SMT solvers for Rodin *

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

[1] Universidade Federal do Rio Grande do Norte, Natal, RN, Brazil
david@dimap.ufrn.br
[2] University of Nancy and INRIA, Nancy, France
Pascal.Fontaine@inria.fr
[3] Systerel, France
{yoann.guyot,laurent.voisin}@systerel.fr

Abstract. Formal development in Event-B generally requires the validation of a large number of proof obligations. Some automatic tools exist to automatically discharge a significant part of them, thus augmenting the efficiency of the formal development. We here investigate the use of SMT (Satisfiability Modulo Theories) solvers in addition to the traditional tools, and detail the techniques used for the cooperation between the Rodin platform and SMT solvers.
Our contribution is the definition of two approaches to use SMT solvers, their implementation in a Rodin plug-in, and an experimental evaluation on a large sample of industrial and academic projects. Adding SMT solvers to Atelier B provers reduces to one fourth the number of sequents that need to be proved interactively.

1 Introduction

The Rodin platform [7] is an integrated design environment for the formal modeling notation Event-B [1]. Rodin is based on the Eclipse framework [18] and has an extensible architecture, where new features, or new versions of existing features, can be integrated by means of plug-ins. It supports the construction of formal models of systems as well as their refinement using the notation of Event-B, based on first-order logic, typed set theory and integer arithmetic. Event-B models should be consistent; for this purpose, Rodin generates proof obligations that need to be discharged (i.e., proved valid).

The proof obligations are represented internally as sequents, and a sequent calculus forms the basis of the verification machinery. Proof rules are applied to a sequent and produce zero, one or more new, usually simpler, sequents. A proof rule producing no sequent is called a discharging rule. The goal of the verification is to build a proof tree corresponding to the application of the proof rules, where all the leaves are discharging rules. In practice, the proof rules are generated by

* This work is partly supported by ANR project DECERT, CNPq/INRIA project SMT-SAVeS, and CNPq grants 560014/2010-4 and 573964/2008-4 (National Institute of Science and Technology for Software Engineering—INES, www.ines.org.br).

so-called *reasoners*. A reasoner is a plug-in that can either be standalone or use existing verification technologies through third-party tools.

The usability of the Rodin platform, and of formal methods in general, greatly depends on several aspects of the verification activity:

Automation Ideally, the proof obligations are validated automatically by reasoners. If human interaction is required for discharging proof obligations (using an interactive theorem prover), productivity is negatively impacted.

Information Validation of proof obligations should not be sensitive to irrelevant modifications of the model. When modifying the model, large parts of the proof can be preserved if the precise facts used to validate each proof obligation are recorded. Moreover, similar proof obligations can be discharged without further need of the reasoner by noticing the same proof applies, even if the proof obligations differ slightly (on irrelevant parts).

Finally, counter-examples of failed proof obligations can be very valuable to the user as hints to improve the model and the invariants.

Trust 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.

In this paper, we address the application of a verification approach that may potentially fulfill these three requirements: *Satisfiability Modulo Theory* (SMT) solvers. 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. In this paper, we propose a translation of Event-B sequents to SMT input, the difficulty lying essentially in the way sets are translated.

The SMT-LIB initiative provides a standard for the input language of SMT solvers, and, in its last version [4], a command language defining a common interface to interact with SMT solvers. We implemented a Rodin plug-in using this interface. The plug-in also extracts from the SMT solvers some additional *information* such as the relevant hypothesis. Some solvers (e.g. Z3 [9] and veriT [6]) are able to generate a comprehensive proof for validated formulas, which can be verified by a *trusted* proof checker [2]. In the longer term, besides automation, and information, trust may be obtained using a centralized proof manager.

Overview. Section 2 presents two approaches to translate Rodin sequents to the SMT-LIB notation. Section 3 illustrates both approaches through a simple example. Section 4 gives some insights on the techniques employed in SMT solvers to handle Rodin sequents and section 5 presents experimental results, based on the verification activities carried out for a variety of Event-B projects. We conclude by discussing future work.

Throughout the paper, formulas are expressed using the Event- B syntax [14], and sentences in SMT-LIB are typeset using a `typewriter` font.

2 Translating Event-B to SMT

Figure 1 gives a schematic view of the cooperation framework between Rodin and the SMT solver. For each Event-B sequent representing a proof obligation to be validated in Rodin, an SMT formula is built. SMT solvers answer the satisfiability question, so that 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. On success a proof and an unsatisfiable core — i.e., the set of facts necessary to prove that the formula is unsatisfiable — may be supplied to Rodin, which will extract a new Event-B proof rule out of it. 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).

The SMT-LIB standard proposes several "logics" that specify the interpreted symbols that may be used in the formulas. Currently, however, none of those logics fits exactly the language of the proof obligations generated by Rodin. There exists a proposal for such a logic [13], 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. 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 $\{n \oftype \mathbb{Z}\}$ and the generated SMT-LIB formula is:

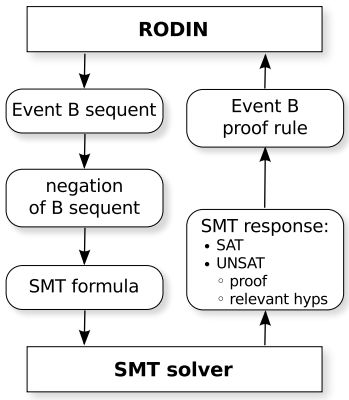

Fig. 1. Schematic view of the interaction between Rodin and SMT solvers.

```
(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. We present successively two approaches. The simplest one, presented shortly in the next section, is based on the representation of sets as characteristic predicates [10]. Since SMT solvers handle first-order logic, this approach does not make it possible to reason about sets of sets. The second approach removes this restriction. It uses the *ppTrans*

translator, already available in the Rodin platform; this translator removes most set-theoretic constructs from proof obligations by systematically expanding their definitions.

2.1 The λ-based approach

This approach implements and extends the principles proposed in [10] to handle simple sets. Essentially, a set is identified with its characteristic function. For instance the singleton $\{1\}$ is identified with $(\lambda x \oftype \mathbb{Z} \mid x = 1)$ and the empty set is identified with the polymorphic λ-expression $(\lambda x \oftype X \mid \text{FALSE})$, where X is a type variable. The union of (two) sets is a polymorphic higher-order function $(\lambda(S_1 \oftype X \to \text{BOOL}) \mapsto (S_2 \oftype X \to \text{BOOL}) \mid (\lambda x \oftype X \mid S_1(x) \lor S_2(x)))$, etc.

SMT-LIB does not provide a facility for λ-expressions, and has limited support for polymorphism. This approach requires several extensions to SMT-LIB: λ-expressions, a polymorphic sort system, and macro-definitions. Those extensions are actually implemented in the veriT parser. Consider the sequent $A \oftype \mathbb{P}(\mathbb{Z}) \vdash A \cup \varnothing = A$, the translator to this extended SMT-LIB language produces:

```
(declare-fun A (Int) Bool)
(define-fun (par (X) (union ((S1 (X Bool)) (S2 (X Bool))) (X Bool)
            (lambda ((x X)) (or (S1 x) (S2 x))))))
(define-fun (par (X) (emptyset () (X Bool) (lambda ((x X)) false))))
(assert (not (= (union A emptyset) A)))
(check-sat)
```

where `X` denotes a sort variable. The function definitions `union` and `emptyset` are inserted by the translator and are part of a corpus of definitions for most of the set-theoretic constructs (see [10, 11] for details). They are divided into a list of sorted parameters, the sort of the result, and the body expressing the value of the result. The macro processor implemented in veriT transforms the goal to

```
(not (forall ((x Int)) (iff (or (A x) false) (A x))))
```

i.e., a first-order formula that may then be handled using usual SMT solving techniques. It is also possible to use veriT only as a pre-processor to produce plain SMT-LIB formulas that are amenable to verification using any SMT-LIB compliant solver.

As already mentioned, the main drawback of this approach is that sets of sets cannot be handled. It is thus restricted to simple sets and relations. Furthermore its reliance on extensions of the SMT-LIB format creates a dependence on veriT as a macro processor. The next approach lifts these restrictions.

2.2 The *ppTrans* approach

Our second approach uses the translator *ppTrans* provided by the *Predicate Prover* available in Rodin in order to obtain first-order logic formulas which are almost free of set-theoretic elements [12]. It also separates arithmetic, Boolean and set-theoretic constructs from each other and performs simplifications. This

approach makes the plug-in independent from veriT, and is more robust with respect to the translation of relations and functions. On formulas suitable for the previous approach, the translator would however produce very similar results compared to this previous simple approach.

Besides the straightforward translations mentioned earlier, the translation from the *ppTrans* output to SMT-LIB provides some specific rules for the translation of set-theoretic constructs such as the membership operator. For instance assume the input has the following typing environment and formulas:

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

First, for each basic set found in the proof obligation, the translation produces a sort declaration in SMT-LIB. In addition, for each combination of basic sets (either through powerset or Cartesian product), an additional sort declaration is produced. Translating the typing environment produces a sort declaration for each basic set, and combination thereof found in the input:

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

Second, the translation produces a function declaration for each constant:

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

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:

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

Finally, the Event-B formula where all non-membership set operators have been expanded to their definition is translated to SMT-LIB. For instance, the formula $A \cup \varnothing = A$ would be translated to $\forall x \cdot (x \in A \vee x \in \varnothing) \Leftrightarrow x \in A$, which *ppTrans* simplifies to $\forall x \cdot (x \in A \vee \bot) \Leftrightarrow x \in A$, would be translated to

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

While the approach presented here covers the whole Event-B mathematical language and does not require polymorphic types or specific extensions to 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)))))))
```

More implementation and optimization details are available in [12]. It is noteworthy that the plug-in based on *ppTrans* detects sequents with only simple sets (i.e., no sets of sets) and uses a translation similar to the λ-based approach in that case. Therefore, the *ppTrans* approach subsumes the λ-based approach.

3 A small Event-B example

As a concrete example of translation, this section presents the model of a simple job processing system consisting of a queue and a processor. The basic sets are *JOBS* (the jobs) and *STATUS* (the possible states of the processor), such that `axm1` : $STATUS = \{RUN, IDLE\}$, and `axm2` : $RUN \neq IDLE$. The state of the model has three variables: `proc` (the current status of the processor) `queue` (the jobs currently queued) and `active` (the job being processed, if any). This state is constrained by the following invariants:

`inv1` : $proc \in STATUS$ (typing)
`inv2` : $active \in JOBS$ (typing)
`inv3` : $queue \in \mathbb{P}(JOBS)$ (typing)
`inv4` : $proc = RUN \Rightarrow active \notin queue$

One of the events of the system describes that the processor takes on a new job. It is specified as follows:

Event $SCHEDULE \mathrel{\widehat{=}}$ (the processor takes on a new job)
  any
    j
  where
    `grd1` : $proc = IDLE$ (the processor must be idle)
    `grd2` : $j \in queue$ (the job j is in the queue)
  then
    `act1` : $queue := queue \setminus \{j\}$
    `act2` : $active := j$
    `act3` : $proc := RUN$
  end

To verify that the invariant labeled `inv4` is preserved by the *SCHEDULE* event, the following sequent must be proved valid:

$$\texttt{axm1}, \texttt{axm2}, \texttt{inv1}, \texttt{inv2}, \texttt{inv3}, \texttt{inv4}, \texttt{grd1}, \texttt{grd2} \vdash \underbrace{RUN}_{proc} = RUN \Rightarrow \underbrace{j}_{active} \notin \underbrace{queue \setminus \{j\}}_{queue}. \quad (1)$$

The generated proof obligations thus aim to show that the following formula is unsatisfiable:

$$\begin{array}{l} STATUS = \{RUN, IDLE\} \land RUN \neq IDLE \land \\ proc \in STATUS \land active \in JOBS \land queue \in \mathbb{P}(JOBS) \land \\ proc = RUN \Rightarrow active \notin queue \land \\ proc = IDLE \land j \in queue \land \\ \neg(RUN = RUN \Rightarrow j \notin queue \setminus \{j\}). \end{array}$$

This proof obligation does not contain sets of sets and the approach described in section 2.1 may be applied resulting in the SMT-LIB input presented in Figure 2. Lines 2 and 3 contain the declarations of the sorts corresponding to the basic sets introduced in the context. Lines 4–9 contain the declarations of the function symbols corresponding to the free variables of the proof obligation, and are produced using the typing environment. Note that set *queue* is represented by a unary predicate symbol. Next, the definitions of the macros corresponding to set operators $\in$ and $\setminus$ are included on lines 10–13. Line 14 is the definition of a macro that represents the singleton set $\{j\}$. Lines 15–21 are the result of the translation of the proof obligation itself.

Of course, this proof obligation is also amenable to translation using the approach described in section 2.2, and the corresponding SMT-LIB input is given in Figure 3. Since the proof obligation includes sets of *JOBS*, a corresponding sort *PJOBS* and membership predicate *MJOBS* are declared in lines 4–5. Then, the function symbols corresponding to free identifiers of the sequent are declared at lines 6–11. Finally, the hypothesis and the goal of the sequent are translated to named assertions (lines 12–18).

```
 1 (set-logic AUFLIA)
 2 (declare-sort STATUS 0)
 3 (declare-sort JOBS 0)
 4 (declare-fun RUN () STATUS)
 5 (declare-fun IDLE () STATUS)
 6 (declare-fun proc () STATUS)
 7 (declare-fun active () JOBS)
 8 (declare-fun j () JOBS)
 9 (declare-fun queue (JOBS) Bool)
10 (define-fun (par (X) (in ((x X) (s (X Bool))) Bool (s x))))
11 (define-fun (par (X)
12              (setminus ((s1 (X Bool)) (s2 (X Bool))) (X Bool)
13                (lambda ((x X)) (and (s1 x) (not (s2 x)))))))
14 (define-fun set1 ((x JOBS)) Bool (= x j))
15 (assert (and (forall ((x STATUS)) (or (= x RUN) (= x IDLE)))
16           (not (= RUN IDLE))
17           (=> (= proc RUN) (not (in active queue)))
18           (= proc IDLE)
19           (in j queue)
20           (not (=> (= RUN RUN)
21                    (not (in j (setminus queue set1)))))))
22 (check-sat)
```

Fig. 2. SMT-LIB input produced using the λ-based approach.

The sequent described in this section is very simple and is easily verified by both Atelier-B provers and SMT-solvers. Section 5 reports experiments with a large number of proof obligations and establishes a better basis to compare the effectiveness of these different verification techniques.

4 Solving SMT formulas

In this section, we provide some insight about the internals of SMT solvers, in order to give to the reader an idea on the kind of formulas that can successfully be handled by SMT solvers. A very schematic view of an SMT solver is presented on Figure 4. Basically it is 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.

4.1 Unquanti ed formulas

Historically, the first goal of SMT solvers was to provide efficient decision procedures for expressive languages, beyond pure propositional logic. Those solvers

```
(set-logic AUFLIA)
(declare-sort STATUS 0)
(declare-sort JOBS 0)
(declare-sort PJOBS 0)
(declare-fun MJOBS (JOBS PJOBS) Bool)
(declare-fun RUN () STATUS)
(declare-fun IDLE () STATUS)
(declare-fun proc () STATUS)
(declare-fun active () JOBS)
(declare-fun queue () PJOBS)
(declare-fun j () JOBS)
(assert (! (forall ((x STATUS)) (or (= x RUN) (= x IDLE))) :named axm1))
(assert (! (not (= RUN IDLE)) :named axm2))
(assert (! (= proc IDLE) :named grd1))
(assert (! (MJOBS j queue) :named grd2))
(assert (! (not (=> (= RUN RUN)
                    (not (and (MJOBS j queue)
                              (not (= j j)))))) :named goal))
(check-sat)
```

Fig. 3. SMT-LIB input produced using the *ppTrans* approach.

have always been based on a cooperation of a Boolean engine, nowadays typically a SAT solver (See [5] for more information on SAT solver techniques and tools), and a theory reasoner to check the satisfiability of a set of literals in the considered language. The Boolean engine generates models for the Boolean abstraction of the input formula, whereas the theory reasoner refutes the sets of literals corresponding to these abstract models by adding conjunctively conflict clauses to the propositional abstraction. This exchange runs until either the Boolean abstraction is sufficiently refined for the Boolean reasoner to conclude that the formula is unsatisfiable, or the theory reasoner concludes that the abstract model indeed corresponds to a model of the formula.

The theory reasoners are themselves based on a combination of decision procedures for various fragments. In our context, the relevant decision procedures are congruence closure — to handle uninterpreted predicates and functions — decision procedure for arrays (typically reduced to some kind of congruence closure), and linear arithmetic. It is possible, using the Nelson-Oppen combination method [15, 19], to build a decision procedure for the union of the languages. The theory reasoner used in most SMT solvers is thus able to decide the satisfiability of literals on a language containing a mix of uninterpreted symbols, linear arithmetic symbols, and array operators.

For the theory reasoner and the SAT solver to cooperate successfully, some techniques are necessary. Among these techniques, if a set of literals is found unsatisfiable, it is most valuable to generate small conflict clauses, in order to refine the Boolean abstraction as strongly as possible. Also, theory propagation,

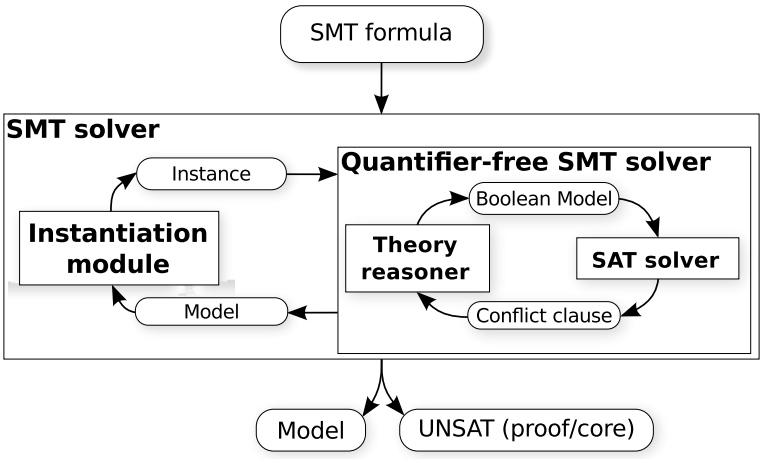

Fig. 4. Schematic view of an SMT solver.

that allows to control the decisions taken inside the SAT solver, has proved to be very worthwhile in practice (more can be found about these techniques and SMT solving in general in [3]).

4.2 Instantiation techniques

Automatically finding the right instances of quantified formulas is a key issue for the verification of sequents (as well as proof obligations produced in the context of a number of software verification tools). The quantifier instantiation module is responsible for producing lemmas of the form $\neg\varphi(\mathsf{t}) \vee \exists \mathsf{x}\, \varphi(\mathsf{x})$. Generating too many instances may overload the solver with useless information and exhaust computing resources. Generating too few instances will result in an "unknown", and useless, verdict. We report here how veriT copes with such quantified formulas. Several instantiation techniques are applied in turn: trigger-based, sort-based and superposition techniques.

In a quantified formula $Q\mathsf{x}\, \varphi(\mathsf{x})$, a trigger is a set of terms $T = \{t_1, \cdots t_n\}$ such that the free variables in T are the quantified variables x and each t_i is a subterm of the matrix $\varphi(\mathsf{x})$ of the quantified formula. Trigger-based instantiation consists in finding, in the formula, sets of ground terms T' that match T, i.e., such that there is a substitution σ on x, where the homomorphic extension of σ over T yields T'. Each such substitution defines an instantiation of the original quantified formula. Some verification systems allow the user to specify instantiation triggers. This is not the case in Rodin, and veriT applies heuristics to annotate quantified formulas with triggers.

If the trigger-based approach does not yield any new instance, veriT resorts to sort-based instantiation. In that case, each quantified variable is instantiated with the ground terms of the formula that have the same sort.

Finally, veriT also features a module to communicate with a superposition-based first-order logic automated theorem prover, namely the E prover [17]. It is built upon automated deduction techniques such as rewriting, subsumption, and superposition and is capable of identifying the unsatisfiability of a set of quantified and non-quantified formulas. When such a set is found satisfiable, lemmas are extracted from its output and communicated to the other reasoning modules of veriT. The E prover, like many saturation-based first-order provers, is complete for first-order logic with equality.

4.3 Unsat core extraction

Additionally to the satisfiability response, it is possible, in case the proof obligation is validated (i.e., when the formula given to the SMT solver is unsatisfiable), to ask for an *unsatisfiable core*. For instance, the sequent (1) discussed in Section 3 and translated into the SMT input on Figure 3 is valid independently of any hypothesis. The SMT input associates labels to the hypotheses and goal, 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 goal. In the present case, it would only return the goal since no named hypothesis was used. The plug-in transmits this information to the platform through a rule stating that the goal is unsatisfiable by itself.

Once this rule has been produced, the Rodin platform uses it to discharge any similar proof objective. In particular, if we modify the current sequent without modifying any predicate of the rule (in this case for instance, by changing any irrelevant invariant), the SMT solver rule will still be applicable and the SMT solver will not need to be run again. This is very important for the end user experience: when the user modifies his model, most proofs get reused and the user does not have to wait for the solvers to run again.

The unsat core production for the veriT solver is related to the proof production feature. The solver is indeed able to produce a proof, and it has moreover a facility to prune the proof of unnecessary proof steps and hypotheses. It suffices thus to check the pruned proof and collect all hypotheses in that proof to obtain a superset of the unsat core. Although not minimal in theory, this superset often corresponds to a minimal unsat core, and thus provides the plug-in with high quality information.

5 Experimental results

We collected a library of proof obligations from several academic (i.e., case studies from books, academic publications, tutorials,. . .) and industrial projects. The SMT solvers are used with a timeout of 3 seconds[4], on a dual-core Intel Core 2 Duo, cadenced at 2.93GHz, with 4GB of RAM, and running Linux Ubuntu 10.04.

[4] This timeout is unusually small for SMT solvers. Larger timeouts would provide better results, but also altering the responsiveness of the Rodin interactive platform.

	Number of proof obligations	Atelier B	alt-ergo-r217	cvc3-2011-11-21	veriT-dev-r2863	veriT & E-prover	z3-3.2	SMT Portfolio (Open)	SMT Portfolio	Portfolio
Academic	2786	474	660	434	646	553	490	271	231	136
Industrial	855	126	212	178	192	177	160	83	62	8
Total	3641	600	872	612	838	730	650	354	293	144

Fig. 5. Experimental results (number of proof obligation *not* discharged by the tools).

Figure 5 presents a summary of the results.[5] The results are detailed separately for academic and industrial projects. The second column gives the number of proof obligations, the next columns the number of them not validated by the tool heading the column, i.e., the number of proof obligations requiring human interaction after automatic application of only this tool.

The column "Atelier B" gives the number of proof obligations that were *not* discharged by the prover from Atelier B. The five following columns give, for several SMT solvers, the number of proof obligations that the solvers were not able to validate. The "SMT Portfolio" column relates the number of proof obligations unproved after trying all considered SMT solvers, whereas the "SMT Portfolio (Open)" column only consider the solvers with a permissive license, (i.e., distributed with the plug-in). The "Portfolio" column gives the remaining sequents after running both the SMT solvers and the prover from Atelier B.

On Figure 6 only the proof obligations undischarged by the prover from Atelier B are considered, and we detail for each solver (or group of solvers) the number of validated formulas.

It is worth noticing that SMT solvers altogether validate more proof obligations than the Atelier B prover. But the important and strong conclusion that can be deduced from these tables is that SMT solvers complement the Atelierb B prover. From 600 proof obligations that are not validated by the prover from Atelier B — and that required human interaction — around 75% are discharged automatically by SMT solvers. It thus *divides by four* the amount of verification conditions requiring human interaction.

Besides this complementarity, the tools have different features that justify having a portfolio of solvers: veriT has a permissive license and produces proofs, from which it is easy to extract unsatisfiable cores; cvc3 is quite efficient, but extracting unsatisfiable cores from its output is not trivial; z3 is certainly very powerful, but has a restrictive license.

[5] Notice that the z3 solver was not used at its full power since its Model Based Quantifier Instantiation feature (MBQI) was not fully functional on the latest currently available version for our system.

	Undischarged by Atelier B	alt-ergo-r217	cvc3-2011-11-21	veriT-dev-r2863	veriT & E-prover	z3-3.2	SMT Portfolio (Open)	SMT Portfolio
Academic	474	121	259	106	155	227	313	338
Industrial	126	91	99	110	105	68	118	118
Total	600	212	358	216	260	295	431	456

Fig. 6. Improvement over Atelier B (number of validated proof obligations)

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, but different translation approaches may be applied to map such constructs to a logic they handle. We presented two such approaches: a basic one that tackles simple sets, and another one that is furthermore able to handle more elaborate structures.

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 to one fourth 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]. Also, as SMT solvers can provide models when a formula is satisfiable, it would be possible, with additional engineering effort, to use such models to report counter-examples in Rodin.

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 [16].

Acknowledgement: we would like to thank the anonymous reviewers for their remarks.

References

1. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.
2. M. Armand, G. Faure, B. Grégoire, 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 *LNCS*, pages 135–150. Springer, 2011.
3. 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.
4. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard Version 2.0, 2010.
5. A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.
6. T. Bouton, D. C. B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: An Open, Trustable and Efficient SMT-Solver. In *Automated Deduction - CADE-22*, volume 5663 of *LNCS*, 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, *Proc. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *LNCS*, 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 *LNCS*, pages 217–230. Springer, 2010.
11. D. Déharbe. Integration of SMT-solvers in B and Event-B development environments. *Science of Computer Programming*, Mar. 2011.
12. M. Konrad and L. Voisin. Translation from Set-Theory to Predicate Calculus. Technical report, ETH Zurich, 2011.
13. 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.
14. C. Métayer and L. Voisin. The Event-B mathematical language, 2009. `http://deploy-eprints.ecs.soton.ac.uk/11/4/kernel_lang.pdf`.
15. G. Nelson and D. C. Oppen. Simplifications by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, Oct. 1979.
16. M. Schmalz. The logic of Event-B, 2011. Technical report 698, ETH Zürich, Information Security.
17. S. Schulz. E - A Brainiac Theorem Prover. *AI Communications*, 15(2/3):111–126, 2002.
18. The Eclipse Foundation. Eclipse SDK, 2009.
19. C. Tinelli and M. T. Harandi. A new correctness proof of the Nelson–Oppen combination procedure. In F. Baader and K. U. Schulz, editors, *Frontiers of Combining Systems (FroCoS)*, Applied Logic, pages 103–120. Kluwer Academic Publishers, Mar. 1996.