results. In Sec. 7.2.1 we present an experimental evaluation of the syntactic abstraction on the size of the models. Then, in Sec. 7.2.2, we compare the execution time to compute A_M and A_A respectively by the semantic abstraction process (Process 1) or by its combination with the syntactic one (Process 2). We also compare the sets of execution paths of the abstractions. Finally, in Sec. 7.2.3, we compare the impact of the abstraction, computed with each of the three propositions defined in Sec. 4.1, on the generated tests. We conclude about these experimental results in Sec. 7.2.4.

7.1 Tools Used for the Experimentation

The experimental results presented in this section were obtained by using a set of tools that we present here.

7.1.1 Semantic Abstraction Generation

We have used *GeneSyst*[3] [BPS05] to generate an abstraction from a behavioral model M and a set of abstraction predicates. This abstraction is a symbolic labelled transition system (LTS) that is an over-approximation of M: it simulates all the executions of M, but possibly adds new ones. *GeneSyst* computes the abstract states according to a set of abstraction predicates, and tries to prove automatically the *non* feasibility of transitions between any two abstract states. It proceeds by weakest precondition computations and satisfiability evaluations over first order logical formulas. *GeneSyst* takes B specifications as input. As indicated in Sec. 2, the weakest precondition of a statement S that leads to the abstract state q', as defined by the B substitution calculus, is denoted by $[S]q'$. If $q \Rightarrow [S]\neg q'$ is valid then no transition from q to q' is feasible by S, hence no transition by S from q to q' is added to the LTS. If the validity of $q \Rightarrow [S]\neg q'$ cannot be established, including the case where the proof is inconclusive, then the transition is added to the LTS, although it is possibly not feasible.

Thus, some of the symbolic tests that we generate from the abstraction may not be possible to instantiate as executions of the behavioral model. This would result in a bad coverage of the abstraction by the instantiated tests. It is possible to use an interactive prover

[2] see `http://lifc.univ-fcomte.fr/testAndAbs/laptop.html`

[3] see `http://perso.citi.insa-lyon.fr/nstouls/?ZoomSur=GeneSyst`

to try to get rid of the proof failures. To keep the process as automatic as possible, we have chosen another alternative: using constraint solving techniques makes it possible to automatically check the feasibility (i.e. the satisfiability of $q \wedge \neg[S]\neg q'$) of the unproved transitions when the state space is finite. We have used the CLPS-B [BLP04] constraint solver, able to deal with B specifications, for that purpose. The applicability of this technique depends on the size of the domains, as it proceeds by partial consistency checking and domain enumeration. The semantic abstractions considered in this paper were obtained by using *GeneSyst* enhanced with a CLSP-B constraint solving phase.

7.1.2 Test Generation and Instantiation

To compute the symbolic abstract tests, we cover every transition of the abstraction but the reflexive ones by running the implementation presented in [Thi03] of the chinese postman algorithm.

We have implemented the symbolic animation of the tests on M to instantiate them. It is possible that a sequence can not be instantiated as it is: an action might not be enabled on a given instance of a symbolic state. Thus we will use a version of the abstraction augmented with its reflexive transitions to complete the instantiation. Indeed, these transitions may lead to another instance of the same symbolic state, from which the action could be enabled. As a result, we insert bounded sub-sequences of (reflexive) action calls into the original sequence. We have implemented this instantiation procedure. Although non optimized and incomplete (invoking reflexive transitions is not always sufficient, sometimes cycles are necessary), our algorithm gave satisfactory instantiation results on our case studies, as shown by our experiments in Table 4.

7.2 Experimental Study

In this section we show the results of the first experiments on the propositions presented in this paper. These are early experiments since not all the tools have been developed yet to allow for dealing with larger examples. In particular, we have no tool yet to compute the sets of abstract variables from the observed ones according to each of the three propositions, nor to perform the resulting slicing on the models. These early experiments nevertheless reveal some tendencies, that we present hereafter.

7.2.1 Impact of the Syntactic Abstraction on the Models

Table 2 indicates the sizes[4] of the source and syntactically abstracted models of the case studies. The symbols “$\sharp$”, “Var.”, “Ev.”, “Pot.”, “Prop.” respectively stand for *number of*, *Variables*, *Events*, *Potential* and *Proposition*. The Robot for example, is modelled with six variables and nine events. It is abstracted w.r.t. two observed variables, which gives three sets of abstract variables, one by proposition.

A direct observable result of the syntactic abstraction is a reduction of the number of variables kept in the model, at least with Propositions 1 and 3. We see that Proposition 1 syntactically removes more variables than the other two propositions, which results in less potential states when there is not an infinity of them. So the models abstracted by means

[4] The 90 lines length of the electrical system model, in Table 2, refers to a “verbose” version of the model, much more readable than our version of Fig. 3.

Case Study	Model M				Syntactically Abstracted Model A					
	♯Var.	♯Ev.	♯Pot. States	♯B Lines	♯Observed Var.	Prop.	♯Abstract Var.	♯Skip Ev.	♯Pot. States	♯B Lines
Robot	6	9	576	110	2	1	3	0	48	100
						2	6	0	576	110
						3	6	0	576	110
QuiDonc	3	4	36	180	2	1	2	0	18	170
						2	3	0	36	180
						3	3	0	36	180
Electr.	3	4	∞	90	1	1	1	2	∞	60
						2	2	0	∞	70
						3	2	0	∞	70
DeMoney	8	11	10^{30}	330	1	1	4	4	10^{20}	150
						2	8	0	10^{30}	330
						3	6	3	10^{25}	280
Elevator	6	5	∞	140	1	1	2	1	∞	90
						2	4	0	∞	110
						3	3	0	∞	100
Laptop	5	6	∞	200	1	1	2	3	∞	160
						2	4	0	∞	190
						3	3	0	∞	180

Table 2 Size of the Case Studies and of their Syntactical Abstractions

of Proposition 1 are the smallest ones. This is not surprising since only the data flow of the abstract variables is preserved by Proposition 1. As for Proposition 2, by preserving both the data and control flow of the abstract variables, there is on the contrary a risk that all the variables become mutually dependent. This is confirmed by our experimental results: in half of the cases, no variable has been removed by Proposition 2. Proposition 3 offers a good compromise by partially preserving the control flow in addition to the data flow. It has simplified four models out of six, without too much loss of precision of the abstraction as Sec. 7.2.2 and Sec. 7.2.3 show.

Table 2 also shows that the simplification reduces by 10% up to 55% the number of lines of the models, when some variables are removed. The next two sub-sections (7.2.2 and 7.2.3) study the impact of the syntactical simplifications on the time and number of proof obligations to generate the abstractions, and on their precision.

7.2.2 Impact of the Processes on the Abstractions and their Computation

Table 3 compares the abstractions computed either directly from the behavioral models (see Process 1 in Fig. 14), or from their syntactic abstractions (see Process 2 in Fig. 14). The abbreviations “Symb.”, “Trans.” and “Unau.” stand respectively for *symbolic*, *transitions* and *unauthorized*.

We see on our examples that there is up to 2.5 fewer POs to compute with Process 2 than with Process 1. In most of the cases, there are less POs after a syntactic abstraction because some events have been reduced to SKIP or to $P \Longrightarrow \text{SKIP}$. Unsurprisingly, the better reduction is obtained in five cases out of six with Proposition 1, but there is also a risk that on the contrary the number of POs grows, if for example an event becomes so much simplified that it can occur all the time, as was the case with the QuiDonc example. The number of POs never grows with Propositions 2 and 3 on our examples.

A gain in the number of POs directly results in a better time to compute the abstractions. With Demoney and Proposition 1, the gain amounts to 95%. More generally, Process 2 takes twice less time in average than Process 1, where no previous syntactic abstraction is

Case study	♯Symb. States	Process 1 : A_M				Process 2 : A_A						Set of Traces Comparison
		♯Trans.	♯Unau. Trans.	♯PO	Time (s)	Prop.	♯Trans.	♯Unau. Trans. Over-Approx.	♯Unau. Trans. Proof Failure	♯PO	Time (s)	
Robot	6	41	5	263	71	1	36	0	0	143	34	$A_A \subset A_M$
						2	41	0	5	263	71	$A_M = A_A$
						3	41	0	5	263	71	$A_M = A_A$
QuiDonc	5	19	2	71	21	1	21	4	0	85	25	$A_M \neq A_A$
						2	19	0	2	71	21	$A_M = A_A$
						3	19	0	2	71	21	$A_M = A_A$
Electr.	2	10	2	24	8	1	10	0	2	12	4	$A_M = A_A$
						2	10	0	1	24	7	$A_M = A_A$
						3	10	0	1	24	7	$A_M = A_A$
DeMoney	3	35	1	78	400	1	35	0	1	33	19	$A_M = A_A$
						2	35	0	1	78	392	$A_M = A_A$
						3	35	0	1	48	292	$A_M = A_A$
Elevator	3	14	2	59	17	1	12	0	0	35	8	$A_A \subset A_M$
						2	14	0	2	59	15	$A_M = A_A$
						3	14	0	2	55	13	$A_M = A_A$
Laptop	3	19	2	64	22	1	20	1	2	30	11	$A_M \subset A_A$
						2	19	0	2	64	21	$A_M = A_A$
						3	19	0	2	64	16	$A_M = A_A$

Table 3 Comparison of the Semantic and Syntactic/Semantic Abstraction Processes

performed. We notice that there is no significative gain of time by using Proposition 2 to preliminarily abstract the models.

The unauthorized transitions are an indication of the precision of an over-approximation: the more unauthorized transitions are added, the more the approximation will define unfeasible paths. By too much over-approximating the source model, Proposition 1 can add new unfeasible transitions: 4 with QuiDonc and 1 with the Laptop case study. But neither Proposition 2 (that bisimulates the source model) nor Proposition 3 have added unfeasible transitions in our experiments. In particular Proposition 3, that nevertheless offered a gain of time in the abstraction computation.

The last result to observe in Table 3 is that, in most of the cases, the abstractions computed by the two processes are identical in terms of their sets of traces, although they are not comparable in the general case. We have obtained all the cases on our examples: $A_M = A_A$ (in 78% of the cases), $A_M \subset A_A$, $A_A \subset A_M$ and $A_M \neq A_A$. Only with Proposition 1 we have observed a difference in the set of traces.

Let us now look more closely at each of these different cases of traces inclusion. For the Laptop case study abstracted with Proposition 1, the set of traces of A_M is included into that of A_A. This is explained by the fact that one transition of A_A results only of the syntactic over-approximation of the model with Proposition 1. In this case, the model is too much simplified by the slicing, so that events that could not be triggered before become triggerable in the syntactically abstracted model. We also observe the dual case ($A_A \subset A_M$) on the Robot and the Elevator abstracted with Proposition 1. In these examples, the syntactically abstracted model creates less and simpler POs than the source one. This results in less proof failures, so that the abstraction computed from the syntactically abstracted model is more precise than the one computed from the source model. The last case is when the sets of traces of A_A and of A_M can not be compared. It appears in the QuiDonc abstracted with Proposition 1. In this example, some transitions result from the over-approximation of the syntactic abstraction in Process 2, but some other transitions that existed due to proof failures in Process 1 have been eliminated because their proof succeeds on the syntactically abstracted model.

So as a conclusion, Proposition 1 gives the best times to compute the abstractions, but they might be too imprecise. Proposition 2, the most precise, did not produce an observable gain of time in our experiments and so Proposition 3 seems to offer a good trade-off as no loss of precision has been observed though the abstractions were produced faster than with Process 1. Demoney, the largest of our examples, is the most demonstrative of that point.

7.2.3 *Impact of the Abstractions on the Generated Tests*

Table 4 compares the test generation and instantiation results of Processes 1 and 2, but also of the three propositions of syntactic abstraction.

Case Study	Process 1 : $\mathsf{A_M}$			Process 2 : $\mathsf{A_A}$			
	$\sharp$Inst. Steps / $\sharp$Steps	State cov. on $\mathsf{A_M}$	Trans. cov. on $\mathsf{A_M}$	Prop.	$\sharp$Inst. Steps / $\sharp$Steps	State cov. on $\mathsf{A_A}$	Trans. cov. on $\mathsf{A_A}$
Robot	29/40 (72%)	5/6 (83%)	29/36 (81%)	1	37/40 (93%)	6/6 (100%)	34/36 (95%)
				2	29/40 (72%)	5/6 (83%)	29/36 (81%)
				3	29/40 (72%)	5/6 (83%)	29/36 (81%)
QuiDonc	20/29 (69%)	5/5 (100%)	14/19 (74%)	1	18/27 (67%)	5/5 (100%)	13/21 (62%)
				2	20/29 (69%)	5/5 (100%)	14/19 (74%)
				3	20/29 (69%)	5/5 (100%)	14/19 (74%)
Electr.	8/8 (100%)	2/2 (100%)	8/8 (100%)	1	8/8 (100%)	2/2 (100%)	8/8 (100%)
				2	8/8 (100%)	2/2 (100%)	8/8 (100%)
				3	8/8 (100%)	2/2 (100%)	8/8 (100%)
DeMoney	64/64 (100%)	3/3 (100%)	34/34 (100%)	1	64/64 (100%)	3/3 (100%)	34/34 (100%)
				2	64/64 (100%)	3/3 (100%)	34/34 (100%)
				3	64/64 (100%)	3/3 (100%)	34/34 (100%)
Elevator	12/12 (100%)	3/3 (100%)	12/12 (100%)	1	12/12 (100%)	3/3 (100%)	12/12 (100%)
				2	12/12 (100%)	3/3 (100%)	12/12 (100%)
				3	12/12 (100%)	3/3 (100%)	12/12 (100%)
Laptop	20/20 (100%)	3/3 (100%)	17/17 (100%)	1	20/20 (100%)	3/3 (100%)	17/17 (100%)
				2	20/20 (100%)	3/3 (100%)	17/17 (100%)
				3	20/20 (100%)	3/3 (100%)	17/17 (100%)

Table 4 Impact of the Abstraction Process on the Test Generation

It appears that for the QuiDonc example, the transitions coverage ratio by the tests is lower on the semantic abstraction $\mathsf{A_A}$ obtained after the source model has been reduced by Proposition 1 than on $\mathsf{A_M}$, obtained by directly applying the semantic abstraction on the source model. This is not surprising: it corresponds to the case where $\mathsf{A_A} \neq \mathsf{A_M}$. In contrast for the Robot example, this transition coverage ratio is greater. In this case, the set of traces of $\mathsf{A_A}$ is included in the set of traces of $\mathsf{A_M}$.

Proposition 2 gives satisfactory results in terms of precision of the abstraction, but the drawback is that often, there is no simplification at all. This happens when all the variables are mutually dependent, as indicated by Table 2 and Table 3. In the QuiDonc case, both Proposition 2 and Proposition 3 give better test coverage ratios than Proposition 1. We note that Proposition 3 is lighter to compute than Proposition 2.

There again, Proposition 3 appears to provide a good trade-off between the efficiency of the simplification and the precision of the abstraction computed. In our examples, the test coverage produced on one hand with Process 1, and on the other hand with Process 2 and Proposition 3 are always the same. But the gain is in terms of number of POs generated, of easiness to discharge them, and of time to compute the abstractions, as indicated in Sec. 7.2.2.

7.2.4 Conclusion of the Experiments

These early experimental results confirm the interest in first performing a syntactic slicing of the model before producing the semantic abstraction. This globally accelerates the process of computing the final abstraction. But this shows that Proposition 1 should be used with care since it might too much over-approximate the source model. It can be used to quickly get an abstraction that gives a first graphical overview of the behavior of the system. Using Proposition 2 was not very conclusive on our case studies since it did not produce a benefit in the time to get the abstraction. It should however be further experimented with larger examples, in particular when not all the variables are mutually dependent. This could occur with a system made of several independent parts. Finally Proposition 3 appears to be the most promising as a compromise between efficiency of the abstraction computation and precision of the abstraction.

8 Related works

The works related to the ones presented in this paper are about program slicing and abstraction methods for test generation.

Our method is an adaptation to model slicing of the program slicing techniques that were introduced in [Wei84]. A survey of these techniques can be found in [Tip95]. Our approach performs a static slicing. The control and data dependencies computation are different in our method than in the program slicing as defined in [Wei84]. In [Wei84], the dependencies are evaluated syntactically by means of data and control dependencies equations whereas in our approach, they are evaluated semantically by simplification of the predicate Mod_X based on the before-after predicates of the events. Hence we only take into account the cases where the variables are actually modified. In program slicing, the static slicing criterion is a pair made of a value of the program counter and of a set of variables. Our model slicing criterion is only a set of state variables. Hence the program slicing preserves the variables computation in the state given by the value of the program counter, whereas our model slicing preserves the variables computation in any observable state. Moreover, notice that in the case of Data-Flow dependency only as well as in the case of Data and partial Control-Flow dependencies, the system can be over-approximated by adding new executions, but it has a very low computation cost.

Slicing has also been used for state-based system models, e.g. for extended hierarchical automata [HW97, DHH$^+$06] or for input/output transitions systems [LGP07]. But most of these approaches work on relatively low-level model representations, in contrast to B models that capture the high-level design intuition.

Our contribution is mainly inspired by [BW05] that proposes a model slicing method based on the CSP-ObjectZ integrated method. Our goal is similar. It is to reduce the size of the specification in order to simplify further verifications. However, we propose new original approaches to compute the set of relevant variables. We don't have the same restrictions since an over approximation of a model allows to generate tests, to check their concrete execution and to instantiate them on the initial model.

Many other works define model abstraction methods to verify properties or to generate tests. The method of [FHNS02] uses an extension of the model-checker Murϕ to compute tests from projected state coverage criteria that eliminate some state variables and project the others on abstract domains. In [DF93], an abstraction is computed by partition analysis of a state-based specification, based on the pre and post conditions of the operations. Constraint solving techniques are used. The methods of [GS97, BLO98, CU98] use theorem proving to compute the abstract model, which is defined over boolean variables that correspond to a set of predicates fixed *a priori*. In contrast, our method first introduces a syntactical abstraction computation from a set of observed variables, and further abstracts it by theorem proving. [CABN97] also performs a syntactic transformation, but requires the use of a constraint solver during a model checking process.

Other automatic abstraction methods [CGL94] are limited to finite state systems. The deductive model checking algorithm of [SUM99] produces an abstraction w.r.t. a LTL property by an iterative refinement process that requires human expertise. Our method can handle infinite state space specifications. The paper [NK00] presents a syntactic abstraction method for guarded command programs based on assignment substitution. The method is sound and complete for programs without unbounded non determinism. However, the method is iterative and does not terminate in the general case. It requires the user to give an upper-bound of the number of iterations. The paper also presents an extension for unbounded non deterministic programs that is sound but not complete, due to an exponential number of predicates generated at each iteration step. In contrast, our syntactic method is iterative on the syntactic structure of the specifications. It is sound but not complete. It handles unbounded non deterministic specifications with no need for other iterative process and always terminates. Above all, our method does not compute any weakest precondition whereas the approach in [NK00] does, which possibly introduces infinitely many new predicates.

9 Conclusion and Further works

We have presented in the B framework a method for abstracting an event system by elimination of some state variables. In this context, we have proposed three methods to compute the set of variables kept in the abstraction according to a set of observed variables. We have proved that when using the first and the third method, the generated abstraction simulates the concrete model, while when using the second method, the generated abstraction bisimulates the concrete model. This is useful for verifying safety properties and generating tests.

In the context of test generation, our method proceeds by initializing the test generation process described in [BBJM10] with a B event model reduced by a syntactic abstraction. Since the syntactic abstraction reduces the size of the model in general, the main advantage of this method is that it generally reduces the set of non instantiable tests, by reducing the level of abstraction. It reduces the number of POs generated and facilitates the proof of the remaining POs. Moreover, this results in a gain of computation time. We believe that the bigger the ratio of the number of state variables to the number of observed variables is, the bigger the gain is. This conjecture, exemplified by the experimental results on the Demoney case study, needs to be confirmed by experiments on industrial size applications.

The syntactic method that we have presented is correct but, in the case of Proposition 1 and Proposition 3, may sometimes produce imprecise over-approximations due to a too strong abstraction (see for example the experiments on the QuiDonc). Proposition 2 produces a bisimulation, but may leave the initial model unchanged, i.e. not abstracted, if all the variables are computed as abstract. We propose by means of Proposition 3 a compromise between the two propositions, that aims at reducing the number of abstract variables, while keeping at least partially the control structure of the operations. Hence this method produces a more precise approximation that improves the results of the test generation application.

Since our main motivation is to propose a method that reduces the time for computing an abstraction of a model, the definition of $Mod_X(S)$ can be seen as out of scope. Indeed, its definition is given in the general case and requires a constraint solver to be fully usable. However, the proposition made in Fig. 7 shows that some syntactic rules can provide a good trade-off between the computation cost of an abstraction and its full simplification. Similarly to the IF substitution, other rules have to be proposed for exploiting all the information provided by the B syntactical sugar.

Also, we think that the transformation rules could be improved in order to get more precise approximations, possibly with a type induction process in order to ease the withdrawing of non-abstract variables. For instance, improving the rules is possible when the invariant contains an equivalence such as $x = c \Leftrightarrow y = c'$. If y is an eliminated variable and x is an observed one, we could substitute all the occurrences of the elementary predicate $y = c'$ with $x = c$. This would preserve the property in the syntactic abstraction A_A, so that the following semantic abstraction would be more precise. Such rules should prevent the addition of transitions in the QuiDonc abstraction A_A w.r.t. A_M.

We think that extending the test generation method introduced in [BBJM10] by using a combination of syntactic and semantic abstractions will improve the method, since the abstraction is more precise if there are less unproved POs. Moreover, as aforementioned, the time for computing the semantic abstraction is reduced by a static slicing of the models.

References

[Abr96a] J.-R. Abrial. Extending B without changing it (for developing distributed systems). In *1st B Conference*, pages 169–190, 1996.

[Abr96b] Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[Bal05] T. Ball. A theory of predicate-complete test coverage and generation. In *FMCO'04*, volume 3657 of *LNCS*, pages 1–22. Springer, 2005.

[BBJM09] F. Bouquet, P.-C. Bué, J. Julliand, and P.-A. Masson. Génération de tests à partir de critères dynamiques de sélection et par abstraction. In *AFADL'09*, pages 161–176, Toulouse, France, January 2009.

[BBJM10] Fabrice Bouquet, Pierre-Christophe Bué, Jacques Julliand, and Pierre-Alain Masson. Test generation based on abstraction and test purposes to complement structural tests. In *A-MOST'10, 6th int. Workshop on Advances in Model Based Testing*, Paris, France, April 2010.

[BCDG07] F. Bouquet, J.-F. Couchot, F. Dadeau, and A. Giorgetti. Instantiation of parameterized data structures for model-based testing. In *B'2007, the 7th Int. B Conference*, volume 4355 of *LNCS*, pages 96–110. Springer, 2007.

[BJK00] Françoise Bellegarde, Jacques Julliand, and Olga Kouchnarenko. Ready-simulation is not ready to express a modular refinement relation. In *FASE'2000*, volume 1783 of *LNCS*, pages 266–283, 2000.

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

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *CAV'98*, volume 1427 of *LNCS*. Springer, 1998.

[BLP04] Fabrice Bouquet, Bruno Legeard, and Fabien Peureux. CLPS-B: A constraint solver to animate a B specification. *STTT, International Journal on Software Tools for Technology Transfer*, 6(2):143–157, August 2004.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *PLDI*, pages 203–213, 2001.

[BPS05] D. Bert, M.-L. Potet, and N. Stouls. GeneSyst: a Tool to Reason about Behavioral Aspects of B Event Specifications. In *ZB'05*, volume 3455 of *LNCS*, 2005.

[BW05] Ingo Brückner and Heike Wehrheim. Slicing an Integrated Formal Method for Verification. In Kung-Kiu Lau and Richard Banach, editors, *ICFEM'05*, volume 3785 of *LNCS*, pages 360–374. Springer, November 2005.

[CABN97] W. Chan, R. Anderson, P. Beame, and D. Notkin. Combining Constraint Solving and Symbolic Model Checking for a Class of Systems with Non-Linear Constraints. In *CAV'97*, volume 1254 of *LNCS*. Springer, 1997.

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

[CGL94] E.M. Clarke, O. Grumberg, and D. Long. Model Checking and Abstraction. *TOPLAS'94, ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[CGS09] J-F. Couchot, A. Giorgetti, and N. Stouls. Graph-based Reduction of Program Verification Conditions. In *AFM'09*, 2009.

[CU98] M.A. Colon and T.E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *CAV'98*, volume 1427 of *LNCS*, 1998.

[DF93] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *FME'93*, pages 268–284, 1993.

[DHH+06] Matthew B. Dwyer, John Hatcliff, Matthew Hoosier, Venkatesh Prasad Ranganath, Robby, and Todd Wallentine. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In *TACAS*, pages 73–89, 2006.

[DJK03] Christophe Darlot, Jacques Julliand, and Olga Kouchnarenko. Refinement preserves PLTL properties. In *Third International Conference of B and Z Users ZB'03 - Formal Specification and Development in Z and B*, volume 2651 of *LNCS*, pages 408–420, Turku, Finland, June 2003.

[FHNS02] G. Friedman, A. Hartman, K. Nagin, and T. Shiran. Projected state machine coverage for software testing. In *ISSTA*, pages 134–143, 2002.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV'97*, volume 1254 of *LNCS*, 1997.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 10(12):576580, 1969.

[HW97] Mats Per Erik Heimdahl and Michael W. Whalen. Reduction and slicing of hierarchical state machines. In *ESEC / SIGSOFT FSE*, pages 450–467, 1997.

[JMT08] J. Julliand, P.-A. Masson, and R. Tissot. Generating security tests in addition to functional tests. In *AST'08*, pages 41–44. ACM Press, 2008.

[JSBM10] Jacques Julliand, Nicolas Stouls, Pierre-Christophe Bué, and Pierre-Alain Masson. Syntactic Abstraction of B Models to Generate Tests. In G. Fraser and A. Gargantini, editors, *TAP'10, 4th Int. Conf. on Tests and Proofs*, volume 6143 of *LNCS*, pages 151–166, Malaga, Spain, July 2010.

[LB08] M. Leuschel and M. Butler. ProB: An automated analysis toolset for the B method. *Software Tools for Technology Transfer*, 10(2):185–203, 2008.

[LGP07] Sébastien Labbé, Jean-Pierre Gallois, and Marc Pouzet. Slicing communicating automata specifications for efficient model reduction. In *ASWEC*, pages 191–200, 2007.

[MM02] R. Marlet and C. Mesnil. Demoney: A demonstrative electronic purse Technical Report SECSAFE-TL-007, Trusted Logic, 2002.

[NK00] K. S. Namjoshi and R. P. Kurshan. Syntactic program transformations for automatic abstraction. In *CAV'00*, volume 1855 of *LNCS*, pages 435–449. Springer, 2000.

[SUM99] H. Sipma, T. Uribe, and Z. Manna. Deductive model checking. *Formal Methods in System Design*, 15(1):49–74, 1999.

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

[Tip95] F. Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

[UL06] M. Utting and B. Legeard. *Practical Model-Based Testing - A tools approach.* Elsevier Science, 2006.

[Wei84] Mark Weiser. Program slicing. *Software Engineering, IEEE Transactions on*, SE-10(4):352–357, july 1984.