Integrating SMT solvers in Rodin ☆

David Déharbe[a], Pascal Fontaine[b], Yoann Guyot[c], Laurent Voisin[d]

[a] *Universidade Federal do Rio Grande do Norte, Natal, RN, Brazil*
[b] *University of Lorraine, Loria, Inria, France*
[c] *Cetic, Belgium*
[d] *Systerel, France*

Abstract

Formal development in Event-B generally requires the validation of a large number of proof obligations. Some tools 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 a translation of Event-B proof obligations to the language of SMT solvers, its implementation in a Rodin plug-in, and an experimental evaluation on a large sample of industrial and academic projects. On this domain, adding SMT solvers to Atelier B provers reduces significantly the number of sequents that need to be proved interactively.

Keywords: Formal methods, Event-B, SMT solving

1. Introduction

The Rodin platform [10] is an integrated design environment for the formal modeling notation Event-B [2]. Rodin is based on the Eclipse framework [25] 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).

☆This work is partly supported by the project ANR-13-IS02-0001-01, the STIC Am-Sud project 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).

Email addresses: `david@dimap.ufrn.br` (David Déharbe), `Pascal.Fontaine@inria.fr` (Pascal Fontaine), `yoann.guyot@cetic.be` (Yoann Guyot), `laurent.voisin@systerel.fr` (Laurent Voisin)

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 so-called *reasoners*. A reasoner is a plug-in that can either be standalone or use existing verification technologies through third-party tools. Most of the time, sequents are not amenable to finite domain encoding, and engines such as model checkers are not appropriate reasoners.

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

Automation Ideally, the validity status of proof obligations is computed automatically by reasoners. If human interaction is required for discharging valid 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. It is important that the reasoners are able to provide such sets of relevant facts, since they can then be used to automatically construct new proof rules to be stored and tried for after model changes. Also, other sequents (valid for the same reason) may be discharged by these rules without requiring another call to the reasoner.

In addition, when reasoners are able to generate counter-examples of failed proof obligations, this information 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. This paper extends the work presented in [15], and provides details of a translation of Event-B sequents to SMT input. The difficulty essentially lies in the way sets are translated. We here propose two approaches to tackle this challenge. Notice that these approaches could also be applied to other set-based formalisms such as the B method [1], TLA+ [20, 19], VDM [16] and Z [27].

The SMT-LIB initiative provides a standard for the input language of SMT solvers, and, in its last version [5], a command language defining a common

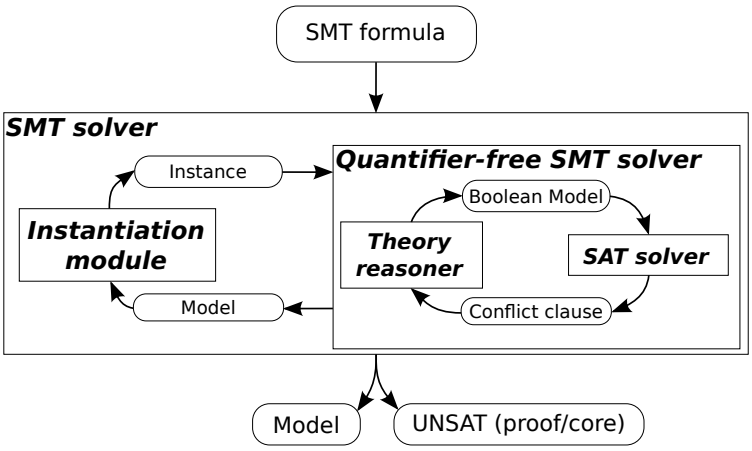

Figure 1: Schematic view of an SMT solver.

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 [12] and veriT [9]) are able to generate a comprehensive proof for validated formulas, which can 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. The plug-in is open-source, distributed under the same license as Rodin, and its source code is available in the main Rodin repository. The plug-in is easily installable by users through the update manager of the Rodin platform.

Overview. We start in Section 2 by giving some insights on the techniques employed in SMT solvers. Section 3 presents the translation of Rodin sequents to the SMT-LIB notation. Section 4 illustrates the approach through a simple example. 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 [21], and sentences in SMT-LIB are typeset using a `typewriter` font.

2. 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 of the kind of formulas that can successfully be handled by SMT solvers. A very schematic view of an SMT solver is presented in Figure 1. 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.

We refer to [4] for more information about the techniques described in this section and SMT solving in general. There are several SMT solvers supporting quantifiers; the plug-in described in this paper makes use of Alt-Ergo [8], CVC3 [6], Z3 [12], and veriT [9]. This last solver is developed by two of the authors of this paper.

2.1. Quantifier-free formulas

Historically, the first goal of SMT solvers was to provide efficient decision procedures for expressive languages, beyond pure propositional logic. Those solvers have always been based on a cooperation of a Boolean engine, nowadays typically a SAT solver (see [7] 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 conjunctively adding 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 [22, 26], 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 further 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. Models of the propositional abstraction are checked for satisfiability while they are being built, so that unsatisfiability can be detected early. Finally, theory propagation, in which the theory reasoner provides hints for the SAT solver decisions, has proved to be very worthwhile in practice.

2.2. Instantiation techniques

Within SMT solvers, solving formulas with quantifiers is done by reduction to quantifier-free formulas, using instantiation. Indeed, formula $\forall \mathbf{x}\, \varphi(\mathbf{x})$ stands for a conjunction over all combinations of values for $\mathbf{x}$. Any formula of the form $\forall \mathbf{x}\, \varphi(\mathbf{x}) \Rightarrow \varphi(\mathbf{t})$, for any terms $\mathbf{t}$, can be added conjunctively to the input without changing its truth value. To show that a formula is unsatisfiable, it

is thus sufficient to find the right instances of quantified formulas to add to the input. In that context, even if the SMT solver abstracts $\forall \mathbf{x}\, \varphi(\mathbf{x})$ to a Boolean proposition, it is able to reason about the formulas with quantifiers. The quantifier instantiation module is responsible for producing lemmas of the form $\forall \mathbf{x}\, \varphi(\mathbf{x}) \Rightarrow \varphi(\mathbf{t})$. 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). 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. Handling quantifiers within SMT solvers is still a very active research subject, and the methods to handle quantifiers vary greatly from one solver to another. We report here how veriT copes with quantified formulas. Several instantiation techniques are applied in turn: trigger-based, sort-based and superposition techniques.

The trigger-based and sort-based instantiation techniques are applied to top-most quantifiers, that is, to quantifiers that are not themselves in the scope of other quantifiers. Remember that, when checking the satisfiability of formulas, existential quantifiers with positive polarity and universal quantifiers with negative polarity can be eliminated by Skolemization. This satisfiability preserving transformation replaces suitable quantified variables by witnesses, introducing new uninterpreted symbols (constants or functions). In veriT, Skolemization automatically occurs for top-most quantifiers, whereas Skolemizable quantifiers in the scope of non-Skolemizable quantifiers are not eliminated. As a consequence, only Skolem constants are introduced, and no Skolem functions. Instantiation will remove top-most non-Skolemizable quantifiers, some quantifiers in the instance may then become top-most Skolemizable quantifiers in the process, and are in turn eliminated with the introduction of Skolem constants. This strategy is effective in practice.

In a quantified formula $Q\mathbf{x}\, \varphi(\mathbf{x})$, a trigger is a set of terms $T = \{t_1, \cdots t_n\}$ such that the free variables in T are the quantified variables $\mathbf{x}$ and each t_i is a sub-term of the matrix $\varphi(\mathbf{x})$ 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 $\mathbf{x}$, 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 falls back to sort-based instantiation. All ground terms in the formula are collected, and each quantified formula is instantiated with every term that has the same type as the quantified variable.

Finally, veriT also features a module to communicate with a superposition-based first-order logic automated theorem prover, namely the E prover [24]. 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.

2.3. Unsatisfiable core extraction

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 4 and translated into the SMT input in Figure 6 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 6, 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 validating again 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.

The unsatisfiable 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 super-set of the unsatisfiable core that is often minimal in practice. Other approaches for unsatisfiable core extraction for SMT are presented and discussed in [4].

3. Translating Event-B to SMT

Figure 2 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

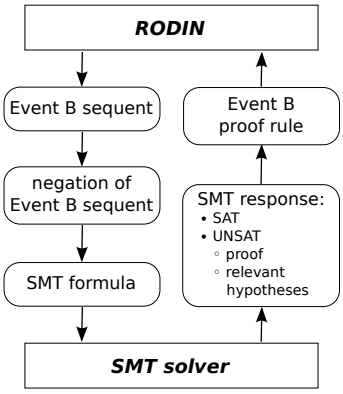

Figure 2: Schematic view of the interaction between Rodin and SMT solvers.

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.

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 [18], 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 $\{n \oftype \mathbb{Z}\}$ 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. We present successively two approaches. The simplest one, presented shortly for completeness, is based on the representation of sets as characteristic predicates [13]. 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.

3.1. The λ-based approach

This approach implements and extends the principles proposed in [13] 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 $A \oftype \mathbb{P}(\mathbb{Z}) \vdash A \cup \{a\} = A$, the translation to this extended SMT-LIB language produces:

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

where `X` denotes a sort variable. The function definitions `union` and `enum` are inserted by the translator. The former is part of a corpus of definitions for most of the set-theoretic constructs (see [13, 14] for details). The latter is created on-the-fly by the translator to denote the set $\{a\}$. Both definitions are composed of 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 this goal to

```
(not (forall ((x Int)) (iff (or (A x) (= x a)) (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.

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

Figure 3: 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).

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.

3.2. The ppTrans approach

Our second approach uses the translator *ppTrans* provided by the *Predicate Prover* available in Rodin [17]. This tool translates an Event-B formula to an equivalent formula in a subset of the Event-B mathematical language. The grammar of this subset is shown in Figure 3. Note that the sole set-theoretic symbol is the membership predicate. In addition, the translator performs *decomposition* of binary relations and *purification*, namely 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'. Not only does this approach make the plug-in independent of veriT, but it is also more general with respect to the translation of relations and functions. However, in the class of formulas suitable for the λ-based approach, *ppTrans* would produce similar results.

Maplet-hiding variables. The rewriting system implemented in *ppTrans* cannot directly transform identifiers that are of type Cartesian product. In a preprocessing 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:

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

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

For instance, assume $x \oftype \mathbb{Z} \times (\mathbb{Z} \times \mathbb{Z})$; 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:
$$\begin{array}{rl} \psi \rightsquigarrow & \forall \mathsf{fv}(\nabla(x_1)) \cdots \mathsf{fv}(\nabla(x_n)) \cdot \\ & \quad (x_1 = \nabla(x_1) \land \cdots x_n = \nabla(x_n)) \Rightarrow \psi[\nabla(x_1)/x_1] \cdots [\nabla(x_n)/x_n]. \end{array}$$

  Ex. $\psi \equiv a = b \land a \in S$ with typing $\{a \oftype S, b \oftype S, S \oftype \mathbb{P}(\mathbb{Z} \times \mathbb{Z})\}$:

$$\begin{array}{rl} \psi \rightsquigarrow & \forall x_0, x_1, x_2, x_3 \cdot \\ & \quad (a = x_0 \mapsto x_1 \ \land \ b = x_2 \mapsto x_3) \Rightarrow \\ & \quad\quad (x_0 \mapsto x_1 = x_2 \mapsto x_3 \land x_0 \mapsto x_1 \in S) \end{array}$$

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 4 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:

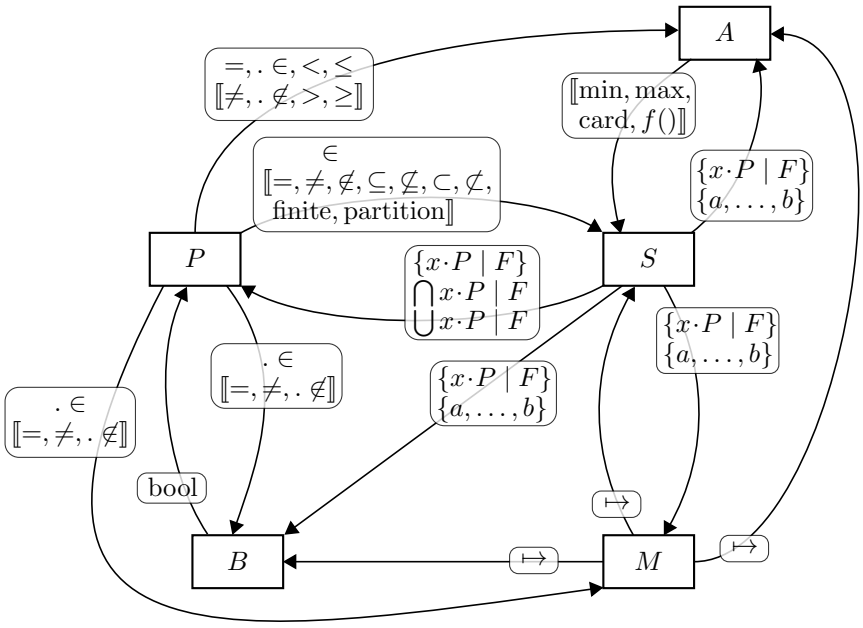

Figure 4: The different syntactic categories and the symbols relating them: A for arithmetic expressions, P for predicates, S for set expressions, B for Boolean expressions and M for maplet expressions. *ppTrans* removes all occurrences of the constructs delimited by double-brackets.

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 $a \mapsto b \in S$.

Appendix A presents the rewriting rules implemented in *ppTrans*. For readability, they are grouped thematically and the presentation order is not the same as the rewriting order. The symbols relating the syntactic categories P (predicates) and S (sets) are reduced to membership ($\in$) and equality ($=$) by application of the rules A.1–A.8 (see Appendix A.1). Moreover rules A.1 and A.2 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 $\notin$. Applications of the equality symbol between syntactic categories S, M and B are removed by rules A.9–A.18 (Appendix A.2). 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 the rules in Appendix A.3. Rules A.19–A.21 first perform purification, and are followed by the application of rules A.22–A.29, responsible for the elimination of the symbols. Appendix A.4 contains rules to rewrite applications of the set membership symbol according to the rightmost argument. Rules A.30–A.39 expand the definitions for the different kinds of relation symbols. Rules A.40–A.61

handle miscellaneous other cases. Finally, Appendix A.5 presents the rules to rewrite applications of the set membership symbol according to both arguments, where the first argument is always a maplet. Again, through the application of rules A.62 to A.77, several symbols may be eliminated from the proof obligations.

All the rules in Appendix A 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 (Rules A.17, A.19 – A.21, A.44) 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
$a \oftype S$	
$b \oftype T$	
$c \oftype U$	$a \in A$
$A \oftype \mathbb{P}(S)$	$a \mapsto b \in r$
$r \oftype \mathbb{P}(S \times T)$	$a \mapsto b \mapsto c \in s$
$s \oftype \mathbb{P}(S \times T \times U)$	

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:

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

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

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

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:

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

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 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)))))))
```

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.

4. 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 job queue and several active jobs. We define a given set *JOBS* to represent 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` : $active \cap queue = \varnothing$ (a job can not be both active and queued)

One of the events of the system describes that a job leaves the queue and becomes active. It is specified as follows:

Event *SCHEDULE* $\widehat{=}$ (some queued job j becomes active)
any
j
where
`grd1` : $j \in queue$ (the job j is in the queue)
then
`act1` : $active := active \cup \{j\}$ (the job becomes active)
`act2` : $queue := queue \setminus \{j\}$ (the job is removed from the queue)
end

To verify that the invariant labeled `inv3` is preserved by the *SCHEDULE* event, the following sequent must be proved valid:

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

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

$$\begin{array}{l} active \subseteq JOBS \land \\ queue \subseteq JOBS \land \\ active \cap queue = \varnothing \land \\ j \in queue \land \\ \neg((active \cup \{j\}) \cap (queue \setminus \{j\}) = \varnothing) \ . \end{array}$$

This proof obligation does not contain sets of sets and both approaches apply. Figure 5 presents the SMT-LIB input obtained when the approach described in section 3.1 is applied. Line 2 contains the declaration of the sort corresponding to the given set *JOBS*. Lines 3–5 contain the declarations of the uninterpreted function symbols of the proof obligation, and are produced using the typing environment. Note that the sets *queue* and *active* are represented by unary predicate symbols. Next, the macros corresponding to the set operators $\varnothing$, $\in$,

```
(set-logic QF_AUFLIA)
(declare-sort JOBS 0)
(declare-fun active (JOBS) Bool)
(declare-fun queue (JOBS) Bool)
(declare-fun j () JOBS)
(define-fun (par (X) (emptyset ((x X)) Bool false)))
(define-fun (par (X) (in ((x X) (s (X Bool))) Bool (s x))))
(define-fun (par (X) (inter ((s1 (X Bool)) (s2 (X Bool))) (X Bool)
                       (lambda ((x X)) (and (s1 x) (s2 x))))))
(define-fun (par (X) (setminus ((s1 (X Bool)) (s2 (X Bool))) (X Bool)
                       (lambda ((x X)) (and (s1 x) (not (s2 x)))))))
(define-fun (par (X) (union ((s1 (X Bool)) (s2 (X Bool))) (X Bool)
                       (lambda ((x X)) (or (s1 x) (s2 x))))))
(define-fun enum ((elem JOBS)) Bool (= elem j))
(define-fun enum0 ((elem0 JOBS)) Bool (= elem0 j))
(assert (= (inter active queue) emptyset))
(assert (in j queue))
(assert (not (= (inter (union active enum) (setminus queue enum0))
                emptyset)))
(check-sat)
```

Figure 5: SMT-LIB input produced using the λ-based approach.

```
(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)
```

Figure 6: SMT-LIB input produced using the *ppTrans* approach.

$\cap$, $\setminus$, and $\cup$ are defined in lines 6–13. Lines 14–15 are the definitions of a macro that represents the singleton set $\{j\}$ (it occurs twice in the formula). Lines 16–19 are the result of the translation of the proof obligation itself.

Figure 6 presents the SMT-LIB input resulting from the translation approach described in section 3.2. 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 decide which approach is applied. When the sequents contains no sets or only simple sets (i.e., no sets of sets), the λ-based approach is applied. Otherwise, the plug-in employs the *ppTrans* 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 [15] 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[1]. These developments originate from 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 proce-

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

dures. 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 of the SMT solvers seems a valuable 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.

	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.

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 improvements 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 [11]. 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 [23].

Acknowledgments: This paper is a revised and extended version of [15]. We thank the anonymous reviewers of paper [15] and of this paper for their careful read and their remarks.

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, 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] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Computer Aided Verification (CAV)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[7] 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.

[8] F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout. The Alt-Ergo automated theorem prover, `http://alt-ergo.lri.fr`.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] D. Déharbe. Integration of SMT-solvers in B and Event-B development environments. *Science of Computer Programming*, 78(3):310 – 326, 2013.

[15] 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.

[16] C. B. Jones. *Systematic software development using VDM*. Prentice Hall International Series in Computer Science. Prentice Hall, 1991.

[17] M. Konrad and L. Voisin. Translation from Set-Theory to Predicate Calculus. Technical report, ETH Zurich, 2011.

[18] 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.

[19] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Boston, Mass., 2002.

[20] S. Merz. On the logic of TLA$^+$. *Computers and Informatics*, 22:351–379, 2003.

[21] C. Métayer and L. Voisin. The Event-B mathematical language, 2009. `http://deploy-eprints.ecs.soton.ac.uk/11/4/kernel_lang.pdf`.

[22] G. Nelson and D. C. Oppen. Simplifications by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, Oct. 1979.

[23] M. Schmalz. The logic of Event-B, 2011. Technical report 698, ETH Zürich, Information Security.

[24] S. Schulz. E - A Brainiac Theorem Prover. *AI Communications*, 15(2/3):111–126, 2002.

[25] The Eclipse Foundation. Eclipse SDK, 2009.

[26] 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.

[27] J. Woodcock and J. Davies. *Using Z: specification, refinement, and proof.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

Appendix A. Rewriting rules in *ppTrans*

Appendix A.1. Rules for sets and predicates

$$x \neq y \rightsquigarrow \neg(x = y) \tag{A.1}$$
$$x \notin s \rightsquigarrow \neg(x \in s) \tag{A.2}$$
$$s \subseteq t \rightsquigarrow s \in \mathbb{P}(t) \tag{A.3}$$
$$s \not\subseteq t \rightsquigarrow \neg(s \in \mathbb{P}(t)) \tag{A.4}$$
$$s \subset t \rightsquigarrow s \in \mathbb{P}(t) \wedge \neg(t \in \mathbb{P}(s)) \tag{A.5}$$
$$s \not\subset t \rightsquigarrow \neg(s \in \mathbb{P}(t)) \vee t \in \mathbb{P}(s) \tag{A.6}$$
$$\text{finite}(s) \rightsquigarrow \forall a \cdot \exists b, f \cdot f \in s \rightarrowtail a..b \tag{A.7}$$
$$\begin{aligned} \text{partition}(s, s_1, s_2, \ldots s_n) \rightsquigarrow \quad & s = s_1 \cup s_2 \cup \ldots \cup s_n \wedge \\ & s_1 \cap s_2 = \varnothing \wedge \cdots \wedge s_1 \cap s_n = \varnothing \wedge \\ & \vdots \\ & s_{n-1} \cap s_n = \varnothing \end{aligned} \tag{A.8}$$

Appendix A.2. Elimination of equalities

The notation $\forall X_T \cdot P(X)$ stands for $\forall \mathsf{fv}(\nabla(e)) \cdot P(\nabla(e))$, with $\mathcal{T}(X) = T$.

$$s = t \rightsquigarrow \forall X_T \cdot X \in s \Leftrightarrow X \in t \tag{A.9}$$
$$x_1 \mapsto x_2 = y_1 \mapsto y_2 \rightsquigarrow x_1 = y_1 \wedge x_2 = y_2 \tag{A.10}$$
$$x = f(y) \rightsquigarrow y \mapsto x \in f \tag{A.11}$$
$$\text{bool}(P) = \text{bool}(Q) \rightsquigarrow P \Leftrightarrow Q \tag{A.12}$$
$$\text{bool}(P) = \text{TRUE} \rightsquigarrow P \tag{A.13}$$
$$\text{bool}(P) = \text{FALSE} \rightsquigarrow \neg P \tag{A.14}$$
$$x = \text{FALSE} \rightsquigarrow \neg(x = \text{TRUE}) \tag{A.15}$$
$$x = \text{bool}(P) \rightsquigarrow x = \text{TRUE} \Leftrightarrow P \tag{A.16}$$
$$e_l[\text{bool}(s)] = e_r \rightsquigarrow \forall^? e_l[\text{bool}(s)^*] = e_r \tag{A.17}$$
$$e = e \rightsquigarrow \top \tag{A.18}$$

Appendix A.3. Elimination of mixed arithmetic symbols
Purification

MOP stands for either min, max, card or a function application, $\prec$ for either $\leq$ or $<$ and $\succ$ for either $\geq$ or $>$.

$$e_l[\text{MOP}(s)] = e_r \rightsquigarrow \forall^? e_l[\text{MOP}(s)^*] = e_r \tag{A.19}$$
$$a[\text{MOP}(s)] \prec b \rightsquigarrow \forall^? a[\text{MOP}(s)^*] \prec b \tag{A.20}$$
$$a \prec b[\text{MOP}(s)] \rightsquigarrow \forall^? a \prec b[\text{MOP}(s)^*] \tag{A.21}$$

Elimination of mixed operators

$$n = \mathrm{card}(s) \rightsquigarrow \exists f \cdot f \in s \rightarrowtail\!\!\!\!\!\rightarrow 1..n \tag{A.22}$$
$$n = \min(s) \rightsquigarrow n \in s \wedge n \leq \min(s) \tag{A.23}$$
$$n = \max(s) \rightsquigarrow n \in s \wedge \max(s) \leq n \tag{A.24}$$
$$a \succ b \rightsquigarrow b \prec a \tag{A.25}$$
$$\max(s) \prec a \rightsquigarrow \forall x \cdot x \in s \Rightarrow x \prec a \tag{A.26}$$
$$\min(s) \prec a \rightsquigarrow \exists x \cdot x \in s \wedge x \prec a \tag{A.27}$$
$$a \prec \min(s) \rightsquigarrow \forall x \cdot x \in s \Rightarrow a \prec x \tag{A.28}$$
$$a \prec \max(s) \rightsquigarrow \exists x \cdot x \in s \wedge a \prec x \tag{A.29}$$

Appendix A.4. Rules based on the right argument of set membership

Elimination of membership in relations

The notation _func(f) specifies that f is a function, and abbreviates: $\forall A_U, B_V, C_V \cdot A \mapsto B \in f \wedge A \mapsto C \in f \Rightarrow B = C$, where $\mathcal{T}(f) = U \times V$.

$$e \in s \twoheadleftarrow\!\!\!\!\!\leftrightarrow t \rightsquigarrow e \in s \twoheadleftarrow\!\!\!\!\!\leftrightarrow t \wedge t \subseteq \mathrm{ran}(e) \tag{A.30}$$
$$e \in s \twoheadleftarrow\!\!\!\!\!\leftrightarrow t \rightsquigarrow e \in s \leftrightarrow t \wedge s \subseteq \mathrm{dom}(e) \tag{A.31}$$
$$e \in s \leftrightarrow\!\!\!\!\!\twoheadrightarrow t \rightsquigarrow e \in s \leftrightarrow t \wedge t \subseteq \mathrm{ran}(e) \tag{A.32}$$
$$e \in s \rightarrowtail\!\!\!\!\!\rightarrow t \rightsquigarrow e \in s \twoheadrightarrow t \wedge \_\mathrm{func}(e^{-1}) \tag{A.33}$$
$$e \in s \twoheadrightarrow t \rightsquigarrow e \in s \rightarrow t \wedge t \subseteq \mathrm{ran}(e) \tag{A.34}$$
$$e \in s \twoheadrightarrow\!\!\!\!\!\rightarrow t \rightsquigarrow e \in s \twoheadrightarrow t \wedge t \subseteq \mathrm{ran}(e) \tag{A.35}$$
$$e \in s \rightarrowtail t \rightsquigarrow e \in s \rightarrow t \wedge \_\mathrm{func}(e^{-1}) \tag{A.36}$$
$$e \in s \rightarrowtail\!\!\!\!\!\twoheadrightarrow t \rightsquigarrow e \in s \twoheadrightarrow t \wedge \_\mathrm{func}(e^{-1}) \tag{A.37}$$
$$e \in s \rightarrow t \rightsquigarrow e \in s \twoheadrightarrow t \wedge s \subseteq \mathrm{dom}(e) \tag{A.38}$$
$$e \in s \twoheadrightarrow t \rightsquigarrow e \in s \leftrightarrow t \wedge \_\mathrm{func}(e) \tag{A.39}$$

Other membership rewriting rules

The notation $\forall X_T \cdot P(X)$ stands for $\forall \mathsf{fv}(\nabla(e)) \cdot P(\nabla(e))$, with $\mathcal{T}(X) = T$.
Likewise, notation $\exists X_T \cdot P(X)$ stands for $\exists \mathsf{fv}(\nabla(e)) \cdot P(\nabla(e))$, with $\mathcal{T}(X) = T$.

$$e \in s \rightsquigarrow \top \quad \text{if } \mathcal{T}(e) = s \tag{A.40}$$

$$e \in \varnothing \rightsquigarrow \bot \tag{A.41}$$

$$e \in \mathbb{P}(t) \rightsquigarrow \forall X_T \cdot X \in e \Rightarrow X \in t \quad \text{if } \mathcal{T}(e) = \mathbb{P}(T) \tag{A.42}$$

$$e \in s \leftrightarrow t \rightsquigarrow \forall X_T \cdot X \in e \Rightarrow X \in s \times t \quad \text{if } \mathcal{T}(e) = \mathbb{P}(T) \tag{A.43}$$

$$e \in f \rightsquigarrow \exists^{?} e^{*} \in f \quad \text{if } f \text{ is an identifier} \tag{A.44}$$

$$e \in \mathbb{N} \rightsquigarrow 0 \leq e \tag{A.45}$$

$$e \in \mathbb{N}_1 \rightsquigarrow 0 < e \tag{A.46}$$

$$e \in \{x \cdot P \mid f\} \rightsquigarrow \exists x \cdot P \wedge e = f \tag{A.47}$$

$$e \in \left(\bigcap x \cdot P \mid f\right) \rightsquigarrow \forall x \cdot P \Rightarrow e \in f \tag{A.48}$$

$$e \in \left(\bigcup x \cdot P \mid f\right) \rightsquigarrow \exists x \cdot P \wedge e \in f \tag{A.49}$$

$$e \in \mathrm{union}(s) \rightsquigarrow \exists x \cdot x \in s \wedge e \in x \tag{A.50}$$

$$e \in \mathrm{inter}(s) \rightsquigarrow \forall x \cdot x \in s \Rightarrow e \in x \tag{A.51}$$

$$e \in r\,[s] \rightsquigarrow \exists X_T \cdot X \in s \wedge X \mapsto e \in r \quad \text{if } \mathbb{P}(T) = \mathcal{T}(\mathrm{dom}(r)) \tag{A.52}$$

$$e \in f(s) \rightsquigarrow \exists X_T \cdot s \mapsto X \in f \wedge e \in X \quad \text{if } T = \mathbb{P}(\mathcal{T}(e)) \tag{A.53}$$

$$e \in \mathrm{ran}(r) \rightsquigarrow \exists X_T \cdot X \mapsto e \in r \quad \text{if } \mathbb{P}(T) = \mathcal{T}(\mathrm{dom}(r)) \tag{A.54}$$

$$e \in \mathrm{dom}(r) \rightsquigarrow \exists X_T \cdot e \mapsto X \in r \quad \text{if } \mathbb{P}(T) = \mathcal{T}(\mathrm{ran}(r)) \tag{A.55}$$

$$e \in \{a_1, \ldots, a_n\} \rightsquigarrow e = a_1 \vee \cdots \vee e = a_n \tag{A.56}$$

$$e \in \mathbb{P}_1(s) \rightsquigarrow e \in \mathbb{P}(s) \wedge [\exists X_T \cdot X \in e] \quad \text{if } \mathbb{P}(T) = \mathcal{T}(e) \tag{A.57}$$

$$e \in a..b \rightsquigarrow a \leq e \wedge e \leq b \tag{A.58}$$

$$e \in s \setminus t \rightsquigarrow e \in s \wedge \neg(e \in t) \tag{A.59}$$

$$e \in s_1 \cap \ \ldots \ \cap s_n \rightsquigarrow e \in s_1 \wedge \ \ldots \ \wedge e \in s_n \tag{A.60}$$

$$e \in s_1 \cup \ \ldots \ \cup s_n \rightsquigarrow e \in s_1 \vee \ \ldots \ \vee e \in s_n \tag{A.61}$$

Appendix A.5. Rules based on both arguments of set membership

$$e \mapsto f \in s \times t \rightsquigarrow e \in s \wedge f \in t \tag{A.62}$$

$$e \mapsto f \in r \rhd t \rightsquigarrow e \mapsto f \in r \wedge \neg(f \in t) \tag{A.63}$$

$$e \mapsto f \in s \lhd r \rightsquigarrow e \mapsto f \in r \wedge \neg(e \in s) \tag{A.64}$$

$$e \mapsto f \in r \rhd t \rightsquigarrow e \mapsto f \in r \wedge f \in t \tag{A.65}$$

$$e \mapsto f \in s \lhd r \rightsquigarrow e \mapsto f \in r \wedge e \in s \tag{A.66}$$

$$e \mapsto f \in \mathrm{id} \rightsquigarrow e = f \tag{A.67}$$

$$e \mapsto f \in r^{-1} \rightsquigarrow f \mapsto e \in r \tag{A.68}$$

$$e \mapsto f \in \mathrm{pred} \rightsquigarrow e = f + 1 \tag{A.69}$$

$$e \mapsto f \in \mathrm{succ} \rightsquigarrow f = e + 1 \tag{A.70}$$

$$\begin{aligned} e \mapsto f \in r_1 \lhd \cdots \lhd r_n \rightsquigarrow\ & e \mapsto f \in r_n \vee \\ & e \mapsto f \in \mathrm{dom}(r_n) \lhd r_{n-1} \vee \\ & e \mapsto f \in \mathrm{dom}(r_n) \cup \mathrm{dom}(r_{n-1}) \lhd r_{n-2} \vee \\ & \vdots \\ & e \mapsto f \in \mathrm{dom}(r_n) \cup \cdots \mathrm{dom}(r_2) \lhd r_1 \end{aligned} \tag{A.71}$$

$$\begin{aligned} e \mapsto f \in r_1; \ldots; r_n \rightsquigarrow\ & \exists X^1_{T_1}, \ldots X^{n-1}_{T_{n-1}} \cdot e \mapsto X^1 \in r_1 \wedge \ldots \wedge X^{n-1} \mapsto f \in r_n \\ & \text{if } \mathcal{T}(\mathrm{ran}(r_i)) = \mathbb{P}(T_i), 1 \leq i \leq n \end{aligned} \tag{A.72}$$

$$e \mapsto f \in r_1 \circ \ldots \circ r_n \rightsquigarrow e \mapsto f \in r_n; \ldots; r_1 \tag{A.73}$$

$$(e \mapsto f) \mapsto g \in \mathrm{prj}_1 \rightsquigarrow e = g \tag{A.74}$$

$$(e \mapsto f) \mapsto g \in \mathrm{prj}_2 \rightsquigarrow f = g \tag{A.75}$$

$$e \mapsto (f \mapsto g) \in p \otimes q \rightsquigarrow e \mapsto f \in p \wedge e \mapsto g \in q \tag{A.76}$$

$$(e \mapsto f) \mapsto (g \mapsto h) \in p \parallel q \rightsquigarrow e \mapsto g \in p \wedge f \mapsto h \in q \tag{A.77}$$