LIFC

LABORATOIRE D'INFORMATIQUE DE L'UNIVERSITE DE FRANCHE-COMTE

EA 4269

B Model Abstraction Combining Syntactic and Semantics Methods

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

Rapport de Recherche n° RR 2009–4

THÈME 2 – November 20, 2009

B Model Abstraction Combining Syntactic and Semantics Methods

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

Thème 2
VESONTIO

November 20, 2009

Abstract: In a model-based testing approach as well as for the verification of properties by model-checking, B models provide an interesting solution. But 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, often combining state variable elimination and domain abstractions of the remaining variables. This paper presents a computer aided abstraction process that combines syntactic and semantic abstraction functions. The first function syntactically transforms a B event system M into an abstract one A, and the second one transforms a B event system into a Symbolic Labelled Transition System. This paper is devoted to define a syntactic transformation that suppresses some variables in M. We show that this function is correct, by proving that A is refined by M, and that a process that combines the syntactic and semantic abstractions significantly reduces the number of proof obligations to prove, and the time cost of abstraction computation.

Key-words: Model Abstraction, Syntactic Abstraction, Refinement

Laboratoire d'Informatique de l'Université de Franche-Comté,
UFR Sciences et Techniques,
16, route de Gray, 25030 Besançon Cedex (France)
Téléphone : +33 (0)3 81 66 64 55 — Télécopie : +33 (0)3 81 66 64 50

Abstraction de modèles B combinant des méthodes syntaxiques et sémantiques

Résumé : Les modèles B constituent une approche intéressante à la fois pour le test à partir de modèles et pour la vérification de propriétés par model-checking. Mais ils sont difficiles à utiliser pour des applications industrielles, en raison de la très grande taille de leur espace d'états. Pour réduire ce nombre d'états, on peut avoir recours à une fonction d'abstraction, qui souvent combine la suppression de variables d'état avec une abstraction des domaines des variables restantes. Ce papier présente un processus d'abstraction assisté par ordinateur qui combine des fonctions d'abstraction syntaxique et sémantique. La première fonction transforme syntaxiquement un système d'événements B de départ M, en un autre abstrait A. La seconde fonction transforme un système d'événements B en un système de transition étiquetté symbolique. Ce papier est dédié à la définition d'une transformation syntaxique qui supprime des variables de M. Nous montrons la correction de cette fonction, en prouvant que A est raffiné par M, et qu'un processus qui combine abstraction syntaxique et abstraction sémantique réduit grandement le nombre d'obligations de preuve à résoudre, ainsi que le temps de calcul de l'abstraction.

Mots-clés : Abstraction de modèle, abstraction syntaxique, raffinement

Laboratoire d'Informatique de l'Université de Franche-Comté,
UFR Sciences et Techniques,
16, route de Gray, 25030 Besançon Cedex (France)
Téléphone : +33 (0)3 81 66 64 55 — Télécopie : +33 (0)3 81 66 64 50

B Model Abstraction Combining Syntactic and Semantics Methods

J. Julliand[1], N. Stouls[2], P.-C. Bué[1], and P.-A. Masson[1]

[1] LIFC, Université de Franche-Comté
16, route de Gray F-25030 Besançon Cedex
{bue, julliand, masson}@lifc.univ-fcomte.fr

[2] Université de Lyon, CITI laboratory, INSA
6 avenue des Arts, F-69621 Villeurbanne Cedex
nicolas.stouls@insa-lyon.fr

Abstract. In a model-based testing approach as well as for the verification of properties by model-checking, B models provide an interesting solution. But 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, often combining state variable elimination and domain abstractions of the remaining variables. This paper presents a computer aided abstraction process that combines syntactic and semantic abstraction functions. The first function syntactically transforms a B event system M into an abstract one A, and the second one transforms a B event system into a Symbolic Labelled Transition System. This paper is devoted to define a syntactic transformation that suppresses some variables in M. We show that this function is correct, by proving that A is refined by M, and that a process that combines the syntactic and semantic abstractions significantly reduces the number of proof obligations to prove, and the time cost of abstraction computation.

Keywords: Model Abstraction, Syntactic Abstraction, Refinement.

1 Introduction

B models are well suited for producing tests of an implementation by means of a *model-based testing* approach [BJK+05,UL06] and to verify dynamic properties by model-checking [LB08]. But model-checking as well as test generation require the models to be finite, and of tractable size. This usually is not the case with industrial applications, and the search for executions instantiated from the model 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. In [BBJM09b], we have proposed and experimented an approach of test generation from abstract models. It appeared that the computation time of the abstraction could be very expensive, as evidenced by an industrial application such as the IAS [GIX04] case study. In other words, we had replaced a problem of search time in a state graph with a problem of proof time. Indeed, computing an abstraction is performed by proving enabledness and reachability conditions on symbolic states [BPS05].

In this paper, we contribute to solve this proof time problem by defining a syntactic abstraction function that does not need proof obligation checking. The function works by suppressing some state variables of a model. When there are domain abstractions on the remaining state variables, we also perform a semantic abstraction that requires proof obligation checking, but it applies to a model that has been syntactically simplified.

In Sec. 2, we define the notions of event system, refined event system and we recall some of the main properties of substitution computation. We also define a Symbolic Labelled Transition System as a B event system abstraction. Section 3 presents the "Electrical System" case study that illustrates our approach. In Sec. 4, we define how our syntactic abstraction function transforms the B predicates and substitutions. We prove that this abstraction is correct in the sense that the source model M refines the generated abstract one A. In this way, the abstraction can be used to verify safety properties and to generate tests. In Sec. 5, we present two processes to compute abstractions. The first one is the semantic abstraction implemented by *GeneSyst* and the second one combines the syntactic abstraction defined in Sec. 4 with the semantic one. In Sec. 6, we compare the two processes on several examples. Section 7 concludes the paper, gives some future research directions and compares our approach to other abstraction methods.

2 Preliminaries

This paper presents an approach for computing an abstraction of an event system such that the abstraction is refined by the source model. We give in this section the background required for reading the paper. We first define general notions about the B method: refinement, primitive forms of substitution, substitution properties, and conjunctive form (CF) of B predicates. Then we summarize the principles of the abstraction of B event systems as considered in [Sto07].

2.1 B Event Systems and Refinement

First introduced by J.-R. Abrial [Abr96a], a B event system defines a closed specification of a system by a set of events. In the remainder of the paper, we will use the following notations to define the event systems: x, y, z are state variables and X is a set of states variables. $\mathcal{P}red$ is the set of B predicates on the variables of X. I, PC, P and P' are in $\mathcal{P}red$: I is an invariant, PC defines properties on the constants, and P and P' are for the other predicates. We use S and S' to denote B generalized substitutions, and E and F to denote B expressions. We distinguish between an event name and its definition. An event has its name ev in a set $\mathcal{E}$. Its definition $ev \mathrel{\widehat{=}} S$ by means of a generalized substitution is in a set Ev.

Many substitutions in the B event systems can be rewritten by means of the five B primitive forms of substitutions of Def. 1. We do not take into account the `PRE` and ";" substitutions as we only consider abstract B event systems.

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

- *single and multiple assignments, denoted as* $x := E$ *and* $x,\ y := E,\ F$
- *substitution with no effect, denoted as skip*
- *guarded substitution, denoted as* $P \Rightarrow S$
- *bounded nondeterministic choice, denoted as* $S[]S'$
- *substitution with local variable, denoted as* $@z.S$*, allowing to express the unbounded non deterministic choice, denoted as* $@z.(P \Rightarrow S)$

Given a substitution S and a post-condition P', we are able to compute the weakest precondition P such that if P is satisfied, then P' is satisfied after the execution of S. The weakest precondition, defined in [Abr96b], is denoted by $[S]P'$. We define correct B event systems in Def. 2.

Definition 2 (Correct B Event System). *A correct B event system is a tuple* $\langle C, PC, X, I, Init, Ev\rangle$ *where:*

- C *is a set of constants,*
- PC *is a predicate defining the constants* C*,*
- X *is a set of state variables,*
- I $(\in \mathcal{P}red)$ *is an invariant predicate over* X*,*
- $Init$ *is a substitution called* initialization*, such that:* $PC \Rightarrow [Init]I$*,*
- Ev *is a set of event definitions in the shape of* $ev_i \mathrel{\widehat{=}} S_i$ *such that the following condition holds for every substitution* S_i*:* $PC \wedge I \Rightarrow [S_i]I$*.*

When we explicitly refer to a given model, we add the name of that model as a subscript to the symbols C, PC, X, I, $Init$ and Ev. I_{M} is for example the invariant of a model M.

In Sec. 4, we will prove that an abstraction A that we compute is refined by its source event system M, and so we give in Def. 3 the definition of the B refinement.

Definition 3 (B Event System Refinement). *Let* A *and* R *be two B event systems.* A *is refined by* R *if:*

- *the initialization satisfy:* $PC_{\mathsf{R}} \wedge PC_{\mathsf{A}} \Rightarrow [Init_{\mathsf{R}}]\neg[Init_{\mathsf{A}}]\neg I_{\mathsf{R}}$*,*
- *any event defined as* $ev \mathrel{\widehat{=}} S_{\mathsf{A}}$ *in* Ev_{A} *and redefined as* $ev \mathrel{\widehat{=}} S_{\mathsf{R}}$ *in* Ev_{R} *satisfies the following condition:* $PC_{\mathsf{R}} \wedge PC_{\mathsf{A}} \wedge I_{\mathsf{A}} \wedge I_{\mathsf{R}} \Rightarrow [S_{\mathsf{R}}]\neg[S_{\mathsf{A}}]\neg I_{\mathsf{R}}$*.*

We use in the remainder of the paper the following main axioms and properties of substitutions:

$$[skip]P \Leftrightarrow P \tag{1}$$

$$[P \Rightarrow S]P' \Leftrightarrow (P \Rightarrow [S]P') \tag{2}$$

$$[S[]S']P \Leftrightarrow [S]P \wedge [S']P \tag{3}$$

$$[@z.S]P \Leftrightarrow \forall z.[S]P \qquad \text{if } z \text{ is not free in } P \tag{4}$$

$$\text{Distributivity: } [S](P \wedge P') \Leftrightarrow [S]P \wedge [S]P' \tag{5}$$

Definition 4 (Conjunctive Form). *A B predicate $P \in Pred$ is in CF when it is a conjunction $p_1 \wedge p_2 \wedge \ldots \wedge p_n$ where any 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:*

- *$E(X)\ r\ E(C)$ or $E(X)\ r\ F(Y)$, where $E(X)$ and $F(Y)$ are B expressions on the sets of variables X and Y, r is a relational operator and $E(C)$ is a constant B expression on the set of constants C,*
- *$\forall z.P$ or $\exists z.P$, where P is a B predicate in CF.*

The aim a putting a predicate in CF is to have the negations applied only to the atomic propositions, so that the non-monotonicity of a predicate transformation function T ($T(\neg P) \neq \neg T(P)$) is not a problem.

2.2 Symbolic Labelled Transition Systems

The event-based semantics of a B event system is defined by its set of execution traces (the set of all its feasible sequences of event executions, starting with the initialization). Hence, a SLTS $\mathcal{A}$ is a semantic abstraction of a B model M if it has the same set of events as M and a set of variables included into that of M (Def. 5), and if every execution trace of M is included into the set of paths of the SLTS $\mathcal{A}$ (Def. 5). Def. 5 is that of [BPS05] where we restrict the labels of the transitions to event names.

Definition 5 (Symbolic Labelled Transition System Associated to a B Model). *A SLTS $\mathcal{A}$ defined by the tuple $\langle X_\mathcal{A}, \mathcal{E}_\mathcal{A}, Q_\mathcal{A}, q_0, Def_\mathcal{A}, T_\mathcal{A}\rangle$ is associated to a B model, if:*

- *$X_\mathcal{A}(\subseteq X)$ is a set of variables, subset of the variables of the B model,*
- *$\mathcal{E}_\mathcal{A}(= \mathcal{E})$ is a set of event names, equal to the one of the B model,*
- *$Q_\mathcal{A}$ is a set of symbolic state names,*
- *$q_0(\in Q_\mathcal{A})$ is the initial state,*
- *$Def_\mathcal{A}(\in Q_\mathcal{A} \rightarrow \mathcal{P}red_\mathcal{A})$ associates a predicate to any symbolic state name,*
- *$T_\mathcal{A}$ is a transition relation ($T_\mathcal{A} \subseteq Q_\mathcal{A} \times \mathcal{E}_\mathcal{A} \times Q_\mathcal{A}$).*

Definition 6 (Semantic Abstraction of a B Model). *A SLTS $\mathcal{A}$ is a semantic abstraction of a B model M, if and only if $\mathcal{A}$ is associated to M according to Def. 5 and every execution trace of M is an existing path over the transitions of $\mathcal{A}$.*

3 Electrical System Example

We describe in this section a B event system that we will use in this paper as a running example to illustrate our proposal.

A device D is powered by one of three batteries B_1, B_2, B_3 as shown in Fig. 1. 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 working of the system must satisfy the three following requirements:

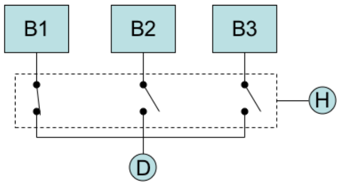

Fig. 1. Electrical System

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

But the batteries are subject to electrical failures. When it occurs to 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 two batteries down at the same time. When there are two batteries down, the requirement Req_3 is relaxed and the clock signal leaves unchanged the switch that is closed.

This system is modeled by means of three variables H, Sw and Bat. H models the clock and takes two values: tic when it asks for a commutation and tac when this commutation has occurred ($H \in \{tic,\ tac\}$). Sw models the state of the three switches by an integer between 1 and 3: $Sw = i$ 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 an electrical failure by a total function ($Bat \in 1..3 \rightarrow \{ok, ko\}$). 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 $Bat(Sw) = ok$:

$$I \mathrel{\widehat{=}} H \in \{tic, tac\} \wedge Sw \in 1..3 \wedge (Bat \in 1..3 \rightarrow \{ok, ko\}) \wedge Bat(Sw) = ok.$$

Notice that the requirement Req_3 is a dynamic property, not formalized in I. The initial state is defined by $Init$ in Fig. 2. The behavior of the system is described by four events, modeled in Fig. 2 with the primitive forms of substitutions:

- **Tic** sends a commutation command,
- **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.

4 Syntactic Abstraction

We consider abstractions obtained by observing only a subset of variables. For instance, to test the electrical system in the particular cases where two batteries

$$
\begin{array}{l}
Init \;\widehat{=}\; H,\ Bat,\ Sw\ :=\ tac,\ \{1 \mapsto ok,\ 2 \mapsto ok,\ 3 \mapsto ok\}, 1 \\
Tic \;\;\widehat{=}\; H = tac \Rightarrow H := tic \\
Com \;\widehat{=}\; \mathrm{card}(Bat \rhd \{ok\}) > 1 \wedge H = tic \Rightarrow \\
\qquad\qquad @ns.(ns \in 1..3 \wedge Bat(ns) = ok \wedge ns \neq Sw \Rightarrow H, Sw := tac, ns) \\
Fail \;\widehat{=}\; \mathrm{card}(Bat \rhd \{ok\}) > 1 \Rightarrow \\
\qquad @nb.(nb \in 1..3 \wedge nb \in \mathrm{dom}(Bat \rhd \{ok\}) \Rightarrow \\
\qquad\qquad nb = Sw \Rightarrow \\
\qquad\qquad\quad @ns.(ns \in 1..3 \wedge ns \neq Sw \wedge Bat(ns) = ok \Rightarrow \\
\qquad\qquad\qquad Sw, Bat := ns, Bat <+ \{nb \mapsto ko\}) \\
\qquad\quad [] nb \neq Sw \Rightarrow Bat := Bat <+ \{nb \mapsto ko\})) \\
Rep \;\;\widehat{=}\; @nb.(nb \in 1..3 \wedge nb \in \mathrm{dom}(Bat \rhd \{ko\}) \Rightarrow Bat := Bat <+ \{nb \mapsto ok\})
\end{array}
$$

Fig. 2. B Specification of the Electrical System

are down, we just have to observe the variable *Bat*. To compute such an abstraction, we define a set of transformation rules that produce a simplified model A. We will prove that A is, by construction, refined by the source model M. Thanks to this property, it is sufficient to verify safety properties on A for them to hold on M. It is also easier to compute test cases from the simplified model.

Let X be a set of variables (or constants) and let T_X be a transformation function of predicates and substitutions according to X, denoted here as $T_X(P)$ or $T_X(S)$. We define the transformation of a B model according to a transformation function T in Def. 7. Then we define a transformation function T which translates a correct model M into a model A that is refined by M (Theorem 2).

Definition 7 (B Event System Transformation). *A correct B event system* $\mathsf{M} = \langle C_\mathsf{M}, PC_\mathsf{M}, X_\mathsf{M}, I_\mathsf{M}, Init_\mathsf{M}, Ev_\mathsf{M}\rangle$ *is transformed as follows, according to a function* T *in the B event system* $\mathsf{A} = \langle C_\mathsf{A}, PC_\mathsf{A}, X_\mathsf{A}, I_\mathsf{A}, Init_\mathsf{A}, Ev_\mathsf{A}\rangle$ *having the same set of event names* $\mathcal{E}_\mathsf{A} = \mathcal{E}_\mathsf{M}$*:*

- $C_\mathsf{A} \subseteq C_\mathsf{M}$*, there is less constants in the abstraction,*
- $PC_\mathsf{A} = T_{C_\mathsf{A}}(PC_\mathsf{M})$*, constants properties are simplified,*
- $X_\mathsf{A} \subseteq X_\mathsf{M}$*, there are less variables in the abstraction,*
- $I_\mathsf{A} = T_{X_\mathsf{A}}(I_\mathsf{M})$*, the invariant is transformed,*
- $Init_\mathsf{A} = T_{X_\mathsf{A}}(Init_\mathsf{M})$*, the initialization is transformed,*
- *for each event* $ev \;\widehat{=}\; S$ *in* Ev_M*,* $ev \;\widehat{=}\; T_{X_\mathsf{A}}(S)$ *exists in* Ev_A*.*

We first present the predicate transformation rules and then we give the generalized substitution rules. We finally prove that, by construction, the initial system is a refinement of the transformed system.

4.1 Predicate Transformation

We define the transformation function T on predicates by induction with the rules given in Fig. 3. Each rule transforms a predicate P w.r.t. the set of variables $X_\mathsf{A}(\subseteq X_\mathsf{M})$ denoted here X. This transformation is denoted as $T_X(P)$. We define a rule R_i for each form of predicate in the conjunctive form (CF) of Def. 4.

An elementary predicate is undetermined when an expression depends on the values of some variables that we do not observe any more (see the rules R_2 and R_4). When all the variables used in the predicate are observed, the transformation leaves it unchanged (see the rules R_1 and R_3). As we want to weaken the predicate so that the events are enabled more often, we replace an undetermined elementary predicate by $true$. Consequently, a predicate $P \wedge P'$ is transformed into P when P' is undetermined, and a predicate $P \vee P'$ is transformed into $true$ when P or P' is undetermined (see the rules R_5 and R_6). These predicates are weakened since $P \wedge P' \Rightarrow P$ and $P \vee P' \Rightarrow true$ are valid formulas. Finally, the transformation of an α-quantified predicate is the transformation of its body w.r.t. the observed variables, augmented with the quantified variable (see the rule R_7). Notice that the quantified variable must not belong to the already observed variables, or else it must be renamed.

R_1	$T_X(E(Y)\ r\ E(C))$	$\widehat{=}\ E(Y)\ r\ E(C)$	if $Y \subseteq X$
R_2	$T_X(E(Y)\ r\ E(C))$	$\widehat{=}\ true$	if $Y \not\subseteq X$
R_3	$T_X(E(Y)\ r\ E(Z))$	$\widehat{=}\ E(Y)\ r\ E(Z)$	if $Y \subseteq X$ and $Z \subseteq X$
R_4	$T_X(E(Y)\ r\ E(Z))$	$\widehat{=}\ true$	if $Y \not\subseteq X$ or $Z \not\subseteq X$
R_5	$T_X(P \vee P')$	$\widehat{=}\ T_X(P) \vee T_X(P')$	
R_6	$T_X(P \wedge P')$	$\widehat{=}\ T_X(P) \wedge T_X(P')$	
R_7	$T_X(\alpha z.P)$	$\widehat{=}\ \alpha z.T_{X\cup\{z\}}(P)$	if $z \notin X$

Fig. 3. CF Predicate Transformation Rules

Similar rules are defined for constants simplification. Due to a lack of space, we do not exhibit these rules in this paper.

For example the predicate invariant I of the electrical system is transformed in $T_{\{Bat\}}(I) \mathrel{\widehat{=}} Bat \in 1..3 \rightarrow \{ok, ko\}$ as in Fig. 4.

$$
\begin{array}{lll}
& T_{\{Bat\}}(H \in \{tic, tac\} \wedge Sw \in 1..3 \wedge Bat \in 1..3 \rightarrow \{ok, ko\} \wedge Bat(Sw) = ok) & \\
= & \begin{array}{l} T_{\{Bat\}}(H \in \{tic, tac\}) \wedge T_{\{Bat\}}(Sw \in 1..3) \\ \wedge\ T_{\{Bat\}}(Bat \in 1..3 \rightarrow \{ok, ko\}) \wedge T_{\{Bat\}}(Bat(Sw) = ok) \end{array} & \text{applying } R_6 \\
= & Bat \in 1..3 \rightarrow \{ok, ko\} & \text{applying } R_1 \text{ and } R_2
\end{array}
$$

Fig. 4. Example of Predicate Transformation

Property 1. Let P be a CF predicate in $\mathcal{P}red$ and let X be a set of variables. $P \Rightarrow T_X(P)$ is valid.

Proof. As we said before, $T_X(P)$ is weaker than P. Indeed, for any predicate P in CF there exist p and p' such that $P = p \wedge p'$ and such that it is transformed either in $p \wedge p'$, or in p, or in p', or in $true$, by application of the transformation rules R_i. For any disjunctive predicate P there exist p and p' such that $P = p \vee p'$ and $p \vee p'$ is transformed either in $p \vee p'$ or in $true$.

4.2 Substitution Transformation

The transformation of substitutions are defined through cases in Fig. 5 on primitive forms of substitutions. We address the problem of the transformation of the

R_8	$T_X(x := E)$	$\widehat{=}\ skip$	if $x \notin X$
R_9	$T_X(x := E)$	$\widehat{=}\ x := E$	if $x \in X$
R_{10}	$T_X(skip)$	$\widehat{=}\ skip$	
R_{11}	$T_X(x,\ y := E,\ F)$	$\widehat{=}\ skip$	if $x \notin X$ and $y \notin X$
R_{12}	$T_X(x,\ y := E,\ F)$	$\widehat{=}\ x := E$	if $x \in X$ and $y \notin X$
R_{13}	$T_X(x,\ y := E,\ F)$	$\widehat{=}\ y := F$	if $x \notin X$ and $y \in X$
R_{14}	$T_X(x,\ y := E,\ F)$	$\widehat{=}\ x,\ y := E,\ F$	if $x \in X$ and $y \in X$
R_{15}	$T_X(P \Rightarrow S)$	$\widehat{=}\ T_X(S)$	if $T_X(P) = true$
R_{16}	$T_X(P \Rightarrow S)$	$\widehat{=}\ T_X(P) \Rightarrow T_X(S)$	elsewhere
R_{17}	$T_X(S[]S')$	$\widehat{=}\ skip$	if $T_X(S) = skip$ and $T_X(S') = skip$
R_{18}	$T_X(S[]S')$	$\widehat{=}\ T_X(S)[]T_X(S')$	elsewhere
R_{19}	$T_X(@z.S)$	$\widehat{=}\ T_X(S)$	if z not free in $T_{X\cup\{z\}}(S)$ and $z \notin X$
R_{20}	$T_X(@z.S)$	$\widehat{=}\ @z.T_{X\cup\{z\}}(S)$	if z free in $T_{X\cup\{z\}}(S)$ and $z \notin X$

Fig. 5. Primitive Substitution Transformation Rules

substitutions assuming that any assignment $x := E$ in the transformed model is such that the expression E is defined only from constant values and from some observed variables, in X when x belongs to X. Therefore, in rules R_8 to R_{14}, we do not transform the expressions E and F. In the context of test generation to which our method is intended, this assumption is not a restriction. It is satisfied when X is computed as a fixpoint, starting from an initial set of variables that is iteratively incremented with the variables that are used in the substitutions that assign variables of X. In Sec. 6, we apply this process to determine the sets of observed variables from the variables used in test purposes.

Intuitively, a substitution is abstracted by *skip* when it does not modify variables from X. The assignment of a variable is replaced by *skip* (i.e. no effect) if the variable is not observed (see rules R_8, R_{11}), otherwise it is left unchanged (see rules R_9, R_{12}, R_{13}, R_{14}). The substitution with no effect is unchanged (see rule R_{10}). The rules R_{15} and R_{16} transform the guarded substitution $P \Rightarrow S$. The substitution becomes $T_X(S)$ when $T_X(P)$ is undetermined ($= true$), so that $T_X(S)$ is enabled more often than S. This is also the case in rule R_{16} since $T_X(P)$ is weaker than P from Prop. 1. The bounded non deterministic choice $S [] S'$ becomes $T_X(S) [] T_X(S')$ (see rule R_{18}) so that $T_X(S)$ and $T_X(S')$ are enabled more often than S and S'. In the case where both are transformed into *skip* (see rule R_{17}), the substitution becomes *skip*. The quantified substitution is transformed only when the quantified variable z is not an observed variable in X. It is transformed into $T_X(S)$ when z is not free in $T_{X\cup\{z\}}(S)$ (see rule R_{19}) and into $@z.T_{X\cup\{z\}}(S)$ elsewhere (see rule R_{20}).

Theorem 1. *Let I be a CF invariant of a correct B event system, let S be a substitution and let X be a set of observed variables. The transformation rules R_8 to R_{20} are such that S refines $T_X(S)$ according to the invariant I.*

Theorem 2. *Let T be the transformation defined in Fig. 5, let X be a set of observed variables, and let A be an abstraction of an event system M defined according to Def. 7. A is refined by M in the sense of Def. 3.*

Theorem 1 establishes that any substitution S refines its transformation $T_X(S)$ for a given set of observed variables X. The associated proof is given in Appendix A. The theorem 2 establishes that a B abstract system obtained by the transformation of Def. 7 is refined by a B event system M, using the transformation rules defined in Fig. 3 and Fig. 5.

Proof (of theorem 2). This is a direct consequence of theorem 1 and Def. 7 since the substitution $Init_{\mathsf{A}} \mathrel{\widehat{=}} T_X(Init_{\mathsf{M}})$ is refined by $Init_{\mathsf{M}}$ and for any event $ev \mathrel{\widehat{=}} S_{\mathsf{M}}$, the substitution $S_{\mathsf{A}} \mathrel{\widehat{=}} T_X(S_{\mathsf{M}})$ is refined by S_{M}.

The electrical system is transformed as shown in Fig. 6 for the set of observed variables $\{Bat\}$. It is a correct B event system. But with the method, there is a risk for the syntactically abstracted systems not to satisfy their invariants $T_X(I_{\mathsf{M}})$, when a property on the observed variables depends on the eliminated ones. This has no consequence on the soundness of the verification of safety properties and of the test generation, but the verification may fail, and some generated tests could be impossible to instantiate. Notice that this happened to none of our eight abstracted systems of Sec. 6: they were all correct. Also notice that it is always possible to get a correct abstracted model by weakening the invariant, for instance by reducing it to typing properties.

$$
\begin{array}{lcl}
Init & \widehat{=} & Bat := \{1 \mapsto ok,\ 2 \mapsto ok,\ 3 \mapsto ok\} \\
Tic & \widehat{=} & skip \\
Com & \widehat{=} & \mathrm{card}(Bat \rhd \{ok\}) > 1 \Rightarrow @ns.(ns \in 1..3 \wedge Bat(ns) = ok \Rightarrow skip) \\
Fail & \widehat{=} & \mathrm{card}(Bat \rhd \{ok\}) > 1 \Rightarrow \\
 & & \quad @nb.(nb \in 1..3 \wedge nb \in \mathrm{dom}(Bat \rhd \{ok\}) \Rightarrow Bat := Bat <\!\!+ \{nb \mapsto ko\}) \\
Rep & \widehat{=} & @nb.(nb \in 1..3 \wedge nb \in \mathrm{dom}(Bat \rhd \{ko\}) \Rightarrow Bat := Bat <\!\!+ \{nb \mapsto ok\})
\end{array}
$$

Fig. 6. B Syntactically Abstracted Specification of the Electrical System

5 Abstraction Process

In [BBJM09b] we have introduced a test generation method based on a semantic abstraction of a B model (see Fig. 7/Process A). The abstraction is computed according to a test purpose. The idea is to observe the state variables that are modified by the operations activated by the test purpose. The domain of the observed variables can be abstracted into a few subdomains. For example, a natural integer n can be abstracted into subdomains $n = 0$ and $n > 0$.

The two main drawbacks of this process are its time cost and the proportion of proof obligations (PO) not automatically proved. Indeed, the semantic abstraction is based on the proof of the feasibility of the transitions between two symbolic states. Each unproved PO adds a transition that is possibly unfeasible. Hence we propose to use a syntactic abstraction in addition to the semantic one. In Fig. 7/Process B, we describe a complete abstraction process in which we combine a syntactic abstraction that eliminates some variables (see Sec. 4), with a semantic abstraction computed by *GeneSyst* [Sto07] that projects the domain of the observed variables onto abstract domains (see Sec. 2.2).

The two processes shown in Fig. 7 are not commutative, which means that the abstract models $\mathcal{A}_{\mathsf{A}}$, computed by combining a syntactic and a semantic abstraction, and $\mathcal{A}_{\mathsf{M}}$, computed directly from the behavioral model M, are incomparable. Both processes add unfeasible transitions, but not always the same ones, as is discussed in Sec. 6.2. Nevertheless, the process is correct, since both $\mathcal{A}_{\mathsf{A}}$ and $\mathcal{A}_{\mathsf{M}}$ are refined by the source behavioral model M.

The main advantage of the process including syntactic abstraction w.r.t. the completely semantic one is the reduction of the number of PO (denoted as $\#PO$) computed by *GeneSyst*. Let $\#e$ be the number of events in the behavioral model M and let $\#s$ be the number of symbolic states. The number of PO in the worst case is defined as: $\#PO = \#s + \#s \times \#e + \#s^2 \times \#e$. There is one PO per symbolic state to compute the initial states, one PO per symbolic state for the enabledness of each event, and one PO for the reachability per pair of symbolic states for each event. The number of generated PO of the syntactically abstracted model A depends on the structure of events. We consider four categories:

- $\#e_{skip}$ is the number of events simplified as *skip*,
- $\#e_{gskip}$ is the number of guarded events simplified as $P \Rightarrow skip$,
- $\#e_{true}$ is the number of events simplified as $true \Rightarrow S$ whose guard is $true$,
- and $\#e_{gs}$ is the number of simplified guarded events whose substitution is different from *skip* and the guard P is different from $true$.

As the symbolic states are, by construction, mutually disjointed in our process, the number of PO for each form of event can be reduced. An event reduced to *skip* makes a reflexive transition on any symbolic state, with no need to prove any PO. Any event reduced to $P \Rightarrow skip$ makes a reflexive transition on any symbolic state in which it is enabled. The other events generate the same PO, but the events that are reduced with a *true* guard generate no PO for the enabledness. Finally, the number of PO in the worst case is defined as:

$$\#PO = \#s + \#s \times (\#e_{gskip} + \#e_{gs}) + \#s^2 \times (\#e_{true} + \#e_{gs})$$

Moreover, the remaining PO are simplified because the abstract events and the abstract invariant are simplified. Notice that the bigger the behavioral model is, the more the simplifications are important, because the ratio of the number of observed variables to the total number of state variables is small. For example, the electrical system in Fig. 1 abstracted on $\{Bat\}$ in Fig. 6 gives the following worst-case results: $\#PO_{\mathcal{A}_{\mathsf{M}}} = 9 + 4 \times 9 + 4 \times 9^2 = 369$ and $\#PO_{\mathcal{A}_{\mathsf{A}}} = 9 + 9 \times (1 + 2) + 9^2 \times (0 + 2)) = 198$ for $\#e = 4$, $\#s = 9$, $\#e_{skip} = 1$ (event Tic), $\#e_{gskip} = 1$ (event Com), $\#e_{true} = 0$ and $\#_{gs} = 2$ (events $Fail$ and Rep).

6 Experimental Results

We have applied our method to four case studies. They are of increasing size, and are various cases of reactive systems: the electrical system[1] (Electr. [Cle01]), a

[1] The 100 lines length of the model, in Table 1, refer to a "verbose" version of the model, much more readable than our version of Fig. 2.

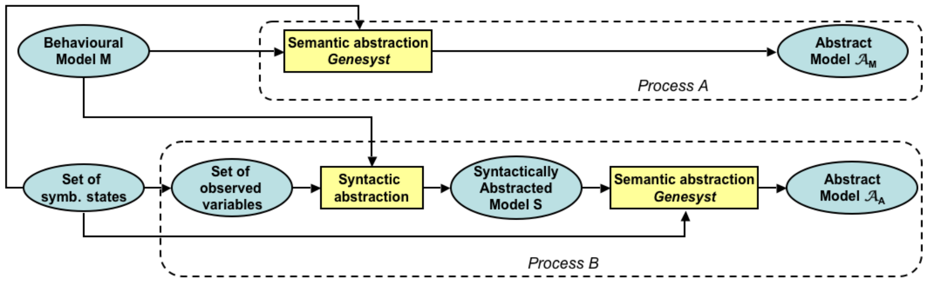

Fig. 7. Abstraction Process

reverse phone book service (Qui-Donc [UL06]), an automatic conveying system (Robot [BBJM09a]) and an electronic purse (DeMoney [MM02]). Each one is abstracted w.r.t. two sets of observed variables . In [BBJM09b], we explain how to extract the set of observed variables by a static analysis of a test purpose.

In Sec. 6.1 we present an experimental evaluation of the syntactic abstraction process. Then, in Sec. 6.2 we compare $\mathcal{A}_{\mathsf{M}}$ with $\mathcal{A}_{\mathsf{A}}$ respectively computed by the semantic abstraction process or by its combination with the syntactic one.

6.1 Syntactic Abstraction

Table 1 gives some metrics about case studies, while Table 2 indicates metrics of the syntactically abstracted models. Symbols "♯", "Ev.", "Enum.", "Var.", "Int.", "Pot.", "Symb.", "Th.", "Pr." and "Trans." stand respectively for *number of*, *Events*, *Enumerated*, *Variables*, *Integers*, *Potential*, *Symbolic*, *Theoretical*, *Practical* and *Transitions*. For example, the Robot defined by 6 variables and 9 events is abstracted w.r.t. two sets of respectively 3 and 4 observed variables. In the first case, one event becomes *skip*, four events become $P \Rightarrow skip$ and the four remaining ones are simplified as $P \Rightarrow S$. There are 6 abstracted states. 263 PO are generated by *GeneSyst* to abstract the original (i.e. not syntactically simplified) specification, while only 143 PO are generated when it has first been syntactically simplified.

Case Study	♯Ev.	♯Enum. Var.	♯Int. var.	♯B lines	♯Pot. states
Robot	9	6	0	100	384
QuiDonc	4	3	0	170	13
Electr.	4	2	1	100	36
DeMoney	11	3	6	330	10^{30}

Table 1. Some Metrics about Case Studies

Depending on the examples, we can see that from 50% up to 90% of the events are simplified as *skip*, $P \Rightarrow skip$ or $true \Rightarrow S$. The direct observable result of syntactic abstraction is a reduction of the number of generated PO, from 10% up to 60% from a theoretical point of view, and from 40% up to 60% from a practical point of view. Also notice that the simplification reduces from 10% up to 50% the number of lines of the model.

Case Study	#Enum. Var.	#Int. Var.	#B Lines	#Pot. States	$\#e_{skip}$	$\#e_{gskip}$	$\#e_{true}$	$\#e_{gs}$	#Symb. States	#PO $\mathcal{A}_{\mathsf{M}}$		#PO $\mathcal{A}_{\mathsf{A}}$	
										Th.	Pr.	Th.	Pr.
Robot	3	0	90	48	1	4	0	4	6	384	263	198	143
	4	0	90	144	0	4	0	5	8	656	402	400	242
QuiDonc	2	0	160	16	0	0	2	2	5	125	71	115	89
	2	0	160	16	0	0	2	2	6	174	89	162	103
Electr.	0	1	50	5	1	1	0	2	2	26	26	16	16
	1	0	40	2	2	0	0	2	2	26	21	14	9
DeMoney	0	1	140	65536	4	0	6	1	3	135	116	69	68
	2	1	150	11	6	0	4	1	9	999	737	423	331

Table 2. Some Metrics about Syntactically Abstracted Case Studies

6.2 Semantic Abstraction

Table 3 gives metrics about the semantic abstractions computed either directly from the behavioral models (process A in Fig. 7), or from their syntactic abstractions (process B in Fig. 7). We can see that in the worst case, there are from twice up to seven times less PO to compute once the model has been syntactically simplified. In practice on our examples, there is between 1.8 and 2.3 times less PO to compute. The semantic abstraction computation takes from twice up to five times less time. There are from twice up to seven times less unproved PO. Finally, there are six cases out of eight where the abstraction $\mathcal{A}_{\mathsf{A}}$ is more precise than $\mathcal{A}_{\mathsf{M}}$ in the sense that it has less transitions, due to the reduction of the number of unproved PO. In these six cases, the set of traces of $\mathcal{A}_{\mathsf{A}}$ is included in the set of traces of $\mathcal{A}_{\mathsf{M}}$. In the two other cases, there is no inclusion at all. The simplification of the invariant in the syntactic abstraction makes that some transitions not enabled in $\mathcal{A}_{\mathsf{M}}$ are enabled in $\mathcal{A}_{\mathsf{A}}$, and the event simplification makes that some transitions enabled in $\mathcal{A}_{\mathsf{M}}$ are not enabled in $\mathcal{A}_{\mathsf{A}}$. Thus the set of traces cannot be compared.

The method is of poor interest on the smallest example (QuiDonc). But, as evidenced by DeMoney, its efficiency grows with the size of the examples, in terms of gain of the abstraction computation time, of reduction of the number of unproved PO and of precision of the abstraction.

Case Study	$\mathcal{A}_{\mathsf{M}}$				$\mathcal{A}_{\mathsf{A}}$				Traces inclusion
	#Trans.	#Not En. Trans.	#Reach. States	Time (s)	#Trans.	#Not En. Trans.	# Reach. States	Time (s)	
Robot	42	5	6	64	36	0	6	35	$\mathcal{A}_{\mathsf{A}} \subseteq \mathcal{A}_{\mathsf{M}}$
	51	0	8	76	50	0	8	49	$\mathcal{A}_{\mathsf{A}} \subseteq \mathcal{A}_{\mathsf{M}}$
QuiDonc	20	2	5	19	25	7	5	21	$\mathcal{A}_{\mathsf{A}} \nsubseteq \mathcal{A}_{\mathsf{M}}$
	25	2	5	21	29	6	5	23	$\mathcal{A}_{\mathsf{A}} \nsubseteq \mathcal{A}_{\mathsf{M}}$
Electr.	13	5	2	7	13	5	2	5	$\mathcal{A}_{\mathsf{A}} \subseteq \mathcal{A}_{\mathsf{M}}$
	7	0	2	5	7	0	2	2	$\mathcal{A}_{\mathsf{A}} \subseteq \mathcal{A}_{\mathsf{M}}$
DeMoney	38	5	3	189	38	5	3	38	$\mathcal{A}_{\mathsf{A}} \subseteq \mathcal{A}_{\mathsf{M}}$
	92	2	7	226	89	2	7	74	$\mathcal{A}_{\mathsf{A}} \subseteq \mathcal{A}_{\mathsf{M}}$

Table 3. Abstraction Comparison

7 Conclusion, Related Works and Further works

We have presented in the B framework a method for abstracting an event system by elimination of some state variables. We have proved that such abstractions are refined by the source model. This is useful for verifying properties and generating tests.

The main advantage of our method is that it first performs syntactic transformations, which reduces the number of PO generated and facilitates the proof of the remaining PO. 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 needs to be confirmed by experiments on industrial size applications.

Many other works define model abstraction methods to verify properties. 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 *a priori* fixed predicates. In contrast, our method firstly 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 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 do not compute any weakest precondition whereas the approach in [NK00] does, which possibly introduces infinitely often new predicates.

The method that we have presented is correct, but may sometimes produce inaccurate over-approximations due to a too strong abstraction of the invariant. We think that rules could be improved to get a finer approximation. 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 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 $\mathcal{A}_{\mathsf{A}}$, so that the following semantic abstraction would be more accurate. Such rules should prevent the addition of transitions in the QuiDonc abstraction $\mathcal{A}_{\mathsf{A}}$ w.r.t. $\mathcal{A}_{\mathsf{M}}$.

We think that extending the test generation method introduced in [BBJM09b] by using a combination of syntactic and semantic abstractions will improve the method, because the abstraction is more accurate when there are less unproved PO. But, what occurs if the abstraction is less accurate ?

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.

[BBJM09a] 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.

[BBJM09b] F. Bouquet, P.-C. Bué, J. Julliand, and P.-A. Masson. Test generation based on abstraction and dynamic selection criteria. Research Report RR2009-02, Laboratoire d'Informatique de l'Université de Franche Comté, September 2009.

[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*. 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.

[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.

[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.

[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.

[Cle01] Clearsy. System engineering Atelier B, version 3.6. Technical report, http://www.atelierb.societe.com, 2001.

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

[GIX04] GIXEL. *Common IAS Platform for eAdministration*, 1.01 premium edition, 2004.

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

[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.

[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.

[Sto07] N. Stouls. *Systèmes de transitions symboliques et hiérarchiques pour la conception et la validation de modèles B raffinés*. Thesis, U. Grenoble, 2007.

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

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

A Proof of Theorem 1

Proof. The refinement theory as defined in B [Abr96b], requires that variable sets from abstraction and variable sets from refinement are disjoint. If a variable x is preserved through the refinement process, then it has to be renamed, i.e. $x_{renamed}$, and associated by a gluing invariant, i.e. $x = x_{renamed}$. In order to prove the correctness of the refinement, we introduce the $\mathsf{Ren}()$ function, which renames every variable from a substitution or a predicate. Hence, the invariant I_{A} abstracted from I_{M} and the substitution S_{A} abstracted from any S_{M} are defined as follows:

$$I_{\mathsf{A}} \mathrel{\hat=} \mathsf{Ren}(T_X(I_{\mathsf{M}})) \qquad\qquad S_{\mathsf{A}} \mathrel{\hat=} \mathsf{Ren}(T_X(S_{\mathsf{M}}))$$

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

$$PC_{\mathsf{A}} \wedge PC_{\mathsf{M}} \wedge I_{\mathsf{A}} \wedge I_{\mathsf{M}} \wedge I_{\mathsf{G}} \Rightarrow [S_{\mathsf{M}}]\neg[S_{\mathsf{A}}]\neg(I_{\mathsf{M}} \wedge I_{\mathsf{G}}) \tag{6}$$

where I_{G} is the gluing invariant $I_{\mathsf{G}} \mathrel{\hat=} \bigwedge_{x_i \in X}(x_i = \mathsf{Ren}(x_i))$. In order to prove formula (6), it is sufficient to establish that the two following formulas hold:

$$PC_{\mathsf{A}} \wedge PC_{\mathsf{M}} \wedge I_{\mathsf{A}} \wedge I_{\mathsf{M}} \wedge I_{\mathsf{G}} \Rightarrow [S_{\mathsf{M}}]\neg[S_{\mathsf{A}}]\neg I_{\mathsf{M}} \tag{7}$$

$$PC_{\mathsf{A}} \wedge PC_{\mathsf{M}} \wedge I_{\mathsf{A}} \wedge I_{\mathsf{M}} \wedge I_{\mathsf{G}} \Rightarrow [S_{\mathsf{M}}]\neg[S_{\mathsf{A}}]\neg I_{\mathsf{G}} \tag{8}$$

Since free variable sets from I_{A} and I_{M} are strictly disjoint, (7) 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 (8) to prove theorem 1. The proof is inductive on the five primitive forms of substitutions. We make a case analysis for each rule in Fig. 5. We use Prop. 1 of Sec. 4.1 and axioms (1 to 5) defined in Sec. 2.1.

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

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

Rule $R_{\mathbf{8}}$ $S_{\mathsf{A}} \mathrel{\hat=} \mathit{skip}$ *when* $x \notin X$

is $\mathit{Hyps} \Rightarrow [x := E]\neg[\mathit{skip}]\neg I_{\mathsf{G}}$ valid ?

It is valid, according to (1), since x is not free in I_{G}.

Rule $R_{\mathbf{9}}$ $S_{\mathsf{A}} \mathrel{\hat=} \mathsf{Ren}(x) := \mathsf{Ren}(E)$ *when* $x \in X$

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

It is valid since Rule $R_{\mathbf{9}}$ is the identity.

Case $S_{\mathsf{M}} \mathrel{\hat=} \mathit{skip}$

Rule $R_{\mathbf{10}}$ $S_{\mathsf{A}} \mathrel{\hat=} \mathit{skip}$

$\mathit{Hyps} \Rightarrow [\mathit{skip}]\neg[\mathit{skip}]\neg I_{\mathsf{G}}$ is obviously valid according to (1).

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

Rules $R_{\mathbf{11}}$ **to** $R_{\mathbf{14}}$ proofs are similar to the first case.

Case $S_{\mathsf{M}} \mathrel{\hat=} P \Rightarrow S$

Rule $\mathbf{R_{15}}$ $S_{\mathsf{A}} \mathrel{\hat=} \mathsf{Ren}(T_X(S))$ *when* $T_X(P) = \mathit{true}$

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

$\equiv \mathit{Hyps} \wedge P \Rightarrow [S]\neg[\mathsf{Ren}(T_X(S))]\neg I_{\mathsf{G}}$ – applying (2)

It is valid w.r.t. the induction hypothesis $\mathit{Hyps} \Rightarrow [S]\neg[\mathsf{Ren}(T_X(S))]\neg I_{\mathsf{G}}$

Rule $\mathbf{R_{16}}$ $S_{\mathsf{A}} \mathrel{\hat=} \mathsf{Ren}(T_X(P)) \Rightarrow \mathsf{Ren}(T_X(S))$ *elsewhere*

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

$\equiv \mathit{Hyps} \Rightarrow P \Rightarrow [S](\mathsf{Ren}(T_X(P)) \wedge \neg[\mathsf{Ren}(T_X(S))]\neg I_{\mathsf{G}})$ – applying (2)

$$\equiv \begin{cases} (R_{16}.1)\ (\mathit{Hyps} \wedge P \Rightarrow [S]\mathsf{Ren}(T_X(P))) \\ \wedge\ (R_{16}.2)\ (\mathit{Hyps} \wedge P \Rightarrow [S]\neg[\mathsf{Ren}(T_X(S))]\neg I_{\mathsf{G}}) \end{cases} \quad \text{– applying (5)}$$

According to Prop 1, $(R_{16}.1)$ holds since S variables are not free in $\mathsf{Ren}(T_X(P))$ and since I_{G} is in *Hyps*. Proof of $(R_{16}.2)$ is similar to proof of (R_{15}).

Case $S_{\mathsf{M}} \hat{=} S \; [] \; S'$

Rule $\mathbf{R_{17}}$ $S_{\mathsf{A}} \hat{=} \mathit{skip}$ *when* $T_X(S) = \mathit{skip}$ *and* $T_X(S') = \mathit{skip}$

is $\mathit{Hyps} \Rightarrow [S \; [] \; S']\neg[\mathit{skip}]\neg I_{\mathsf{G}}$ valid ?

It is valid since S variables are not free in I_{G}.

Rule $\mathbf{R_{18}}$ $S_{\mathsf{A}} \hat{=} \mathsf{Ren}(T_X(S))[]\mathsf{Ren}(T_X(S'))$ *elsewhere*

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

$\equiv \mathit{Hyps} \Rightarrow [S \; [] \; S'](\neg[\mathsf{Ren}(T_X(S))]\neg I_{\mathsf{G}} \vee \neg[\mathsf{Ren}(T_X(S'))]\neg I_{\mathsf{G}})$ – applying (3)

$\equiv \begin{cases} \quad (\mathit{Hyps} \Rightarrow [S](\neg[\mathsf{Ren}(T_X(S))]\neg I_{\mathsf{G}} \vee \neg[\mathsf{Ren}(T_X(S'))]\neg I_{\mathsf{G}})) \\ \wedge (\mathit{Hyps} \Rightarrow [S'](\neg[\mathsf{Ren}(T_X(S))]\neg I_{\mathsf{G}} \vee \neg[\mathsf{Ren}(T_X(S'))]\neg I_{\mathsf{G}})) \end{cases}$ – applying (3)

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

1. $\mathit{Hyps} \Rightarrow [S]\neg[\mathsf{Ren}(T_X(S))]\neg I_{\mathsf{G}}$,
2. $\mathit{Hyps} \Rightarrow [S']\neg[\mathsf{Ren}(T_X(S'))]\neg I_{\mathsf{G}}$.

Case $S_{\mathsf{M}} \hat{=} @z.S$

Rule $\mathbf{R_{19}}$ $S_{\mathsf{A}} \hat{=} \mathsf{Ren}(T_X(S))$ *when* z *is not free in* $T_{X \cup \{z\}}(S)$ *and* $z \notin X$

is $\mathit{Hyps} \Rightarrow [@z.S]\neg[\mathsf{Ren}(T_X(S))]\neg I_{\mathsf{G}}$ valid ?

$\equiv \mathit{Hyps} \Rightarrow \forall z.[S]\neg[\mathsf{Ren}(T_X(S))]\neg I_{\mathsf{G}}$ – applying (4)

$\equiv \mathit{Hyps} \Rightarrow \forall z.([S]\neg[\mathsf{Ren}(T_X(S))]\neg I_{\mathsf{G}})$ – since $z \notin X$

valid because implied by the induction hypothesis.

Rule $\mathbf{R_{20}}$ $S_{\mathsf{A}} \hat{=} \mathsf{Ren}(@z.T_{X \cup \{z\}}(S))$ *when* z *is free in* $T_{X \cup \{z\}}(S)$ *and* $z \notin X$

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

$\equiv \mathit{Hyps} \Rightarrow \forall z.[S]\neg\forall\mathsf{Ren}(z).[\mathsf{Ren}(T_{X \cup \{z\}}(S))]\neg I_{\mathsf{G}}$ – applying (4)

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

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

Hence, Theorem 1 holds.

L I F C

Laboratoire d'Informatique de l'université de Franche-Comté
UFR Sciences et Techniques, 16, route de Gray - 25030 Besançon Cedex (France)

LIFC - Antenne de Belfort : IUT Belfort-Montbéliard, rue Engel Gros, BP 527 - 90016 Belfort Cedex (France)
LIFC - Antenne de Montbéliard : UFR STGI, Pôle universitaire du Pays de Montbéliard - 25200 Montbéliard Cedex (France)

http://lifc.univ-fcomte.fr