Introduction to the Integration of SMT-Solvers in Rodin*

David Déharbe[1], Pascal Fontaine[2], Yoann Guyot[3], and Laurent Voisin[4]

[1] Federal University of Rio Grande do Norte, Brazil
[2] University of Lorraine, Loria, Inria, France
[3] Cetic, Belgium
[4] Systerel, France

Abstract. Event-B is a system specification approach based on set theory, integer arithmetics and refinement, supported by the Rodin platform, an Eclipse-based IDE. Event-B development requires the validation of proof obligations, often with set operators. This approach relies on the existence of theorem proving techniques that handle such set constructs. One such technique is known as Satisfiability Modulo Theory (SMT) solving. However, there is no direct support for set-theory operators in the standard logics supported by SMT solvers.
We have developed an *ad hoc* encoding approach enabling the verification of Event-B proof obligations using SMT-solving technology. Its full description is available in [12], and we present here the general principles.

Keywords: Event-B · SMT · Rodin

1 Introduction

Event-B [2] is a notation and a theory for formal system modeling and refinement, based on first-order logic, typed set theory and integer arithmetic. The Rodin platform [7] is an integrated design environment for Event-B. It is based on the Eclipse framework [16], and has an extensible architecture, where features can be integrated and updated by means of plug-ins. Event-B models must be proved consistent; for this purpose, Rodin generates proof obligations that need to be proved valid.

The proof obligations are represented internally as sequents, the sequent calculus providing the basis of the verification framework. Rodin applies proof rules to a sequent to produce zero, one or more new, usually simpler, sequents. A proof rule producing no sequent is a *discharging rule*. The goal of the verification is to build a proof tree corresponding to the application of the proof rules, where

* This work is partly supported by the project ANR-13-IS02-0001-01, the STIC AmSud MISMT, CAPES grant BEX 2347/13-0, CNPq grants 308008/2012-0 and 573964/2008-4 (National Institute of Science and Technology for Software Engineering—INES, `www.ines.org.br`), and EU funded project ADVANCE (FP7-ICT-287563).

the leaves are discharging rules. In practice, the proof rules are generated by so-called *reasoners*. A reasoner is a plug-in that can either be standalone or use existing verification technologies through third-party tools.

The verification shall be automated, informative and trustable. Although full automation is theoretically impossible, and eventually interactive theorem proving needs to be supported, fine-tuning existing automatic verification techniques can yield significant improvements in the productivity (and usability) of the engineers applying this method. Moreover, when a model is edited and modified, large parts of the proof can be preserved if the precise facts used to validate each proof obligation are recorded. So it is important that the reasoners inform such sets of relevant facts, that are then used to automatically construct new proof rules to be stored and tried for after model changes. A bonus is that other sequents (valid for the same reason) may be discharged by these rules without requiring another call to the reasoner. When reasoners inform counter-examples of failed proof obligations, this is valuable information to the user to improve the model and the invariants. We must also be aware that, when a prover is used, either the tool itself or its results need to be certified; otherwise the confidence in the formal development is jeopardized.

This paper presents a SMT plug-in for Rodin, enabling a verification approach that has the potential of fulfilling such requirements. Indeed, SMT solvers can *automatically* handle large formulas of first-order logic with respect to some background theories, or a combination thereof, such as different fragments of arithmetic (linear and non-linear, integer and real), arrays, bit vectors, etc. They have been employed successfully to handle proof obligations with tens of thousands of symbols stemming from software and hardware verification. This paper presents a new approach to encode set-theory formulas in first-order logic. The full details of this approach are presented in [12].

Outline of the paper. Sections 2 and 3 introduce briefly the technical background of this paper, namely SMT-solving and Event-B. The encoding approach for the verification of sequents with set constructs using existing SMT-solvers is then presented in section 4. In section 5, we report the results of an experimental appraisal of this approach. We then present our conclusions in section 6.

Throughout the paper, formulas are expressed using the Event-B syntax [2], and sentences in SMT-LIB are typeset using a typewriter font.

2 SMT Solvers

A SMT solver is basically a decision procedure for quantifier-free formulas in a rich language coupled with an instantiation module that handles the quantifiers in the formulas by grounding the problem. For quantified logic, SMT solvers are of course not decision procedures anymore, but they work well in practice if the necessary instances are easy to find and not too numerous. We refer to [4] for more information about the techniques described in this section and SMT solving in general.

The SMT-LIB initiative [5] provides a standard input language of SMT solvers and a command language defining a common interface to interact with SMT solvers. Some solvers (e.g. Z3 [9] and veriT [6]) implement commands that generate a comprehensive proof for validated formulas; such a proof can then be verified by a *trusted* proof checker [3]. In the longer term, besides automation and information, trust may be obtained using a centralized proof manager.

Historically, the first goal of SMT solvers was to provide efficient decision procedures for expressive languages, beyond pure propositional logic. The SMT-LIB includes a number of pre-defined such "logics", which the existing solvers handle at least in part. *But there is not yet an agreed-upon theory on sets in the SMT-LIB.* To apply a SMT solver to a proof obligation with set constructs, one could include in this proof obligation an axiomatization of these operators. However such axiomatization often include quantified formulas which happen to be problematic in practice for SMT solvers. Another approach is to find an encoding of the set operators in one of the logics defined in the SMT-LIB. We shall present one such encoding in this paper.

Additionally to the satisfiability response, it is possible, in case of an unsatisfiable input, to ask for an *unsatisfiable core*. It may indeed be very valuable to know which hypotheses are necessary to prove a goal in a verification condition. For instance, the sequent (1) discussed in Section 3 and translated into the SMT input in Figure 4 is valid independently of the assertion labeled `grd1`; the SMT input associates labels to the hypotheses, guards, and goals, using the reserved SMT-LIB annotation operator `!`. A solver implementing the SMT-LIB unsatisfiable core feature could thus return the list of hypotheses used to validate the goals. In the case of the example in Figure 4, the guard is not necessary to prove unsatisfiability, and would therefore not belong to a good unsatisfiable core.

Recording unsatisfiable cores for comparison with new proof obligations is particularly useful in our context. Indeed, users of the Rodin platform will want to modify their models and their invariants, resulting in a need of revalidating proof obligations mostly but not fully similar to already validated ones. If the changes do not impact the relevant hypotheses and goal of a proof obligation, comparison with the (previous) unsatisfiable core will discharge the proof obligation and the SMT solver will not need to be run again. Also a same unsatisfiable core is likely to discharge similar proof obligations, for instance generated for a similar transition, but differing for the guard.

3 Event-B

We introduce the Event-B notation through excerpts of a simple example: a model of a simple job processing system consisting of a job queue and several active jobs. Set *JOBS* represents the jobs. The state of the model has two variables: *queue* (the jobs currently queued) and *active* (the jobs being processed). This state is constrained by the following invariants:

`inv1` : $active \subseteq JOBS$ (typing)
`inv2` : $queue \subseteq JOBS$ (typing)

`inv3` : $\mathit{active} \cap \mathit{queue} = \varnothing$ (a job can not be both active and queued)

One of the events is when a job leaves the queue to become active:

```
Event  SCHEDULE ≙                    (some queued job j becomes active)
    any
        j
    where
        grd1 : j ∈ queue             (the job j is in the queue)
    then
        act1 : active := active ∪ {j}    (the job becomes active)
        act2 : queue := queue \ {j}      (the job is removed from the queue)
    end
```

The invariant `inv3` is preserved by this event, if the following sequent is valid:

$$\texttt{inv1}, \texttt{inv2}, \texttt{inv3}, \texttt{grd1} \vdash \underbrace{(\mathit{active} \cup \{j\})}_{\mathit{active'}} \cap \underbrace{(\mathit{queue} \setminus \{j\})}_{\mathit{queue'}} = \varnothing\ . \tag{1}$$

Thus, the corresponding proof obligation consists in showing that the following formula is unsatisfiable:

$$\mathit{active} \subseteq \mathit{JOBS}\ \wedge \mathit{queue} \subseteq \mathit{JOBS}\ \wedge \mathit{active} \cap \mathit{queue} = \varnothing\ \wedge j \in \mathit{queue}\ \wedge \\ \neg((\mathit{active} \cup \{j\}) \cap (\mathit{queue} \setminus \{j\}) = \varnothing)\ .$$

A typical development in Event-B contains hundreds or even thousands of such proof obligations, and often some share many common sub-formulas.

4 Translating Event-B to SMT

Figure 1 gives a schematic view of the cooperation framework between Rodin and the SMT solver. Within the Rodin platform, each proof obligation is represented as a sequent, i.e. a set of hypotheses and a conclusion. These sequents are discharged using Event-B proof rules. Our strategy to prove an Event-B sequent is to build an SMT formula, call an SMT solver on this formula, and, on success, introduce a new suitable proof rule. This strategy is presented as a *tactic* in the Rodin user interface. Since SMT solvers answer the *satisfiability* question, it is necessary to take the negation of the sequent (to be validated) in order to build a formula to be refuted by the SMT solver. If the SMT solver does not implement unsatisfiable core generation, the proof rule will assert that the full Event-B sequent is valid (and will only be useful for that specific sequent). Otherwise an unsatisfiable core — i.e., the set of facts necessary to prove that the formula is unsatisfiable — is supplied to Rodin, which will extract a stronger Event-B proof rule containing only the necessary hypotheses. This stronger proof rule will hopefully be applicable to other Event-B sequents. If, however, the SMT solver is not successful, the application of the tactic has failed and the proof tree remains unchanged.

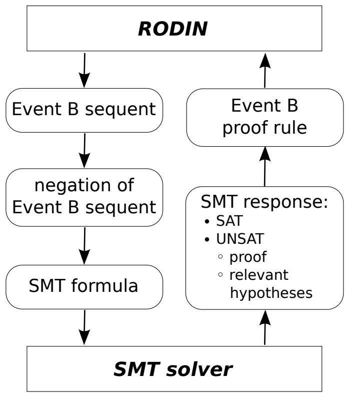

Fig. 1. Schematic view of the interaction between Rodin and SMT solvers.

The SMT-LIB standard proposes several "logics" that specify the interpreted symbols that may be used in the formulas. Currently, however, none of these logics fits exactly the language of the proof obligations generated by Rodin. There exists a proposal for such a logic [14], but the existing SMT solvers do not yet implement corresponding reasoning procedures. Our pragmatic approach is thus to identify subsets of the Event-B logics that may be handled by the current tools, either directly or through some simple transformations. The translation takes as input the Event-B proof obligations. The representation of proof obligations is such that each identifier has been annotated with its type. In the type system, integers and Booleans are predefined, and the user may create new basic sets, or compose existing types with the powerset and Cartesian product constructors. Translating Boolean and arithmetic constructs is mostly straightforward, since a direct syntactic translation may be undertaken for some symbols: Boolean operators and constants, relational operators, and most of arithmetic (division and exponentiation operators are currently translated as uninterpreted symbols). As an example of transformation of an Event-B sequent to an SMT formula, consider the sequent with goal $0 < n + 1$ under the hypothesis $n \in \mathbb{N}$; the type environment is $\{n ⦂ \mathbb{Z}\}$ and the generated SMT-LIB formula is:

```
(set-logic AUFLIA)
(declare-fun n () Int)
(assert (>= n 0))
(assert (not (< 0 (+ n 1))))
(check-sat)
```

The main issue in the translation of proof obligations to SMT-LIB is the representation of the set-theoretic constructs. Different approaches are possible, especially considering the expressiveness of the set operators found in the proof obligation. For instance, the approach presented in [10] encodes sets as predicates, but does not make it possible to reason about sets of sets. We present here an approach that removes this restriction. It uses the *ppTrans* translator [13], already available in the Rodin platform; it removes most set-theoretic constructs from proof obligations by systematically expanding their definitions. It translates an Event-B formula to an equivalent formula in a subset of the Event-B mathematical language (see the grammar of this subset in Fig. 2). The sole set-theoretic symbol is the membership predicate. In addition, the translator performs *decomposition* of binary relations and *purification*, i.e., it separates arithmetic, Boolean and set-theoretic terms. Finally *ppTrans* performs basic Boolean simplifications on formulas. In the following, we provide details on those transformations, using the notation $\varphi \rightsquigarrow \varphi'$ to express that the formula (or sub-term) φ is rewritten to φ'.

$$
\begin{aligned}
P &::= P \Rightarrow P \mid P \equiv P \mid P \wedge \cdots \wedge P \mid P \vee \cdots \vee P \mid \\
&\quad\; \neg P \mid \forall L \cdot P \mid \exists L \cdot P \mid \\
&\quad\; A = A \mid A < A \mid A \leq A \mid M \in S \mid B = B \mid I = I \\
L &::= I \cdots I \\
I &::= \mathit{Name} \\
A &::= A - A \mid A \operatorname{div} A \mid A \bmod A \mid A \exp A \mid \\
&\quad\; A + \cdots + A \mid A \times \cdots \times A \mid -A \mid I \mid \mathit{IntegerLiteral} \\
B &::= \mathsf{true} \mid I \\
M &::= M \mapsto M \mid I \mid \mathsf{integer} \mid \mathsf{bool} \\
S &::= I
\end{aligned}
$$

Fig. 2. Grammar of the language produced by *ppTrans*. The non-terminals are P (predicates), L (list of identifiers), I (identifiers), A (arithmetic expressions), B (Boolean expressions), M (maplet expressions), S (set expressions).

Maplet-hiding variables The rewriting system implemented in *ppTrans* cannot directly transform identifiers that are of type Cartesian product. In a pre-processing phase, such identifiers are thus decomposed, so that further rewriting rules may be applied. This decomposition introduces fresh identifiers of scalar type (members of some given set, integers or Booleans) that name the components of the Cartesian product. Technically, this pre-processing is as follows. We assume the existence of an attribute $\mathcal{T}$, such that $\mathcal{T}(e)$ is the type of expression e. Also, let $\mathsf{fv}(e)$ denote the free identifiers occurring in expression e. The decomposition of the Cartesian product identifiers is specified, assuming an unlimited supply of fresh identifiers (e.g. x_0, x_1,...), using the following two definitions ∇ and ∇^T:

$$
\nabla(i) = \begin{cases} \nabla^T(\mathcal{T}(i)) & \text{if } i \text{ is a product identifier,} \\ i & \text{otherwise.} \end{cases}
$$

$$
\nabla^T(T) = \begin{cases} \nabla^T(T_1) \mapsto \nabla^T(T_2) & \text{if } \exists T_1, T_2 \cdot T = T_1 \times T_2, \\ \text{a fresh identifier } x_i & \text{otherwise.} \end{cases}
$$

For instance, assume $x \mathbin{⦂} \mathbb{Z} \times (\mathbb{Z} \times \mathbb{Z})$; then $\nabla(x) = x_0 \mapsto (x_1 \mapsto x_2)$ and $\mathsf{fv}(\nabla(x)) = \{x_0, x_1, x_2\}$ are fresh identifiers.

The pre-processing behaves as follows:

– Quantified sub-formulas $\forall x \cdot \varphi(x)$, such that x is a product identifier, are rewritten to
$$\forall \mathsf{fv}(\nabla(x)) \cdot \varphi[\nabla(x)/x],$$
where $e[e'/x]$ denotes expression e where expression e' has been substituted for all free occurrences of x.
Ex. $\forall a {\cdot} a = 1 \mapsto (2 \mapsto 3) \rightsquigarrow \forall a_0, a_1, a_2 \cdot a_0 \mapsto (a_1 \mapsto a_2) = 1 \mapsto (2 \mapsto 3)$.
– Let ψ denote the top-level formula and let $x_1 \dots x_n$ be the free Cartesian product identifiers of ψ. Then:
$$\begin{aligned}\psi \rightsquigarrow\ & \forall \mathsf{fv}(\nabla(x_1)) \cdots \mathsf{fv}(\nabla(x_n)) \cdot \\ & \quad (x_1 = \nabla(x_1) \wedge \cdots x_n = \nabla(x_n)) \Rightarrow \psi[\nabla(x_1)/x_1] \cdots [\nabla(x_n)/x_n].\end{aligned}$$
Ex. $\psi \equiv a = b \wedge a \in S$ with typing $\{a \mathbin{⦂} S, b \mathbin{⦂} S, S \mathbin{⦂} \mathbb{P}(\mathbb{Z} \times \mathbb{Z})\}$:
$$\begin{aligned}\psi \rightsquigarrow\ & \forall x_0, x_1, x_2, x_3 \cdot \\ & \quad (a = x_0 \mapsto x_1 \ \wedge\ b = x_2 \mapsto x_3) \Rightarrow \\ & \quad\ (x_0 \mapsto x_1 = x_2 \mapsto x_3 \wedge x_0 \mapsto x_1 \in S)\end{aligned}$$

Purification The goal of this phase is to obtain pure terms, i.e. terms that do not mix symbols of separate syntactic categories: arithmetic, predicate, set, Boolean, and maplet symbols. This is done by introducing new variables. In Event-B, heterogeneous terms result from the application of symbols with a signature with different sorts (e.g. symbol $\subseteq$ yields a predicate from two sets). This phase also eliminates some syntactic sugar. Figure 3 depicts the different syntactic categories, how the Event-B operators relate them, and the effect of desugarization. There is an arrow from category X to category Y if a term from X may have an argument in Y. For instance $. \in$ labels the arrow from P to A since the left argument of $\in$ may be an arithmetic term, e.g. in $x + y \in S$.

First, let us introduce informally the notation $Q^? P[e^*]$, where Q is $\forall$ or $\exists$, P a predicate, and e an expression in P such that the syntactic category of e is not the same as that of its parent (identifiers are considered to belong to all syntactic categories). This denotes the possible introduction of the quantifier Q on a fresh variable, so that heterogeneous sub-terms in e are purified, yielding e^*, as illustrated by the following examples:

1. $\exists^? (a \mapsto (1 \mapsto 2))^* \in S$ represents $\exists x_0, x_1 {\cdot} x_0 = 1 \wedge x_1 = 2 \wedge a \mapsto (x_0 \mapsto x_1) \in S$ as 1 and 2 are not in the same syntactic category as the maplet.
2. $\forall^? (a \mapsto b)^* \in S$ does not introduce a quantification and denotes $a \mapsto b \in S$.

Due to lack of space, we select some rules of the rewrite system implemented in *ppTrans*. The symbols relating the syntactic categories P (predicates) and S (sets) are reduced to membership ($\in$) and equality ($=$) by application of the rules such as:

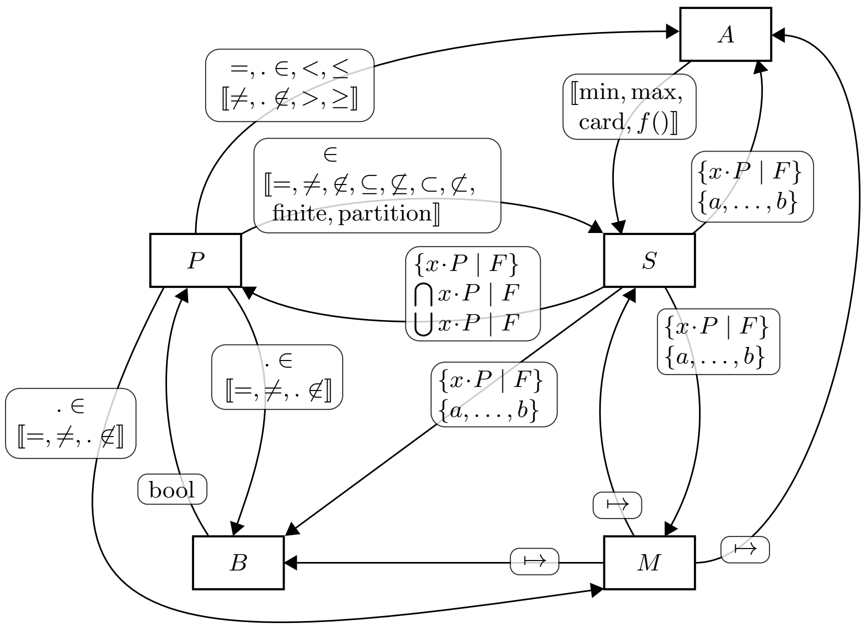

Fig. 3. The different syntactic categories and the symbols relating them: A for arithmetic expressions, P for predicates, S for set expressions, B for Boolean expressions and M for maplet expressions. *ppTrans* removes all occurrences of the constructs delimited by double-brackets.

$$x \neq y \rightsquigarrow \neg(x = y) \quad (2)$$

$$s \subseteq t \rightsquigarrow s \in \mathbb{P}(t) \quad (3)$$

$$x \notin s \rightsquigarrow \neg(x \in s) \quad (4)$$

$$\text{finite}(s) \rightsquigarrow \forall a \cdot \exists b, f \cdot f \in s \rightarrowtail a..b \quad (5)$$

Moreover rules 2 and 4 are also applied when the arguments belong to other syntactic categories and are responsible for the elimination of all the occurrences of symbols $\neq$ and $\notin$. Examples to eliminate equalities between syntactic categories S, M are:

$$x_1 \mapsto x_2 = y_1 \mapsto y_2 \rightsquigarrow x_1 = y_1 \land x_2 = y_2 \quad (6)$$

$$\text{bool}(P) = \text{bool}(Q) \rightsquigarrow P \Leftrightarrow Q \quad (7)$$

$$\text{bool}(P) = \text{TRUE} \rightsquigarrow P \quad (8)$$

$$x = f(y) \rightsquigarrow y \mapsto x \in f \quad (9)$$

$$x = \text{FALSE} \rightsquigarrow \neg(x = \text{TRUE}) \quad (10)$$

Due to the symmetry property of equality, *ppTrans* also applies a symmetric version of each such rule. The symbols that embed arithmetic terms are taken care of with rules such as:

$$n = \text{card}(s) \rightsquigarrow \exists f \cdot f \in s \rightarrowtail\!\!\!\!\rightarrow 1..n \quad (11)$$

$$n = \max(s) \rightsquigarrow n \in s \land \max(s) \leq n \quad (12)$$

$$a \succ b \rightsquigarrow b \prec a \quad (13)$$

$$a \prec \max(s) \rightsquigarrow \exists x \cdot x \in s \land a \prec x \quad (14)$$

The remaining rules perform the following roles: rewrite applications of the set membership symbol according to the rightmost argument (e.g. 15), eliminate some symbols (e.g. 17), and handle miscellaneous other cases:

$$e \in s \twoheadleftarrow\!\!\!\!\twoheadrightarrow t \rightsquigarrow e \in s \twoheadleftarrow\!\!\!\!\rightarrow t \land t \subseteq \text{ran}(e) \quad (15)$$

$$e \in s \twoheadleftarrow\!\!\!\!\rightarrow t \rightsquigarrow e \in s \leftrightarrow t \land s \subseteq \text{dom}(e) \quad (16)$$

$$e \mapsto f \in s \times t \rightsquigarrow e \in s \land f \in t \quad (17)$$

$$e \mapsto f \in \text{id} \rightsquigarrow e = f \quad (18)$$

$$e \mapsto f \in r^{-1} \rightsquigarrow f \mapsto e \in r \quad (19)$$

$$e \mapsto f \in s \lhd r \rightsquigarrow e \mapsto f \in r \land e \in s \quad (20)$$

$$e \mapsto f \in \text{pred} \rightsquigarrow e = f + 1 \quad (21)$$

The full system consists of 80 rules. They are either sound purification rules, or the equivalence of the left and right side terms can easily be derived from the definitions (see [1]) of the eliminated symbols. Purification rules eliminate heterogeneous terms and are only applied once. It is not difficult to order all other rules such that no eliminated symbol is introduced in subsequent rules. The rewriting system is thus indeed terminating.

Output to SMT-LIB format Once *ppTrans* has completed rewriting, the resulting proof obligation is ready to be output in SMT-LIB format. The translation from *ppTrans*' output to SMT-LIB follows specific rules for the translation of the set membership operator. For instance assume the input has the following typing environment and formulas:

Typing environment	Formulas
$a \mathrel{⦂} S$	
$b \mathrel{⦂} T$	
$c \mathrel{⦂} U$	$a \in A$
$A \mathrel{⦂} \mathbb{P}(S)$	$a \mapsto b \in r$
$r \mathrel{⦂} \mathbb{P}(S \times T)$	$a \mapsto b \mapsto c \in s$
$s \mathrel{⦂} \mathbb{P}(S \times T \times U)$	

Firstly, for each basic set found in the proof obligation, the translation produces a sort declaration in SMT-LIB. However, as there is currently no logic in the SMT-LIB with powerset and Cartesian product sort constructors, *ppTrans* handles them by producing an additional sort declaration for each combination of basic sets (either through powerset or Cartesian product). Translating the typing environment thus produces a sort declaration for each basic set, and combination thereof found in the input. In SMT-LIB, sorts have a name and an arity, which is non-null for polymorphic sorts. The sorts produced have all arity 0, and for the above example, the following is produced:

$$\begin{aligned} S &\leadsto \texttt{(declare-sort S 0)} \\ T &\leadsto \texttt{(declare-sort T 0)} \\ U &\leadsto \texttt{(declare-sort U 0)} \\ \mathbb{P}(S) &\leadsto \texttt{(declare-sort PS 0)} \\ \mathbb{P}(S \times T) &\leadsto \texttt{(declare-sort PST 0)} \\ \mathbb{P}(S \times T \times U) &\leadsto \texttt{(declare-sort PSTU 0)} \end{aligned}$$

Secondly, for each constant, the translation produces a function declaration of the appropriate sort:

$$\begin{aligned} a \mathrel{⦂} S &\leadsto \texttt{(declare-fun a () S)} \\ b \mathrel{⦂} T &\leadsto \texttt{(declare-fun b () T)} \\ c \mathrel{⦂} U &\leadsto \texttt{(declare-fun c () U)} \\ A \mathrel{⦂} \mathbb{P}(S) &\leadsto \texttt{(declare-fun A () PS)} \\ r \mathrel{⦂} \mathbb{P}(S \times T) &\leadsto \texttt{(declare-fun r () PST)} \\ s \mathrel{⦂} \mathbb{P}(S \times T \times U) &\leadsto \texttt{(declare-fun s () PSTU)} \end{aligned}$$

Third, for each type occurring at the right-hand side of a membership predicate, the translation produces fresh SMT function symbols:

```
(declare-fun (MS0 (S PS) Bool))
(declare-fun (MS1 (S T PST) Bool))
(declare-fun (MS2 (S T U PSTU)) Bool)
```

The Event-B atoms can then be translated as follows:

$$\begin{aligned} a \in A &\leadsto \texttt{(MS0 a A)} \\ a \mapsto b \in r &\leadsto \texttt{(MS1 a b r)} \\ a \mapsto b \mapsto c \in s &\leadsto \texttt{(MS2 a b c s)} \end{aligned}$$

For instance, $A \cup \{a\} = A$ would be translated to $\forall x \cdot (x \in A \lor x = a) \Leftrightarrow x \in A$, that is, in SMT-LIB format:

```
(forall ((x S)) (= (or (MS0 x A) (= x a)) (MS0 x A)))
```

While the approach presented here covers the whole Event-B mathematical language and is compatible with the SMT-LIB language, the semantics of some Event-B constructs is approximated because some operators become uninterpreted in SMT-LIB (chiefly membership but also some arithmetic operators such as division and exponentiation). However, we can recover their interpretation by adding axioms to the SMT-LIB benchmark, at the risk of decreasing the performance of the SMT solvers. Some experimentation is thus needed to find a good balance between efficiency and completeness.

Indeed, it appears experimentally that including some axioms of set theory to constrain the possible interpretations of the membership predicate greatly improves the number of proof obligations discharged. In particular, the axiom of elementary set (singleton part) is necessary for many Rodin proof obligations. The translator directly instantiates the axiom for all membership predicates. Assuming `MS` is the membership predicate associated with sorts `S` and `PS`, the translation introduces thus the following assertion:

```
(assert (forall ((x S))
          (exists ((X PS)) (and (MS x X)
                              (forall ((y S)) (=> (MS y X) (= y x)))))))
```

This particular assertion eliminates non-standard interpretations where some singleton sets do not exist. Without it, some formulas are satisfiable because of spurious models and the SMT solvers are unable to refute them.

A small example As a concrete example of translation, we consider the sequent presented in Section 3. Figure 4 presents the SMT-LIB input resulting from the translation approach described in this paper. Since the proof obligation includes sets of *JOBS*, a corresponding sort *PJ* and membership predicate *MS* are declared in lines 3–4. Then, the function symbols corresponding to the free identifiers of the sequent are declared at lines 5–7. Finally, the hypotheses and the goal of the sequent are translated to named assertions (lines 8–14).

The sequent described in this section is very simple and is easily verified by both Atelier-B provers and SMT solvers. It is noteworthy that the plug-in inspects sequents to choose the most suitable encoding approach. The next section reports experiments with a large number of proof obligations and establishes a better basis to compare the effectiveness of these different verification techniques.

5 Experimental results

We evaluated experimentally the effectiveness of using SMT solvers as reasoners in the Rodin platform by means of the techniques presented in this paper. This evaluation complements the experiments presented in [11] and reinforces their conclusions. We established a library of 2,456 proof obligations stemming from Event-B developments collected by the European FP7 project Deploy and publicly available on the Deploy repository[5]. These developments originate from

[5] `http://deploy-eprints.ecs.soton.ac.uk`

```
(set-logic AUFLIA)
(declare-sort JOBS 0)
(declare-sort PJ 0)
(declare-fun MS (JOBS PJ) Bool)
(declare-fun active () PJ)
(declare-fun j () JOBS)
(declare-fun queue () PJ)
(assert (! (forall ((x JOBS))
              (not (and (MS x active) (MS x queue)))) :named inv3))
(assert (! (MS j queue) :named grd1))
(assert (! (not (forall ((x0 JOBS))
                   (not (and (or (MS x0 active) (= x0 j))
                             (MS x0 queue)
                             (not (= x0 j)))))) :named goal))
(check-sat)
```

Fig. 4. SMT-LIB input produced using the *ppTrans* approach.

examples from Abrial's book [2], academic publications, tutorials, as well as industrial case studies.

One main objective of introducing new reasoners in the Rodin platform is to reduce the number of valid proof obligations that need to be discharged interactively by humans. Consequently, the effectiveness of a reasoner is measured by the number of proof obligations proved automatically by the reasoner.

Obviously, effectiveness should depend on the computing resources given to the reasoners. In practice, the amount of memory is seldom a bottleneck, and usually the solvers are limited by setting a timeout on their execution time. In the context of the Rodin platform, the reasoners are executed by synchronous calls, and the longer the time limit, the less responsive is the framework to the user. We have experimented different timeouts and our experiments have shown us that a timeout of one second seems a good trade-off: doubling the timeout to two seconds increases by fewer than 0.1% the number of verified proof obligations, while decreasing the responsiveness of the platform.

Table 1 compares different reasoners on our set of benchmarks. The second column corresponds to Rodin internal normalization and simplification procedures. It shows that more than half of the generated proof obligations necessitate advanced theorem-proving capabilities to be discharged. The third column is a special-purpose reasoner, namely Atelier-B provers. They were originally developed for the B method and are also available in the Rodin platform. Although they are extremely effective, the Atelier-B provers now suffer from legacy issues. The last five columns are various SMT solvers applied to the proof obligations generated by the plug-in. The SMT solvers were used with a timeout of one second, on a computer equipped with an Intel Core i7-4770, cadenced at 3.40 GHz, with 24 GB of RAM, and running Ubuntu Linux 12.04. They show decent results, but they are not yet as effective reasoners as the Atelier-B theorem provers.

Number of proof obligations	Rodin	Atelier-B	alt-ergo-r217	cvc3-2011-11-21	veriT-dev-r2863	veriT & E-prover	z3-3.2
2456	1169	2260	2017	2218	2051	2160	2094

Table 1. Number of proof obligations discharged by the reasoners.

Although this comparison is interesting to evaluate and compare the different reasoners, it is not sufficient to evaluate the effectiveness of the approach presented in this paper. Indeed, nothing prevents users to use several other reasoners once one has failed to achieve its goal. In Table 2, we report how many proof obligations remain unproved after applying the traditional reasoners (Atelier-B theorem provers and the Rodin reasoner) in combination with each SMT solver, and with all SMT solvers.

Number of proof obligations left	alt-ergo-r217	cvc3-2011-11-21	veriT-dev-r2863	veriT & E-prover	z3-3.2	All SMT solvers
196	114	61	94	103	92	31

Table 2. Number of proof obligations *not* discharged by special-purpose reasoners and by each SMT solver.

Each SMT solver seems a useful complement to the special-purpose provers. However, we would also like to know whether the reasoning capacity of some of these solvers is somehow subsumed by another solver, or whether each SMT solver could provide a significant contribution towards reducing the number of proof obligations that need to be discharged by humans. Table 3 synthesizes a pairwise comparison of the SMT solvers on our universe of proof obligations.

This comparison signals the results obtained when all available reasoners are applied: only 31 proof obligations are unproved, down from 196 resulting from the application of Atelier-B provers. It is also noteworthy that even though each SMT solver is individually less effective than Atelier-B provers, applied altogether, they prove all but 97 proof obligations. The important conclusion of our experiments is that there is strong evidence that SMT solvers complement in an effective and practical way the Atelier B provers, yielding significant improve-

	alt-ergo	cvc3	veriT	veriT+E	z3
alt-ergo	2017	2001	1880	1967	1911
cvc3	2001	2218	1953	2088	2031
veriT	1880	1953	2051	1958	1878
veriT+E	1967	2088	1958	2160	2067
z3	1911	2031	1878	1972	2094

Table 3. Number of proof obligations verified by SMT solver A also discharged by solver B.

ments in the usability of the Rodin platform and its effectiveness to support the development of Event-B models.

6 Conclusion

SMT solving is a formal verification technique successfully applied to various domains including verification. SMT solvers do not have built-in support for set-theoretic constructs found in Rodin sequents. We presented here a translation approach to tackle this issue. We evaluated experimentally the efficiency of SMT solvers against proof obligations resulting from the translation of Rodin sequents. In our sample of industrial and academic projects, the use of SMT solvers on top of Atelier B provers reduces significantly the number of unverified sequents. This plug-in is available through the integrated software updater of Rodin (instructions at `http://wiki.event-b.org/index.php/SMT_Plug-in`).

The results are very encouraging and motivate us to progress further by implementing and evaluating new translation approaches, such as representing functions using arrays in the line of [8]. Elaborating strategies to apply different reasoners, based on some characteristics of the sequents is also a promising line of work. Another feature of some SMT solvers is that they can provide models when a formula is satisfiable. In consequence, it would be possible, with additional engineering effort, to use such models to report counter-examples in Rodin.

We believe that the approach presented in this paper could also be applied successfully for other set-based formalisms such as: the B method, TLA+, VDM and Z.

Cooperation of deduction tools is very error-prone, not only because it relies on the correctness of many large and complex tools, but also because of the translations. Certification of proofs in a centralized trusted proof manager would be the answer to this problem. Preliminary works in this direction exist [15].

Acknowledgements: This paper is an abbreviated version of [12].

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

2. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.
3. M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In *First Int'l Conference on Certified Programs and Proofs, CPP 2011*, volume 7086 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 2011.
4. C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, Feb. 2009.
5. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard Version 2.0, 2010.
6. T. Bouton, D. C. B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: An Open, Trustable and Efficient SMT-Solver. In *Proc. Conference on Automated Deduction (CADE)*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.
7. J. Coleman, C. Jones, I. Oliver, A. Romanovsky, and E. Troubitsyna. RODIN (Rigorous open Development Environment for Complex Systems). In *Fifth European Dependable Computing Conference: EDCC-5 supplementary volume*, pages 23–26, 2005.
8. J.-F. Couchot, D. Déharbe, A. Giorgetti, and S. Ranise. Scalable Automated Proving and Debugging of Set-Based Specifications. *Journal of the Brazilian Computer Society*, 9:17–36, 2003.
9. L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
10. D. Déharbe. Automatic Verification for a Class of Proof Obligations with SMT-Solvers. In M. Frappier, G. Uwe, K. Sarfraz, R. Laleau, and S. Reeves, editors, *Proceedings 2nd Int'l Conf. Abstract State Machines, Alloy, B and Z, ABZ 2010*, volume 5977 of *Lecture Notes in Computer Science*, pages 217–230. Springer, 2010.
11. D. Déharbe, P. Fontaine, Y. Guyot, and L. Voisin. SMT solvers for Rodin. In J. Derrick, J. A. Fitzgerald, S. Gnesi, S. Khurshid, M. Leuschel, S. Reeves, and E. Riccobene, editors, *Proc 3rd Int. Conf. Abstract State Machines, Alloy, B, VDM, and Z (ABZ 2012)*, volume 7316 of *Lecture Notes in Computer Science*, pages 194–207. Springer, 2012.
12. D. Déharbe, P. Fontaine, Y. Guyot, and L. Voisin. Integrating SMT solvers in Rodin. *Science of Computer Programming*, 2014. Available online.
13. M. Konrad and L. Voisin. Translation from Set-Theory to Predicate Calculus. Technical report, ETH Zurich, 2011.
14. D. Kröning, P. Rümmer, and G. Weissenbacher. A Proposal for a Theory of Finite Sets, Lists, and Maps for the SMT-LIB Standard. In *Informal proceedings, 7th Int'l Workshop on Satisfiability Modulo Theories (SMT) at CADE 22*, 2009.
15. M. Schmalz. The logic of Event-B, 2011. Technical report 698, ETH Zürich, Information Security.
16. The Eclipse Foundation. Eclipse SDK, 2009.