# Integration of SMT-Solvers in B and Event-B Development Environments

David Déharbe

*Universidade Federal do Rio Grande do Norte*
*Departamento de Informática e Matemática Aplicada*
*Natal, RN, Brazil*

**Abstract**

Software development in B and Event-B generates proof obligations that have to be discharged using theorem provers. The cost of such developments depends directly on the degree of automation and efficiency of theorem proving techniques for the logics in which these lemmas are expressed. This paper presents and formalizes an approach to transform a class of proof obligations essentially similar to those generated in the Rodin platform into the input language of a category of automatic theorem provers known as SMT solvers. The work presented in the paper handles proof obligations with Booleans, integer arithmetics, basic sets and relations and has been implemented as a plug-in for Rodin.

*Key words:* Formal methods, Event-B, SMT-solving

## 1. Introduction

Formal software development using frameworks such as B [1], Event-B [2] and Z [30] produces large quantities of proof obligations (POs), typically expressed in a first-order language including arithmetic and set-theoretic constructs. Indeed, integrated development environments such as Atelier-B [10] and Rodin [11] include theorem provers able to discharge automatically a significant portion of these POs. The remaining POs need the intervention of the users to be addressed, though. For each such PO, three outcomes are possible. First, when there is an error in the model, the PO is not valid. The user must inspect the PO to understand the cause of the error, correct the model and verify it again. Second, the PO may be valid, but cannot be automatically proved because the proof system lacks some axioms, as the specification logic is incomplete. In that case, it is sometimes possible to patch the prover with additional rules so that it finds a proof for the verification condition. Care must be taken not to make the system unsound. Third, the PO may be true, but cannot be proved automatically within the space and time bounds set by the user, due to computational complexity of the verification system. In that case, the user has the possibility to interact with the verification sub-system and help the theorem prover find the proof. All such interactions are time-consuming and impact negatively on the productivity of the software development process. Progress in automatic theorem proving

*Email addresses:* david.deharbe@pq.cnpq.br (David Déharbe)

techniques for formal software development framework is therefore key to increase the effectiveness, application and dissemination of formal methods (see e.g. [28] for an account of application of several verification techniques to Event-B model checking). This may be achieved with (1) more efficient verification algorithms, (2) better integration with the development framework (providing counter-examples and dependencies between proofs and specification elements), (3) native support for more expressive logics.

The Satisfiability Modulo Theory (SMT) approach to theorem proving [17, 15, 14] successfullt addresses many software-related problems. SMT-solvers [6, 5, 7, 8, 18, 13] are tools that implement this approach. They combine a Boolean satisfiability engine to handle the propositional structure of the PO, optimized decision procedures for individual theories, such as arrays and arithmetics, a framework to combine these decision procedures [24], and other optimizations, such as theory propagation [26]. Some solvers are also able to build a proof of their result. The international SMT-LIB initiative provides a common input format [29] language and a repository of benchmarks for SMT-solvers.

Recently, the formal methods community has shown interest in applying SMT-solvers in the verification of POs, which seem to be good candidates to achieve progress in at least two of the above mentioned directions: efficiency and integration. SMT-solving techniques have also drawn the interest of other families of formal methods practitioners, as shown by a recent experiment to apply a SMT-solver to verify a system specified in Alloy [19]. Moreover, the inclusion in the SMT-LIB, the standard input format for SMT-solvers, of a theory for state-based specification languages has recently been proposed [22]. However this proposition considers many specification constructs such as sets, sequences and maps that are not yet fully handled by current SMT-solving techniques.

The goal of the work presented in this paper is to extend an existing approach [16] to apply *state-of-the-art* SMT-solvers to POs, restricted to a subset of the logic constructs of B and Event-B: Booleans, linear integer arithmetics, basic sets and basic relations, plus mixed operators relating sets with integers such as cardinality. It considers the POs generated in the Rodin platform and shows how they may be mapped to the SMT-LIB language, and thus verified using existing SMT-solvers. This approach should also be readily applied in other development environments, e.g. Atelier B, as the proof obligations are expressed in the same logic.

The rest of the paper is organized as follows. The next two sections lay the ground for the technical development of this work. Section 2 presents the formal development methods B and Event-B and describes the class of proof obligations that are handled in this work. Section 3 then introduces the SMT approach to automatic theorem proving and the SMT-LIB format. The general principles of this translation are described in section 4 and the detailed presentation of its formalization in section 5. Section 6 describes an implementation of this translation within Rodin [11] in a plug-in that verifies proof obligations using existing SMT-solvers. This implementation was applied to a set of benchmarks and the result of their verification with an existing SMT-solver are presented in section 7. Conclusions and future work are discussed in section 8.

## 2. Formal Model-Driven Software Design with B and Event-B

The B method for software development [1] is based on the B *Abstract Machine Notation* (AMN) and the use of formally proved refinements up to a model sufficiently concrete that programming code can be automatically generated from it. Event-B [2] is closely related to B, but with an emphasis on system modelling. Both are based on refinement calculus, first order logic, integer arithmetic, set theory, and their constructs are similar to those of the Z notation [30]. B and Event-B models are finite state machines. The state is a tuple of values of a collection of variables, and the transitions may assign new values to those variables. The language also provides modularization and structuring constructs similar to those of programming languages. B is supported by a number of tools [10, 3], while all the tool support for Event-B is integrated in the Rodin platform [11], a development environment based on Eclipse [31].

### *2.1. Overview*

A B or Event-B model defines a set of valid states, including initial states, and operations (named events in Event-B) that usually provoke a state transition. The design process starts with an abstract yet *functional* model of the system. The developer needs to check several verifications conditions on such a model: that all expressions are well-defined, that the model constraints are satisfiable, that the initial states are valid and that operations do not map valid states to invalid states. For each such verification, the tools generate automatically proof obligations that are then verified using theorem proving technology.

Once an initial functional model has been constructed and verified, B and Event-B provide constructs to define *refinement* modules. A refinement is always associated to another, more abstract, model and specifies either the formalization of new functional requirements or a design decision: either about the concrete representation of the state, or about the algorithmic realization of an operation (or both). Verification conditions are proved to establish that each refinement conforms to the refined module.

### *2.2. The B and Event-B notations*

Essentially, a B or Event-B module contains two main parts: a state space definition and the operations available to access and modify state. It may additionally contain auxiliary clauses in many forms (parameters, constants, assertions), but those are essentially for practical purposes (i.e. to promote modularity, reuse, etc.) and do not extend the expressive power of the notation. In the remainder, we will restrict our discussion to the core clauses of the module specification.

The specification of the state components appears in the VARIABLES and INVARIANT clauses. The former enumerates the state components, and the latter defines restrictions on the possible values they can take. If $V$ denotes the state variables of a machine, the invariant is a predicate on $V$. Verifications carried out throughout the development process have the intention of checking that no invalid state will ever be reached as long as the operations of the machine are used as specified.

**CONTEXT** *definitions*
**SETS** $COLOR$
**CONSTANTS** $red, yellow, green$
**AXIOMS** $COLOR = \{red, yellow, green\}$
**END**

**MACHINE** *traffic_light*
**SEES** *definitions*
**VARIABLES** *light*
**INVARIANTS** $light \in COLOR$
**INITIALISATION** $light :\in COLOR$
**EVENTS**
*tored* = **WHEN** $light = yellow$ **THEN** $light := red$ **END**
*toyel* = **WHEN** $light = green$ **THEN** $light := yellow$ **END**
*togre* = **WHEN** $light = red$ **THEN** $light := green$ **END**
**END**

Figure 1: An example of a functional model in Event-B

For the specification of the initialisation as well as the operations, the notation offers a set of so-called *substitutions*. These are "imperative-like" constructs with translation rules that define their semantics as the effect they have on the values of any (global or local) variables to which they are applied. The semantics of the substitutions is defined by the *substitution calculus*, formalizing how the different substitution forms rewrite to formulas in first-order logic. At an abstract level of specification, it is common to use non-deterministic substitutions. Let $S$ denote a substitution, $E$ an expression, then $[S]E$ denotes the result of applying $S$ to $E$. For instance, an operation that would increment a counter variable $v$ can be specified as $v := v + 1$. Indeed, the basic substitution is very similar to the side-effect free assignment construct found in imperative programming languages. Applying such a substitution to an expression consists in substituting the target variable $v$ with the source expression. For instance, $[v := v + 1]v \geqslant 0$ simplifies to $v + 1 \geqslant 0$. Besides the basic substitution, the notations provide more elaborate substitution constructions, such as non-deterministic substitution **ANY** $v$ **WHERE** $C$ **THEN** $S$ **END**, which applies substitution $S$ with variable $v$ having any value that satisfies predicate $C$. Substitution $v :\in V$, where $V$ is a set, is equivalent to **ANY** $x$ **WHERE** $x \in V$ **THEN** $v := x$ **END**.

The example shown in Figure 1 illustrates these concepts with the Event-B model of a traffic light. Event-B specification modules are either contexts, for auxiliary definitions, or machines, with the actual functional specification. In the example, the context is named *definitions* and defines a set named *COLOR* with three elements *red*, *green* and *yellow*. The behavior is specified in the machine named *traffic_light*. Its model has a single variable, named *light*, and the invariant states that its value must be an element of the set *COLOR*. The initialization non-deterministically chooses one value in this set and assigns it to variable *light*. The possible transitions are specified in the model as three events. Each event has a guard and assigns a new value to the state.

Two consecutive refinements for the example machine are presented in figure 2. The first refinement introduces a bounded integer variable, named *count* to represent the value of the variable *light* from the abstract model: it is called

**MACHINE** $traffic\_light\_r1$
**REFINES** $traffic\_light$
**CONSTANTS** $color\_refine, color\_step$
**PROPERTIES**
$color\_refine \in COLOR \rightarrow \mathbb{N} \wedge$
$color\_refine = \{green \mapsto 0, yellow \mapsto 1, red \mapsto 2\}$
**VARIABLES** $count$
**INVARIANT** $count \in \mathbb{N} \wedge count \in 0..2 \wedge count = color\_refine(light)$
**INITIALISATION** $count := 0$
**EVENTS**
$tored$ = **WHEN** $count = 1$ **THEN** $count := (count + 1) \bmod 3$ **END**
$toyel$ = **WHEN** $count = 0$ **THEN** $count := (count + 1) \bmod 3$ **END**
$togre$ = **WHEN** $count = 2$ **THEN** $count := (count + 1) \bmod 3$ **END**
**END**

**MACHINE** $traffic\_light\_r2$
**REFINES** $traffic\_light\_r1$
**VARIABLES** $count$
**INVARIANT** $count \in \mathbb{N} \wedge count \in 0..2$
**INITIALISATION** $count := 0$
**EVENTS**
$step$ =
**REFINES** $tored, toyel, togre$
**THEN** $count := (count + 1) \bmod 3$ **END**
**END**

Figure 2: Two successive Event-B refinements for the machine shown in figure 1: the first is a data refinement, and the second is used to merge events.

a data refinement since it provides a representation of the data that is closer to an actual implementation. The second refinement illustrates how to take advantage of an opportunity to factor definitions: the three events of the system are merged into a single one, called *step*.

### *2.3. Verification*

The correctness of a full design with the B method is established by proving that functional models are realizable and consistent and that each refinement is compatible with the module it refines. To this end, proof obligations need to be generated and checked.

A functional model is considered realizable if the different constraints that its component must obey are satisfiable. It is considered consistent when:

1. The initialization actions take the machine into a valid state.
2. No operation may take the machine from a valid state to an invalid state, as long as the user provided parameters and the machine variables are such that the guard *GRD* for application of the substitution *S* corresponding to this operation evaluates to true.

Refinements are also subject to verification. The B and Event-B method generates proof obligations establishing that the result of a refinement is compatible with the original specification. Although the methods have different

definitions of refinements, the corresponding proof obligations have essentially the same structure and underlying logic.

The verification of such proof obligations is delegated to theorem provers. The efficiency and effectiveness of verification of such tools depends on the language elements that are used to model the system. The richer the language, the more complex the data model and the associated operations, and the more difficult is the verification of such proof obligations.

Section 2.4 describes the structure of the representation of the proof obligations in the Rodin platform.

In addition to the generation of proof obligation and the verification thereof using automatic or interactive theorem provers, it is also possible to carry out verification tasks using a model checker and animator such as ProB [23]. ProB combines on-the-fly state space construction and constraint solving. If, one the one hand, this approach is applicable only to finite models, on the other hand, it is also able to verify automatically a larger class of properties, including dynamic behavioral properties expressed in, e.g., temporal logic.

### *2.4. Proof Obligations in Rodin*

One of the roles of the Rodin platform [11] is to manage the POs produced in a development. The abstract syntax of the format is described hereafter. A PO is structured in four sections[1]:

| | | | | | |
|---|---|---|---|---|---|
| $L$ | ::= | $T^+$ | $E$ | $H^*$ | $G$ |
| | | *(theories)* | *(typing environment)* | *(hypothesis)* | *(goal)* |

The theories are pre-defined names corresponding to different fragments of the specification logic. This work only considers theories that correspond to Booleans, integers, basic sets (i.e. no set of sets is allowed), basic relations (i.e. only $n$-ary relations over simple sets are allowed), but it could be straightforwardly extended to include real numbers if the specification logic would consider them. The theories of a PO can be computed by a static syntactical analysis of the hypothesis and the goal.

The scope of this work is a fragment of the specification logic, and the following grammar defines the possible corresponding typing environments:

| | | | |
|---|---|---|---|
| $E$ | ::= | $D^*$ | *(typing environment : a sequence of declarations)* |
| $D$ | ::= | $T \mid V$ | *(declaration: a carrier set or a variable declaration)* |
| $T$ | ::= | **set** *name* | *(carrier set declaration : a user-defined name)* |
| $V$ | ::= | *name* $S$ | *(declaration: a name followed by a sort)* |
| $S$ | ::= | $B \mid \mathbb{P}(B)$ | *(sort: a basic sort, or a set thereof)* |
| $B$ | ::= | $C \mid B \times B$ | *(basic sort: a carrier set or a binary relation)* |
| $C$ | ::= | $\mathbb{Z} \mid$ **BOOL** $\mid$ *name* | *(carrier set: integers, Booleans, or user-defined)* |

[1] Regexp superscript operators $^+$ and $^*$ indicate respectively one or more and zero or more of the preceding element.

Sorts **BOOL** and $\mathbb{Z}$ are pre-defined. In B, new sorts (also called basic types) are carrier sets and may be introduced either as deferred sets (only their name is given, and they are assumed to be non-empty) or as enumeration (and the only values in the sorts are those that are enumerated, and they are pairwise distinct).

Finally, the hypothesis and the goal are first-order formulas. The language for formulas is specified by the following grammar, where non-terminals $\varphi$ and $\tau$ stand respectively for formulas and terms. POs are well-typed and no type checking or type inference is needed.

$$
\begin{array}{lll}
\varphi ::= & \neg\varphi \mid \varphi \Rightarrow \varphi \mid \varphi \Leftrightarrow \varphi & \\
& \mid \varphi \wedge \cdots \wedge \varphi \mid \varphi \vee \cdots \vee \varphi & \textit{(Boolean connectors)} \\
& \mid \exists\, x \bullet \varphi \mid \forall\, x \bullet \varphi & \textit{(quantifiers)} \\
& \mid \tau = \tau \mid \tau \neq \tau & \textit{(generic relational operators)} \\
& \mid \tau < \tau \mid \tau > \tau \mid \tau \leqslant \tau \mid \tau \geqslant \tau & \textit{(integer relational operators)} \\
& \mid \tau \subseteq \tau \mid \tau \subset \tau \mid \tau \in \tau & \textit{(set relational operators)}
\end{array}
$$

The considered terms have sort **BOOL** and $\mathbb{Z}$, user-defined carrier sets as well as their powerset and cartesian products, and powerset of such cartesian products. Basic sets may be ranges of integers, or defined in extension or intentionally.

$$
\begin{array}{lll}
\tau ::= & \textit{name} \mid & \\
& \textbf{TRUE} \mid \textbf{FALSE} \mid \textbf{bool}(\varphi) \mid & \textit{(Boolean terms)} \\
& \textit{num} \mid -\tau \mid \tau - \tau \mid \tau\ \textbf{div}\ \tau \mid \tau\ \textbf{mod}\ \tau \mid \tau + \cdots + \tau \mid \tau * \cdots * \tau \mid & \textit{(integer terms)} \\
& \emptyset \mid \{\tau, \cdots, \tau\} \mid \{x \mid \varphi\} \mid \tau \cup \cdots \cup \tau \mid \tau \cap \cdots \cap \tau \mid \tau \setminus \tau \mid & \textit{(set terms)} \\
& \tau..\tau \mid & \textit{(integer range)} \\
& \tau \times \tau \mid \tau \mapsto \tau \mid & \textit{(cartesian product, pair)} \\
& \mathrm{dom}\,\tau \mid \mathrm{ran}\,\tau \mid & \textit{(relation domain, range)} \\
& \tau[\tau] \mid & \textit{(relational image)} \\
& \tau \lhd \tau \mid \tau \rhd \tau \mid & \textit{(relation restriction on domain, range)} \\
& \tau \mathrel{\lhd\!\!\!-} \tau \mid \tau \mathrel{\rhd\!\!\!-} \tau \mid & \textit{(relation subtraction on domain, range)} \\
& \tau^{-1} \mid \tau \fatsemi \tau \mid \tau \Leftarrow \tau \mid & \textit{(inverse, composition, overwrite)} \\
& \mathrm{id}(\tau) \mid & \textit{(identity relation)} \\
& \tau \leftrightarrow \tau \mid & \textit{(set of relations)} \\
& \tau \rightarrow \tau \mid \tau \nrightarrow \tau \mid & \textit{(total, partial functions)} \\
& \tau \rightarrowtail \tau \mid \tau \mathrel{\nrightarrow\!\!\!\!\!\!\rightarrowtail} \tau \mid & \textit{(total, partial injective functions)} \\
& \tau \twoheadrightarrow \tau \mid \tau \mathrel{\nrightarrow\!\!\!\!\!\!\twoheadrightarrow} \tau \mid & \textit{(total, partial surjective functions)} \\
& \tau \rightarrowtail\!\!\!\!\!\!\twoheadrightarrow \tau \mid & \textit{(bijective functions)} \\
& \lambda\, x \bullet (\varphi \mid \tau) & \textit{(lambda expression)}
\end{array}
$$

Finally there are mixed operators that are neither basic set operators nor integer arithmetic operators; they are:

$$\varphi \quad ::= \quad \mathbf{finite}(\tau) \qquad \textit{(test for finiteness)}$$

$$\tau \quad ::= \quad \mathbf{min}(\tau) \mid \mathbf{max}(\tau) \mid \qquad \textit{(set minimum and maximum)}$$
$$\mathbf{card}(\tau) \qquad \textit{(cardinality)}$$

## 3. SMT-Solvers

### *3.1. The SMT approach to theorem proving*

The Satisfiability Modulo Theory (SMT) problem is a decision problem to determine if a given logic formula is satisfiable with respect to a combination of theories expressed in first-order logic. Theories of interest for the work described in this paper include uninterpreted functions with equality and integer arithmetics. Since the validity of a proof obligation can be decided by checking the unsatisfiability of its negation, SMT-solvers are natural candidates for discharging the verification conditions generated in the application of the B and Event-B methods.

SMT-solvers can for example handle a formula like

$$x \leq y \ \wedge \ y \leq x + f(x) \ \wedge \ P(h(x) - h(y)) \ \wedge \ \neg P(0) \ \wedge \ f(x) = 0$$

which contains linear arithmetics on reals $(0, +, -, \leq)$, and uninterpreted symbols $(P, h, f)$. SMT-solvers use decision procedures for the disjoint languages (for instance, congruence closure for uninterpreted symbols [25], and simplex for linear arithmetics) and combine them to build a decision procedure for the union of the languages.

There are a number of available SMT-solvers (e.g. [7, 5, 8, 18, 13]. They differ mostly by the theories they handle, their efficiency (which may vary depending on the theories they handle), the ability to handle quantified formulas as well as to construct certificates (proofs) of their results. SMT-COMP is a yearly competition where the different SMT-solvers may participate in different categories (corresponding to different theories). This competition uses the SMT-LIB library of benchmarks, and the corresponding format is the de *facto standard* input language for SMT-solvers. In this work, it is the language used to input proof obligations to SMT-solvers and is described in Section 3.2.

### *3.2. The SMT-LIB Format*

The SMT-LIB format [4, 29] is the result of an international effort to establish a common input language for SMT-solvers. In addition, this initiative also collects and classifies proof obligations expressed in this format, thus establishing a benchmark library for SMT-solvers. Both the format and the benchmarks are used as a reference to establish quantitative measures of SMT-solvers and their evolution. The SMT-LIB format is thus the *de facto* standard input language for SMT-solvers. This work considers the newly published version 2.0 ([16] considered version 1.2).

SMT-solvers are interactive and are used through a sequence of commands that thus form a scripting language. A session with a SMT-solver is initiated with a sequence of declarations of different entities: logic, sorts, functions, predicates. The logic is a pre-established name, to which are associated sort, function and predicate declarations, possibly some syntactical restrictions, as well as a semantics. For instance, the logic QF_IDL corresponds to quantifier-free formulas with integer arithmetic terms where the constraints bound numerically subtractions between integer-sorted terms. Also, a logic for POs from the Vienna Development Method [9] has been proposed for inclusion in the SMT-LIB [22] . This theory includes finite sets, lists and maps and its goal is close to that of this work; it is however currently not supported by existing SMT-solvers.

SMT-LIB has commands to declare a sort by giving its name and arity, and a function by giving its name and signature (i.e. the list of the sorts of the parameters followed by the sort of the result). The sort *Bool* of arity zero and the binary equality operator belong to every pre-defined logic, such as QF_IDL. In addition, the language offers the possibility to define the semantics of such symbols axiomatically with first-order formulas.

In addition to commands for defining the specification language, the scripting language has commands to add (conjunctively) and remove hypotheses and goal in a global stack of sets of formulas. Such formulas are first-order multi-sorted logic formulas expressed using both the symbols of the declared logic and the additional symbols declared in the benchmark. Groups of formulas may be incrementally stacked and eventually verified through a command to activate the solver. This causes the solver to check if the conjunction of the stacked formulas is satisfiable or not. There are also commands to query the result once the solver has completed its execution. In addition to the satisfiability status, clients (such as the Rodin platform) may query a model when the stack of formulas is satisfiable, and obtain an unsatisfiable subset when it is not.

## 4. Rationale of the Translation

On the one hand, SMT-solvers integrate and combine reasoning engines for a number of logics, such as arrays, bit-vectors, abstract data types, and different fragments of integer and real-number arithmetics. On the other hand, Rodin POs may contain arbitrary integer arithmetic expressions, arbitrary combinations of set constructs, including derived entities such as relations, functions and sequences, as well as operators mapping sets to integers and vice-versa. The remainder of this section contains a discussion on the rationale for mapping Booleans, arithmetic, basic sets and relations to proof obligations that are amenable for efficient verification by current SMT-solvers: sets and relations are represented as predicates, expressed using lambda expressions, and set and relation operators are expressed using macros.

### *4.1. Booleans*

The logic of Event-B and B contains terms of sort Boolean, where they are different syntactical entities than formulas. In addition, these notations provide the operator **bool** that “casts” formulas to Boolean terms. The same

distinction occurs in the Rodin POs. This translation maps both entities (formulas and Boolean terms) to terms sorted as Boolean in the SMT-LIB logic. This approach should take better advantage of the efficient handling of complex Boolean structures in SMT-solvers.

### *4.2. Arithmetic*

Considering arithmetics, the SMT-LIB benchmarks are divided into several divisions according to the class of constraints: difference logic, linear arithmetics, non-linear arithmetics. The benchmarks are further divided according to whether the formulas are quantified or quantifier-free and whether the numeric sort is integers or real numbers. This information is available to SMT-solvers in a benchmark attribute; they may use it to select the most efficient procedure for the given benchmark. The translation systematically sets this attribute to quantified linear integer arithmetic as it reflects the addressed class of POs.

### *4.3. Basic sets*

Currently, no SMT solver provides direct support for set theory. A possible approach is to map set constructs to symbols of a theory that can be handled in one of the existing SMT-solvers. It is possible to map sets to arrays of Booleans indexed by the elements of the domain of the set (see e.g. in [12, 22]). Another solution is to represent sets by their characteristic predicate, i.e. a Boolean-valued function $f$ taking as argument a value of the corresponding carrier set. This is the approach used to discharge set-theoretic arguments in a proof assistant embedding a SMT-solver [21] as well as in the *Predicate Prover*, available in Rodin and Atelier-B. The former solution has the advantage of being more general, as it allows nesting sets. The latter solution can only deal with basic sets (no set of sets) but provides a direct mapping to first-order logic with uninterpreted functions and predicates, for which efficient reasoning engines are available: this paper investigates this approach. Even if a SMT solver eventually supports all the constructs found in Rodin POs, this does not cancel the usefulness of this work, as the SMT-LIB POs produced using the proposed encoding may be easier to prove than in the general case.

Sets and set operators are thus translated to predicates and Boolean connectors as usual: false for the empty set, disjunction for union, implication for set inclusion, predicate application for set membership, etc. To implement this approach, we employ extensions to the SMT-LIB library implemented in the veriT SMT-solver [8], namely macro definitions and lambda expressions. veriT processes such constructs, applying macro expansion and beta-reduction, producing formulas conforming to the SMT-LIB format that may be directly processed in veriT or by any SMT-solver equipped with the right decision procedures.

For instance, union is a macro used in the translation of set union (Section 5):

$$\mathsf{union} \quad \equiv \quad (\mathsf{lambda}\ (\mathsf{X}\ (\mathsf{'s\ Bool}))(\mathsf{Y}\ (\mathsf{'s\ Bool}))(\mathsf{lambda}\ (\mathsf{e\ 's})\ (\mathsf{or}\ (\mathsf{X\ e})\ (\mathsf{Y\ e}))))$$

Here, union is declared as a macro with two arguments X and Y. Both arguments are sorted as ('s Bool), i.e as a unary predicate, where 's is a sort variable. The macro union expands to the lambda expression given in the right

Empty set, intersection and set difference:

```
empty    ≡  (lambda (e 's) false)                                  /*   ∅      */
inter    ≡  (lambda  (X ('s Bool))(Y ('s Bool))                    /*   X ∩ Y  */
                (lambda (e 's) (and (X e) (Y e))))
setminus ≡  (lambda  (X ('s Bool))(Y ('s Bool))                    /*   X \ Y  */
                (lambda (e 's) (and (X e) (not (Y e)))))
```

Set membership, inclusion and strict inclusion:

```
in       ≡  (lambda (e 's) (X ('s Bool))(X e))                     /*   e ∈ X  */
subseteq ≡  (lambda  (X ('s Bool))(Y ('s Bool))                    /*   X ⊆ Y  */
                (forall (e 's) (implies (X e) (Y e))))
subset   ≡  (lambda  (X ('s Bool))(Y ('s Bool))                    /*   X ⊂ Y  */
                (and (subseteq X Y) (not (= X Y))))
```

Range of integers:

```
range    ≡  (lambda  (lo Int)(hi Int)                              /*   lo..hi */
                (lambda (i Int) (and (<= lo i)(<= i hi))))
```

Figure 3: Macros used in the translation of set expressions.

hand side of the expression: it denotes a unary predicate with formal parameter e of sort 's, defined as disjunction of the appliction of X and Y to e. veriT implements a type inference mechanism to handle macros with sort variables similar to that for "let polymorphism" [27]. The remaining macros are presented in figure 3. In this figure, and in the following, X and Y denote unary predicates (representing sets), e, f, g and h denote variables (representing set elements), and 's, 't and 'u denote sort variables (representing carrier sets).

A last observation is in order. The result of the application of macro expansion may result in formulas where equality is applied at the predicate level. For instance, assume that $S$ is a set over some sort s, the formula $S \cup \emptyset = S$, valid in set theory, expands to:

$$\text{(lambda (e 's) (or } (p_S\ e) \text{ false))} \quad = \quad \text{(lambda (e 's) } (p_S\ e)),$$

where $p_S$ is the unary predicate symbol characterizing set $S$. The original equality between sets results in an equality between formulas. In first-order logic, this equality is expressed with universal quantification and equivalence. This can be performed by rewriting, and corresponds to the application of the set extensionality axiom schema:

(= X Y) is rewritten to (forall (e 't) (iff (X e) (Y e)) whenever $X$ and $Y$ are unary predicates on sort t.

Applying this rule followed by beta reduction to the example yields the first-order logic tautology:

$$\text{(forall (e 's) (iff (or } (p_S\ e) \text{ false)} (p_S\ e))).$$

*4.4. Basic relations*

Cartesian product is the construction that introduces relations. Since this work is concerned solely with *basic relations*, it is restricted to products combining basic sets, a basic set with a basic relation, or two basic relations, as described in section 2.4.

Following the line used to handle basic sets, where each set is represented by a unary predicate, a binary relation could naturally be represented as a binary predicate. However this approach does not work well for some constructions of the specification logic, e.g. for maplets, or pairs, that is elements of basic relations that would not have a proper representation. A solution is thus to introduce a binary parametric sort, say Pair in the SMT-LIB proof obligation as well as a parametric constructor, say pair. In the following, pairs will be denoted with symbols p, p′ and q, and relations with symbols r, r1 and r2. The following axioms may be introduced as assumptions in the SMT-LIB proof obligation:

$$\text{(forall (p (Pair 's 't))(exists (e 's) (f 't)(= p (pair e f))))} \tag{1}$$

$$\text{(forall (e 's) (f 't)(g 's) (h 't)(implies (= (pair e f)(pair g h)) (and (= e g)(= f h))))} \tag{2}$$

In addition, symbols fst and snd are introduced. They may also be axiomatized as:

$$\text{(forall (e 's) (f 't) (= (fst (pair e f)) e))} \tag{3}$$

$$\text{(forall (e 's) (f 't) (= (snd (pair e f)) f))} \tag{4}$$

In the translation, a binary relation in the Rodin proof obligation is represented as a predicate, such that its domain is (an instance) of the parametric sort Pair. The cartesian product of two sets may then be translated with the help of the following macro that, given two predicates X and Y representing two sets $X$ and $Y$, builds the predicate representing the set $X \times Y$.

$$\text{prod} \quad \equiv \quad \text{(lambda (X ('s Bool))(Y ('t Bool)) (lambda (p (Pair 's 't)) (and (X (fst p)) (q (snd p)))))} \tag{5}$$

For instance, assume $a$, $b$, and $c$ are all elements of a carrier set $S$ and the following proof obligation:

$$\{a \mapsto b, a \mapsto c\} \subseteq \{a\} \times \{a, b, c\}.$$

Formula $\{a \mapsto b, a \mapsto c\} \subseteq \{a\} \times \{a, b, c\}$ would then be translated to the following (assuming $a$, $b$, $c$ and $S$ are

respectively represented as a, b, c and S in the translation):

```
(subseteq
  (lambda (p (Pair S S))(or (=  p (pair a b))(=  p (pair a c))))
  (prod (lambda (e S)(=  e a))(lambda (e S)(or (=  e a)(=  e b)(=  e c)))))
```

Expanding the prod macro, followed by beta-reduction, the previous expression rewrites to:

```
(subseteq
  (lambda (p (Pair S S)) (or (=  p (pair a b)) (=  p (pair a c))))
  (lambda (p (Pair S S))
    (and  (=  (fst p) a)
          (or (=  (snd p) a) (=  (snd p) b) (=  (snd p) c)))))
```

Now, expanding the subseteq macro, the resulting expression is:

```
(forall (p (Pair S S)
  (implies  (or (=  p (pair a b)) (=  p (pair a c))))
            (and  (=  (fst p) a)
                  (or (=  (snd p) a) (=  (snd p) b) (=  (snd p) c)))))
```

At this point, the expression is in first-order logic and it is possible to rely on the SMT-solvers quantifier instantiation capacities to handle such proof obligation, together with the three axioms mentioned at the start of the section.

However it is possible to simplify further the proof obligation. Indeed the combination of axiom 1 and Skolemization is used to substitute all terms of sort Pair by applications of the pair function. Next, applying repeatedly axiom 2 oriented left-to-right as a rewrite rule, every equality between applications of the pair function is substituted by conjunction of equalities between the corresponding pairs of terms. Conducting this process on the running example yields a formula with a single interpreted symbol (equality).

```
(forall (e S) (f S)
  (implies  (or (and(=  e a) (=  f b))(and(=  e a) (=  f c)))
            (and  (=  e a) (or (=  f a) (=  f b) (=  f c)))))
```

Again, macros are employed in the translation of the relational operators (see figure 4). The next section provides a formalized account of the translation described hitherto.

Domain, range and relational image:

```
dom  ≡  (lambda (r ((Pair 's 't) Bool))                          /*  dom r     */
          (lambda (e 's) (exists (f 't)(r (pair e f)))))
ran  ≡  (lambda (r ((Pair 's 't) Bool))                          /*  ran r     */
          (lambda (f 't) (exists (e 's)(r (pair e f)))))
img  ≡  (lambda (r ((Pair 's 't) Bool)(X ('s Bool))              /*  r[X]      */
          (lambda (f 't) (exists (e 's)(and (X e) (r (pair e f))))))
```

Domain restriction and subtraction, range restriction and subtraction:

```
domr ≡  (lambda (r ((Pair 's 't) Bool) (X ('s Bool))             /*  X ◁ r     */
          (lambda (p (Pair 's 't) (and (r p)(X (fst p))))))
doms ≡  (lambda (r ((Pair 's 't) Bool)(X ('s Bool))              /*  X ⩤ r     */
          (lambda (p (Pair 's 't) (and (r p)(not (X (fst p)))))))
ranr ≡  (lambda (r ((Pair 's 't) Bool)(X ('t Bool))              /*  r ▷ X     */
          (lambda (p (Pair 's 't) (and (r p)(X (snd p))))))
rans ≡  (lambda (r ((Pair 's 't) Bool)(X ('t Bool))              /*  r ⩥ X     */
          (lambda (p (Pair 's 't) (and (r p)(not (X (snd p)))))))
```

Inverse, composition, overwrite and identity:

```
inv  ≡  (lambda (r ((Pair 's 't) Bool)                           /*  r⁻¹       */
          (lambda (p (Pair 's 't) (r (pair (snd p)(fst p))))))
comp ≡  (lambda (r1 ((Pair 's 't) Bool)(r2 ((Pair 't 'u) Bool)   /*  r1 ⨾ r2   */
          (lambda (p (Pair 's 'u))
            (exists (e 't) (and (r1 (pair (fst p) x)) (r2 (pair e (snd p)))))))
ovr  ≡  (lambda (r1 ((Pair 's 't) Bool)(r2 ((Pair 's 't) Bool)   /*  r1 ✶ r2   */
          (lambda (p (Pair 's 'u)) (or (r2 p)
            (and (r1 p)(not (exists (q (Pair 's 't))
                  (and (r2 q) (=  (fst q) (fst p))))))))))
id   ≡  (lambda (X ('t Bool))                                    /*  id(X)     */
          (lambda (p (Pair 't 't)) (and (X (fst p)) (=  (fst p) (snd p)))))
```

Figure 4: Macros used in the translation of operators on relations

Auxiliary properties on relations:

```
funp  ≡  (lambda (r ((Pair 's 't ) Bool))
           (forall (p (Pair 's 't))(p' (Pair 's 't))
             (implies (and (r p) (r p')) (implies (=  (fst p) (fst p'))(=  (snd p) (snd p'))))))
injp  ≡  (lambda (r ((Pair 's 't ) Bool))(funp (inv r)))
totp  ≡  (lambda (X ('s Bool)) (r ((Pair 's 't) Bool))
           (forall (p (Pair 's 't)) (=  (r p) (X (fst p)))))
surp  ≡  (lambda (Y ('t Bool)) (r ((Pair 's 't) Bool))
           (forall (p (Pair 's 't)) (=  (r p) (Y (snd p)))))
```

Figure 5: Auxiliary macros used in the definition of relation constructors

### *4.5. Sets of relations*

The B notation provides several arrows-like operators to denote different kinds of sets of relations, according to the desired properties. For instance, $X \leftrightarrow Y$ denotes the set of relations between $X$ and $Y$, while $X \rightarrow Y$ denotes the set of total functions between these two sets. Each such operator takes as argument two sets and yields a (high-order) predicate on relations.

Now consider the following formula:

$$\{a \mapsto b\} \in \{a\} \leftrightarrow \{b\} \tag{6}$$

Combining lambda expressions with pre-processing, (first-order) formulas such as 6 may be translated using the macros presented in figures 5 and 6. First, figure 5 contains a series of auxiliary macros that are helpful to characterize elementary properties of relations such as being functional (macro funp), injective (macro injp), total (macro totp), and surjective (macro surp). Next, these auxiliary macros are combined to define macros that expand to characterization for $\leftrightarrow$ (relation, macro rel), $\nrightarrow$ (partial function, macro pfun), $\rightarrow$ (total function, macro tfun), $\rightarrowtail$ (partial injective function, macro pinj), $\rightarrowtail$ (total injective function, macro tinj), $\twoheadrightarrow$ (partial surjective function, macro psur), $\twoheadrightarrow$ (total surjective function, macro tsur) and $\rightarrowtail\!\!\!\!\twoheadrightarrow$ (bijective function, macro bij).

Assuming $a$ and $b$ are translated to a and b, both of sort S, formula 6 would then be translated to:

```
(in (lambda (r ((Pair S S) Bool)(=  p (pair a b)))) (rel (lambda (e S) (=  e a)) (lambda (e S) (=  e b)))).
```

The expansion of macro rel yields:

```
(in  (lambda (p ((Pair S S) Bool)(=  p (pair a b))))
   (lambda (r ((Pair S S)Bool))(forall (p (Pair S S))
      (implies(r p) (and ((lambda (e S) (=  e a)) (fst p))((lambda (f S) (=  f b)) (snd p)))))))).
```

Sets of relations, functions (partial/total, injective/surjective/bijective):

```
rel  ≡  (lambda (X ('s Bool)) (Y ('t Bool))                                  /*  X ↔ Y  */
          (lambda (r ((Pair 's 't) Bool))
            (forall (p (Pair 's 't)) (implies (r p) (and (X (fst p))(Y (snd p)))))))
pfun ≡  (lambda (X ('s Bool)) (Y ('t Bool))                                  /*  X ⇸ Y  */
          (lambda (r ((Pair 's 't) Bool)) (and ((rel X Y) r) (funp r))))
tfun ≡  (lambda (X ('s Bool)) (Y ('t Bool))                                  /*  X → Y  */
          (lambda (r ((Pair 's 't) Bool)) (and ((pfun X Y) r) (totp X r))))
pinj ≡  (lambda (X ('s Bool)) (Y ('t Bool))                                  /*  X ⤔ Y  */
          (lambda (r ((Pair 's 't) Bool)) (and ((pfun X Y) r) (injp r))))
tinj ≡  (lambda (X ('s Bool)) (Y ('t Bool))                                  /*  X ↣ Y  */
          (lambda (r ((Pair 's 't) Bool)) (and ((pinj X Y) r) (totp X r))))
psur ≡  (lambda (X ('s Bool)) (Y ('t Bool))                                  /*  X ⤀ Y  */
          (lambda (r ((Pair 's 't) Bool)) (and ((pfun X Y) r) (surp Y r))))
tsur ≡  (lambda (X ('s Bool)) (Y ('t Bool))                                  /*  X ↠ Y  */
          (lambda (r ((Pair 's 't) Bool)) (and ((psur X Y) r) (totp X r))))
bij  ≡  (lambda (X ('s Bool)) (Y ('t Bool))                                  /*  X ⤖ Y  */
          (lambda (r ((Pair 's 't) Bool)) (and ((tsur X Y) r) ((tinj X Y) r))))
```

Figure 6: Macros used in the translation of arrow operators

Beta-reduction is applied to the two inner-most lambda-expressions:

```
(in  (lambda (p ((Pair S S) Bool)(=  p (pair a b))))
     (lambda (r ((Pair S S) Bool))(forall (p (Pair S S))(implies(r p) (and (=  (fst p) a)) (=  (snd p) b))))).
```

Alpha-conversion of the first lambda-expression, followed by expansion of macro in yields:

```
(forall (p (Pair S S))(implies((lambda (q ((Pair S S) Bool)(=  q (pair a b)))) p) (and (=  (fst p) a)) (=  (snd p) b))).
```

Again, beta-reduction is applied to the inner-most lambda-expression, resulting in the following first-order formula (which is a consequence of axioms 3 and 4):

```
(forall (p (Pair S S))(implies(=  p (pair a b)) (and (=  (fst p) a)) (=  (snd p) b)))
```

### *4.6. Processing syntactic sugar*

Finally, macros may also be used to express the translation of syntactic constructs found in the logic of the proof obligations but that do not have counterpart in SMT-LIB. One such construct is the disequality operator ≠; the definition of the corresponding macro neq is given in figure 7.

$$\text{neq} \quad \equiv \quad (\text{lambda } ((\text{t1 Bool}) (\text{t2 Bool})) (\text{not } (= \text{ t1 t2}))) \qquad\qquad /* \quad t_1 \neq t_2 \quad */$$

Figure 7: Macro used in the translation of disequality

## 5. Formalizing the Translation

This section presents the formalization of the translation of Rodin lemmas to SMT-LIB format extended with macros and lambda expressions. The translation is as a tree traversal, recursing over the syntactic structure of the lemma. This traversal is specified as a set of rules that follow the style of structural operational semantics: each rule is composed of a possibly empty set of hypothesis and a conclusion, separated by a horizontal line.

### *5.1. Preliminary Definitions and Notations*

The rules propagate an evaluation context $\Gamma$ that gathers incrementally the contents of the different sections that compose the SMT-LIB format. They are: sorts, a set of SMT-LIB identifiers; funs, a map from SMT-LIB function symbols to SMT-LIB sorts; assumptions, a set of formulas in SMT-LIB syntax; formula, the goal, a formula in SMT-LIB syntax. In addition, the context maintains a mapping nm from Rodin symbols to SMT-LIB identifiers. $\Gamma_I$ denotes the initial context and is such that[2]:

$$\begin{aligned}
\Gamma_I = \{\ \text{nm} \quad = \quad \{\ & \mathbb{Z} \mapsto \text{Int}, \mathbf{BOOL} \mapsto \text{Bool}, \\
& \mathbf{TRUE} \mapsto \text{true}, \mathbf{FALSE} \mapsto \text{false}, \\
& \neg \mapsto \text{not}, = \mapsto =, \neq \mapsto \text{neq}, \wedge \mapsto \text{and}, \vee \mapsto \text{or}, \Rightarrow \mapsto \text{implies}, \Leftrightarrow \mapsto \text{iff}, \\
& \exists \mapsto \text{exists}, \forall \mapsto \text{forall}, \\
& = \mapsto =, < \mapsto <, \leqslant \mapsto <=, > \mapsto >, \geqslant \mapsto >=, \\
& \subset \mapsto \text{subset}, \subseteq \mapsto \text{subseteq}, \in \mapsto \text{in}, \\
& + \mapsto +, * \mapsto *, \mathbf{div} \mapsto /, \mathbf{mod} \mapsto \%, _ - _ \mapsto -, -_ \mapsto \tilde{} \\
& \cup \mapsto \text{union}, \cap \mapsto \text{inter}, .. \mapsto \text{range}, \\
& \setminus \mapsto \text{setminus}, \emptyset \mapsto \text{empty}, \\
& _ \mapsto _ \mapsto \text{pair}, \times \mapsto \text{prod}, \text{dom} \mapsto \text{dom}, \text{ran} \mapsto \text{ran}, _[_] \mapsto \text{img}, \\
& \lhd \mapsto \text{domr}, \lhd \mapsto \text{doms}, \rhd \mapsto \text{ranr}, \rhd \mapsto \text{rans}, \\
& _^{-1} \mapsto \text{inv}, _ \fatsemi _ \mapsto \text{comp}, \lhd\!\!\lhd \mapsto \text{ovr}, \mathbf{id} \mapsto \text{id}, \\
& \leftrightarrow \mapsto \text{rel}, \rightarrow \mapsto \text{tfun}, \nrightarrow \mapsto \text{pfun}, \rightarrowtail \mapsto \text{tinj}, \rightarrowtail\!\!\!\!\!\!\nrightarrow \mapsto \text{pinj}, \twoheadrightarrow \mapsto \text{tsur}, \nrightarrow\!\!\!\!\!\!\twoheadrightarrow \mapsto \text{psur}, \rightarrowtail\!\!\!\!\!\!\twoheadrightarrow \mapsto \text{bij}, \\
& \lambda \mapsto \text{lambda}\} \\
\text{sorts} \quad = \quad & \{\text{Int}, \text{Bool}, \text{Pair 'u 's}\}, \\
\text{funs} \quad = \quad & \{\}, \text{assumptions} = \{\}, \text{formula} = \{\} \quad \}
\end{aligned}$$

[2]To facilitate reading, some operator symbols are decorated with _ characters as argument placeholders.

In the following, $\Gamma.\mathsf{sec}$ denotes the content of Section $\mathsf{sec}$ of $\Gamma$; when $\mathsf{sec}$ is a map, $\Gamma[\mathsf{sec} \oplus s]$ denotes update of $\mathsf{sec}$ with the map $s$, otherwise it denotes inclusion of $s$ to the set $\mathsf{sec}$; $\Gamma[\mathsf{sec} \ominus s]$ $\mathsf{sec}$ denotes removal of $s$ from $\mathsf{sec}$. We assume the existence of a function *btype* that, given an expression $e$ in a Rodin expression, returns the basic type $e$. Also *fresh* is a binary predicate, with arguments a context $\Gamma$, and a set of symbols $S$, and tests if all symbols in $S$ are fresh with respecto to $\Gamma$. Symbols that are part of the SMT-LIB language are typeset in a sans serif font.

### *5.2. Translation Rule for a PO*

Rule 1 specifies the operator $[\![]\!]_L$ which, given a PO $L$, composed of the sections $T\,E\,H^*G$, results in a context $\Gamma$ that represents the components of SMT-LIB format for the Rodin PO $L$:

$$1\ \frac{L = T\,E\,H^*\,G \qquad [\![E;\ \Gamma_I]\!]_E = \Gamma_0 \qquad [\![H^*;\ \Gamma_0]\!]_H = \Gamma_1 \qquad [\![G;\ \Gamma_1]\!]_G = \Gamma}{[\![L]\!]_L = \Gamma}$$

The operator $[\![]\!]_L$ is defined using auxiliary operators $[\![]\!]_E$, $[\![]\!]_H$ and $[\![]\!]_G$, responsible for translating the typing environment $E$, the hypotheses $H^*$ and the goal $G$ respectively. Operator $[\![]\!]_E$ has as arguments a sequence of declarations and an initial context, and returns a new context enriched with the given declarations. Its definition is given in section 5.3. Operator $[\![]\!]_H$ also has two arguments: a sequence of hypothesis formulas, and a context; it yields a new context enriched with the translation of the given hypothesis. Finally, operator $[\![]\!]_G$ takes as argument a goal formula, a context, and results in a context enriched with the translation of this formula. The definition of these last two operators is specified in section 5.4.

### *5.3. Translation Rules for the Typing Environment*

To facilitate reading, the translation rules follow a naming convention. Rodin types and SMT-LIB sorts will be denoted by symbols based on $s$ and $\mathsf{s}$, respectively; names, formulas, terms, and bound variables will be denoted by $n$, $\varphi$, $\tau$ and $x$ when referring to Rodin entities and by $\mathsf{n}$, $\mathsf{f}$, $\mathsf{t}$ and $\mathsf{x}$ when referring to SMT-LIB entities.

Rules 3 and 2 specify that the typing environment lemma is translated by a sequential traversal, applying the translation operator $[\![]\!]_D$ to each declaration:

$$2\ \frac{[\![D^*;\ \Gamma]\!]_E = \Gamma_0 \qquad [\![D;\ \Gamma_0]\!]_D = \Gamma_1}{[\![D^*D;\ \Gamma]\!]_E = \Gamma_1} \qquad 3\ \frac{}{[\![;\ \Gamma]\!]_E = \Gamma}$$

An auxiliary operator, named $[\![]\!]_T$, that associates B data types with SMT-LIB sorts, is used in the translation rules for declarations. Given a type $s$ and an environment $\Gamma$, $[\![s;\ \Gamma]\!]_T$ is a SMT-LIB sort that satisfies the following rules 4, 5 and 6. Rule 4 states that for a carrier set, the context $\Gamma$ shall associate a corresponding sort. The next two rules defines that sets are represented by predicates and pairs are encoded by the polymorphic functional sort $\mathsf{Pair}$.

$$4\ \frac{s \in \mathrm{dom}\,\Gamma.\mathsf{nm}}{[\![s;\ \Gamma]\!]_T = \Gamma.\mathsf{nm}(s)}$$

$$5\ \frac{\Gamma.\mathsf{nm}(t) = \mathsf{s}}{[\![\mathbb{P}\,s;\ \Gamma]\!]_T = (\mathsf{s}\ \mathsf{Bool})} \qquad 6\ \frac{\Gamma.\mathsf{nm}(s) = \mathsf{s} \qquad \Gamma.\mathsf{nm}(s') = \mathsf{s}'}{[\![s \times s';\ \Gamma]\!]_T = (\mathsf{Pair}\ \mathsf{s}\ \mathsf{s}')}$$

The declarations of a Rodin lemma, grouped in the typing environment, either new carrier sets (i.e. basic types), or pairs $n, s$, where $n$ is the name of the declared entity, and $s$ is its basic type. Rule 7 describes the case of a carrier set declaration: a new sort is introduced in the SMT-LIB context. Otherwise, $n$ is a new function symbol of the corresponding SMT-LIB sort $\mathsf{s}$ (rule 8).

$$7\ \frac{\mathit{fresh}(\Gamma, \{\mathsf{s}\})}{[\![\textbf{set}\ s;\ \Gamma]\!]_D = \Gamma[\mathsf{nm} \oplus \{s \mapsto \mathsf{s}\}]}\ \text{(basic type)}$$

$$8\ \frac{\mathit{fresh}(\Gamma, \{\mathsf{n}\}) \qquad [\![s;\ \Gamma]\!]_T = \mathsf{s}}{[\![n\ s;\ \Gamma]\!]_D = \Gamma[\mathsf{nm} \oplus \{s \mapsto \mathsf{s}\}][\mathsf{funs} \oplus \{\mathsf{n} \mapsto \mathsf{s}\}]}\ \text{(set element)}$$

*5.4. Translation Rules for Hypothesis and Goal*

Rules 9 and 10 specify that the hypothesis section is translated by a sequential traversal, applying the translation operator $[\![]\!]_\tau$ to each hypothesis, and yielding a formula $\mathsf{f}$ and a context $\Gamma$. The former is added to the latter as a result of the translation.

$$9\ \frac{}{[\![;\ \Gamma]\!]_H = \Gamma} \qquad 10\ \frac{[\![\varphi^*;\ \Gamma]\!]_H = \Gamma_0 \qquad [\![\varphi;\ \Gamma_0]\!]_\tau = \mathsf{f};\ \Gamma_1}{[\![\varphi^* \varphi;\ \Gamma]\!]_H = \Gamma_1[\mathsf{assumptions} \oplus \{\mathsf{f}\}]}$$

The goal section of the lemma is also translated using operator $[\![]\!]_\tau$, and the resulting expression is set as the goal formula in the context:

$$11\ \frac{[\![\varphi;\ \Gamma]\!]_\tau = \mathsf{f};\ \Gamma_1}{[\![\varphi;\ \Gamma]\!]_G = \Gamma_1[\mathsf{formula} \oplus \{\mathsf{f}\}]}$$

*5.5. Translation Rules for Terms*

The operator $[\![]\!]_\tau$ recurses over the structure of terms and formulas found in Rodin lemmas, and builds terms according to the SMT-LIB syntax. Symbol $\tau$ will be used wherever a term of a formula is possible, and symbol $\varphi$ will be used whenever only a formula is possible. The base cases of the recursion are identifiers (rule 12) and numeric literals (rule 13). Boolean conversion is also dealt with (rule 21).

$$12\ \frac{}{[\![\mathit{name};\ \Gamma]\!]_\tau = \Gamma.\mathsf{nm}(\mathit{name});\ \Gamma} \qquad 13\ \frac{}{[\![\mathit{num};\ \Gamma]\!]_\tau = \mathit{num};\ \Gamma}$$

Next, rules 14 and 15 define, respectively, the translation of unary and binary terms.

$$14\ \frac{o \in \{\neg, -\_, \mathbf{id}, \_^{-1}, \mathrm{dom}, \mathrm{ran}\} \qquad [\![\tau;\ \Gamma]\!]_\tau = \mathsf{t};\ \Gamma' \qquad \mathsf{o} = \Gamma.\mathsf{nm}(o)}{[\![o\ \tau;\ \Gamma]\!]_\tau = (\mathsf{o}\ \mathsf{t});\ \Gamma'}$$

$$15\ \frac{\begin{array}{c} o \in \{\Rightarrow, \Leftrightarrow, =, \neq, <, >, \leqslant, \geqslant, \subset, \subseteq, \in, \lhd, \lhd, \rhd, \rhd, .., \setminus, \times, \_\ ;\ \_, \Leftarrow, \mathsf{div}, \mathsf{mod}, \_ - \_\} \\ [\![\tau_1;\ \Gamma]\!]_\tau = \mathsf{t}_1;\ \Gamma_1 \qquad [\![\tau_2;\ \Gamma_1]\!]_\tau = \mathsf{t}_2;\ \Gamma_2 \qquad \mathsf{o} = \Gamma.\mathsf{nm}(o) \end{array}}{[\![\tau_1\ o\ \tau_2;\ \Gamma]\!]_\tau = (\mathsf{o}\ \mathsf{t}_1\ \mathsf{t}_2);\ \Gamma_2}$$

The rule 16 for translating polyadic, associative and commutative operators is:

$$16\ \frac{o \in \{\wedge, \vee, +, \times, \cup, \cap\} \quad \mathsf{o} = \Gamma.\mathsf{nm}(o) \quad [\![\tau_1;\ \Gamma]\!]_\tau = \mathsf{t}_1;\ \Gamma_1 \ \cdots\ [\![\tau_n;\ \Gamma_{n-1}]\!]_\tau = \mathsf{t}_n;\ \Gamma_n}{[\![\tau_1 o\ \cdots o\ \tau_n;\ \Gamma]\!]_\tau = (\mathsf{o}\ \mathsf{t}_1\ \cdots\ \mathsf{t}_n);\ \Gamma_n}$$

Rule 17 handles quantified formulas. As SMT-LIB requires that quantified variables be sorted, the basic type of the quantified variable $x$ is identified and the corresponding sort $\mathsf{s}$ is obtained from the context. A temporary association is associated to the context that is used to translate the matrix of the quantified formula.

$$17\ \frac{Q \in \{\exists, \forall\} \quad \mathsf{Q} = \Gamma.\mathsf{nm}(Q) \quad \Gamma.\mathsf{nm}(btype(x)) = \mathsf{s} \quad [\![\varphi;\ \Gamma[\mathsf{nm} \oplus \{x \mapsto \mathsf{x}\}]]\!]_\tau = \mathsf{f};\ \Gamma_1}{[\![Qx \bullet \varphi;\ \Gamma]\!]_\tau = (\mathsf{Q}(\mathsf{x}\ \mathsf{s})\ \mathsf{f});\ \Gamma_1[\mathsf{nm} \ominus x]}$$

The following rules specify the translation of some specific set terms: rule 18 deals with sets defined in intention, while rule 19 handles sets defined in extension.

$$18\ \frac{btype(x) = s \quad [\![s, \Gamma]\!]_T = \mathsf{s} \quad fresh(\Gamma, \{\mathsf{x}\}) \quad \Gamma_1 = \Gamma[\mathsf{nm} \oplus \{x \mapsto \mathsf{x}\}] \quad [\![\tau, \Gamma_1]\!]_\tau = \mathsf{t};\ \Gamma_2}{[\![\{x \mid \tau\};\ \Gamma]\!]_\tau = (\mathsf{lambda}\ (\mathsf{x}\ \mathsf{s})\ \mathsf{t});\ \Gamma_2[\mathsf{nm} \ominus \{\}]}$$

$$19\ \frac{btype(\tau_1) = s \quad [\![s, \Gamma]\!]_T = \mathsf{s} \quad fresh(\Gamma, \{\mathsf{x}\}) \quad [\![\tau_1, \Gamma]\!]_\tau = \mathsf{t}_1;\ \Gamma_1 \quad [\![\tau_2, \Gamma_1]\!]_\tau = \mathsf{t}_2;\ \Gamma_2 \quad \cdots \quad [\![\tau_n, \Gamma_{n-1}]\!]_\tau = \mathsf{t}_n;\ \Gamma_n}{[\![\{\tau_1, \cdots, \tau_n\};\ \Gamma]\!]_\tau = (\mathsf{lambda}\ (\mathsf{x}\ \mathsf{s})(\mathsf{or}\ (=\ \mathsf{x}\ \mathsf{t}_1) \cdots (=\ \mathsf{x}\ \mathsf{t}_n)));\ \Gamma_n}$$

Next, rule 20 specifies the translation for $\lambda$-terms. In B, a $\lambda$-term is composed of a guard, say $\varphi$, that specifies the domain of the function, and an expression, say $\tau$, that specifies the result of the function. The translation yields a predicate on pairs, such that the first element of the pair shall satisfy the translation of $\varphi$, and the second element is equal to the translation of $\tau$.

$$20\ \frac{\begin{array}{cc} btype(x) = s_1 \quad btype(\tau) = s_2 & [\![s_1, \Gamma]\!]_T = \mathsf{s}_1 \quad [\![s_2, \Gamma]\!]_T = \mathsf{s}_2 \\ fresh(\Gamma, \{\mathsf{p}, \mathsf{x}_1, \mathsf{x}_2\}) & \Gamma_1 = \Gamma[\mathsf{nm} \oplus \{x \mapsto \mathsf{x}_1\}] \\ [\![\varphi;\ \Gamma_1]\!]_\tau = \mathsf{f};\ \Gamma_2 & [\![\tau;\ \Gamma_2]\!]_\tau = \mathsf{t};\ \Gamma' \end{array}}{[\![\lambda x \bullet \varphi \mid \tau;\ \Gamma]\!]_\tau = (\mathsf{lambda}\ (\mathsf{p}\ (\mathsf{Pair}\ \mathsf{s}_1\ \mathsf{s}_2))(\mathsf{exists}\ (\mathsf{x}_1\ \mathsf{s}_1)(\mathsf{x}_2\ \mathsf{s}_2)(\mathsf{and}\ (=\ \mathsf{p}\ (\mathsf{pair}\ \mathsf{x}_1\ \mathsf{x}_2))\ (\mathsf{f}\ \mathsf{x}_1)\ (=\ \mathsf{t}\ \mathsf{x}_2)))), \Gamma'}$$

The B logic includes a conversion operator, called **bool**, from formulas to terms that is useless in SMT-LIB, since the latter has no (longer) specific syntactic entity for formulas. Rule 21 defines this translation.

$$21\ \frac{}{[\![\mathbf{bool}(\tau);\ \Gamma]\!]_\tau = [\![\tau;\ \Gamma]\!]_\tau}$$

### *5.6. Mixed Operators*

Finally, translation of mixed-sort operators require additional definitions. The following operators are considered for translation: **min** and **max** that yield respectively the smallest and highest value of a non-empty set of integers, the set predicate operator **finite** and the cardinality operator **card**.

In order to define the translation of the first two operators, the following two macros are introduced:

$$
\begin{array}{rcl}
\mathsf{ismin} & \equiv & (\mathsf{lambda}\ \ (\mathsf{m\ Int})\ (\mathsf{t\ (Int\ Bool)}) \\
 & & \quad (\mathsf{and(in\ m\ t)(forall\ (x\ Int)\ (implies\ (in\ x\ t)(<=\ m\ x)))))} \\
\mathsf{ismax} & \equiv & (\mathsf{lambda}\ \ (\mathsf{m\ Int})\ (\mathsf{t\ (Int\ Bool)}) \\
 & & \quad (\mathsf{and(in\ m\ t)(forall\ (x\ Int)\ (implies\ (in\ x\ t)(<=\ x\ m)))))}
\end{array}
$$

Thus, (ismin m t) expands to a formula stating that $m$ is the smallest value in the set t. This formula is the conjunction of the properties of the **min** operator [1]. The translation of the operators **min** and **max** is specified by rules 22 and 23. Each application of these rules add a fresh integer constant and an assumption to the context.

$$
22\ \frac{[\![\tau, \Gamma]\!]_\tau = \mathsf{t};\ \Gamma_1 \qquad fresh(\Gamma_1, \{\mathsf{m}\}) \qquad \Gamma_2 = \Gamma_1[\mathsf{funs} \oplus \{\mathsf{m} \mapsto \mathsf{Int}\} \mid \mathsf{assumptions} \oplus \{(\mathsf{ismin\ m\ t})\}]}{[\![\mathbf{min}(\tau), \Gamma]\!]_\tau = \mathsf{m};\ \Gamma_2}
$$

$$
23\ \frac{[\![\tau, \Gamma]\!]_\tau = \mathsf{t};\ \Gamma_1 \qquad fresh(\Gamma_1, \{\mathsf{m}\}) \qquad \Gamma_2 = \Gamma_1[\mathsf{funs} \oplus \{\mathsf{m} \mapsto \mathsf{Int}\} \mid \mathsf{assumptions} \oplus \{(\mathsf{ismax\ m\ t})\}]}{[\![\mathbf{max}(\tau), \Gamma]\!]_\tau = \mathsf{m};\ \Gamma_2}
$$

The operator finite is a predicate that is true of finite sets. The following macro relates a proposition p, a unary predicate t, a labelling function f and a constant k. Informally, p is an atomic proposition stating that the argument set is finite, k is an upper bound on the cardinality of the set, and f maps injectively elements of the set with a positive integer smaller than k.

$$
\begin{array}{rcl}
\mathsf{finite} & \equiv & (\mathsf{lambda}\ \ (\mathsf{p\ Bool})\ (\mathsf{t\ ('s\ Bool)})\ (\mathsf{f\ ('s\ Int)})\ (\mathsf{k\ Int})) \\
 & & \quad (\mathsf{iff\ p\ (and\ \ (forall\ (x\ s)(implies\ (in\ x\ t)(in\ (f\ x)(range\ 1\ k))))} \\
 & & \qquad (\mathsf{forall\ (x\ s)(y\ s)(implies\ \ (and\ \ (in\ x\ t)} \\
 & & \qquad\qquad (\mathsf{in\ y\ t)} \\
 & & \qquad\qquad (\mathsf{not\ (equal\ x\ y)))} \\
 & & \qquad (\mathsf{not\ (equal\ (f\ x)(f\ y)))))))}
\end{array}
$$

Rule 24 specifies the translation of the predicate application **finite**($\tau$): the context is enriched with an atomic proposition p, a constant k and a function f, of domain s, the sort for the basic type of the elements of $\tau$. The context is also augmented with an assumption obtained by an expansion of macro finite.

$$
24\ \frac{[\![\tau, \Gamma]\!]_\tau = \mathsf{t};\ \Gamma_1 \quad btype(\tau) = \mathbb{P}(s) \quad \Gamma.\mathsf{nm}(s) = \mathsf{s} \quad fresh(\Gamma_1, \{\mathsf{p}, \mathsf{k}, \mathsf{f}\}) \quad \Gamma_2 = \Gamma_1[\mathsf{preds} \oplus \{\mathsf{p} \mapsto ()\} \mid \mathsf{funs} \oplus \{\mathsf{k} \mapsto \mathsf{Int}, \mathsf{f} \mapsto (\mathsf{s\ Int})\} \mid \mathsf{assumptions} \oplus (\mathsf{finite\ p\ t\ f\ k})]}{[\![\mathbf{finite}(\tau), \Gamma]\!]_\tau = \mathsf{p};\ \Gamma_2}
$$

The operator **card** is a function from sets to integers. The following macro relates a unary predicate t, a labelling

function f and a constant k. Informally, t is the characteristic function of a set, k is the cardinality of the set, and f maps bijectively elements of the set with the range [1; k].

$$
\begin{array}{lll}
\mathsf{card} & \equiv & (\mathsf{lambda}\ \ (\mathsf{t}\ (\mathsf{'s\ Bool}))\ (\mathsf{f}\ (\mathsf{'s\ Int}))\ (\mathsf{k\ Int})) \\
 & & (\mathsf{forall}\ (\mathsf{x\ s})(\mathsf{implies}\ (\mathsf{in\ x\ t})(\mathsf{in}\ (\mathsf{f\ x})(\mathsf{range\ 1\ k})))) \\
 & & (\mathsf{forall}\ (\mathsf{x\ s})(\mathsf{y\ s})(\mathsf{implies}\ \ (\mathsf{and}\ (\mathsf{in\ x\ t})\ (\mathsf{in\ y\ t})) \\
 & & \qquad\qquad (\mathsf{iff}\ (\mathsf{equal\ x\ y})\ (\mathsf{equal}\ (\mathsf{f\ x})(\mathsf{f\ y}))))))
\end{array}
$$

Rule 25 specifies the translation of the application of operator **card** to a set $\tau$; the context is enriched with a constant k and a function f, and an assumption that relates both new symbols to set $\tau$ using the macro card.

$$
25\ \frac{\begin{array}{c}[\![\tau, \Gamma]\!]_\tau = \mathsf{t};\ \Gamma_1 \qquad \mathit{btype}(\tau) = \mathbb{P}(s) \qquad \Gamma.\mathsf{nm}(s) = \mathsf{s} \qquad \mathit{fresh}(\Gamma_1, \{\mathsf{p}, \mathsf{k}, \mathsf{f}\}) \\ \Gamma_2 = \Gamma_1[\mathsf{funs} \oplus \{\mathsf{k} \mapsto \mathsf{Int}, \mathsf{f} \mapsto (\mathsf{s\ Int})\} \mid \\ \qquad\qquad \mathsf{assumptions} \oplus (\mathsf{card\ t\ f\ k})]\end{array}}{[\![\mathbf{card}(\tau), \Gamma]\!]_\tau = \mathsf{k};\ \Gamma_2}
$$

## 6. Implementation

The Rodin platform [11] is a development environment for Event-B. It is based on Eclipse [31] and can be extended by developing new plug-ins in the Java programming language. The translation described in this paper was used as a specification for a new plug-in dedicated to the automatic verification of proof obligations using SMT-solvers. The plug-in communicates with the SMT-solvers using files and operating system commands. The configuration of the plug-in includes a choice of SMT-solvers: Alt-Ergo [6], cvc3 [5], veriT [8] and Z3 [13] at this time (see figure 8). It is now available to the formal methods community as an exploratory package through Rodin's official source code repository [3].

Currently, the verification with SMT-solver has to be activated by the user by clicking a button (see figure 9, left part). Whenever the verification is successful (see figure 9, right part), the status of the proof obligation is updated and the user may move to the next proof obligation.

## 7. Experimental Results

To evaluate the application of SMT-solvers in the context of the approach described in this paper, we considered all the proof obligations generated by the developments made publicly available by the European project Deploy at its Web site (http://www.deploy-project.eu/). F Since the prover is a satisfiability modulo theory (SMT) solver, to check the validity of a lemma, the negation of the lemma is given to the solver. If this negation is found unsatisfiable, then the original formula is valid.

[3] `http://rodin-b-sharp.svn.sourceforge.net/viewvc/rodin-b-sharp/trunk/_exploratory/fages/`

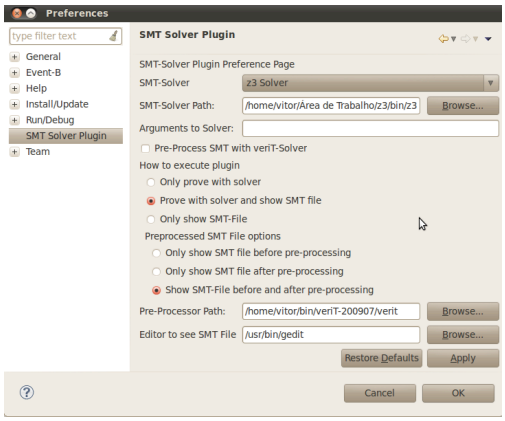


Figure 8: Screen capture of the plugin preference window. Options include the choice of a SMT-solver and visualization of the result translation for debugging purposes.

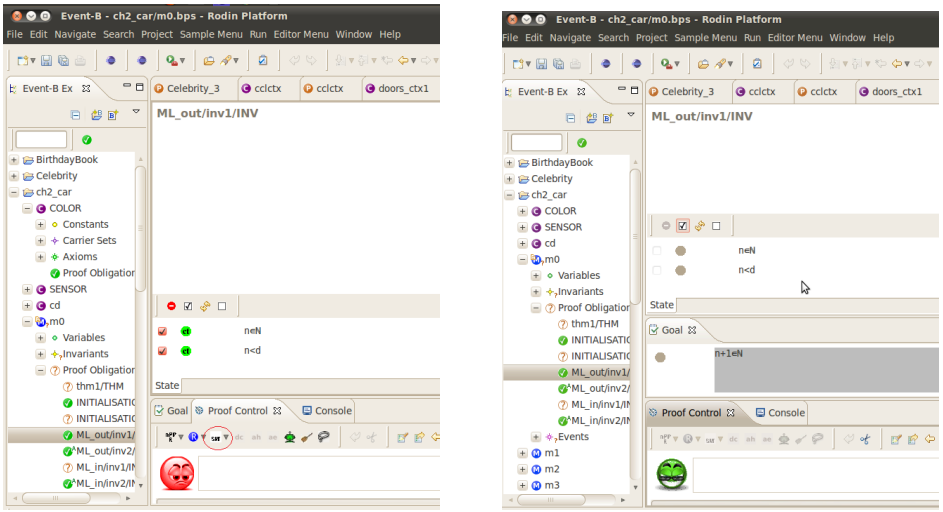


Figure 9: Screen captures of the proof. On the left, before the proof, the button is active and the status is “unproved”. On the right, after a successful proof, the button has been disactivated and the status is “proved”.

The selected benchmarks were translated automatically by a Rodin plug-in described in Section 6 and then processed with the SMT-solver veriT [8], that integrates the necessary pre-processing steps (e.g. macro expansion, beta reduction) to handle the constructs introduced in these translation rules. Although this tool chain is still experimental and thus not yet integrated to the official releases of the Rodin platform, the results we obtained are rather positive, as shown in the table of figure 1. In our experiment, we applied the Predicate Prover to all of the 2359 proof obligations, and it terminated successfully in 54% of the cases. When applying the approach described in this paper, and using veriT as the SMT-solver, the ratio of discharged proof obligations jumps to 63%. It is important to note that all the runs of the SMT-solver take a negligible amount of time. The remaining proof obligations will be communicated to the community of researchers on SMT-solving techniques in the hope that use them as a direction for future extensions and improvements.

## 8. Conclusions

This paper addresses the verification of proof obligations generated in formal systems and software developments in B and Event-B using SMT-solvers. A fragment of the logic of such proof obligations was chosen: essentially it combines basic sets and relations, fragments of integer arithmetics and Booleans. The scope therefore compares to that of existing tools such as the Predicate Prover.

The use of pre-processing constructs simplifies the specification and implementation of the translation from set theory to predicate logic: e.g. macros are associated to the main operators of set theory. A set of rules specifies the translation of proof obligations to the SMT-LIB format extended with macro-definitions and lambda expressions. The second stage of translation, applying classic pre-processing techniques such as macro-expansion, produces fully compliant SMT-LIB files that can be handled with existing SMT-solvers.

A significant set of proof obligations was used to assess the usefulness of the approach. The translation system was applied to the proof obligations and the resulting SMT-LIB files were verified with an existing SMT-solver. This experiment validates the approach: existing SMT-solvers may be employed to discharge automatically and quickly a number of proof obligations in formal software developments such as Event-B. They can thus be used to complement the panoply of provers currently available in integrated development environments for B and Event-B. Indeed, the translation presented in this paper was used to implement a verification plug-in for the Rodin platform targeting several SMT-solvers.

*Future Work.* The translation presented herein may be employed in other development platforms, such as Atelier B. Also some capabilities of SMT-solvers can be used to improve the workflow in such development platforms. First, one can define classes of proof obligations where the solver is complete and where more accurate results can be reported, using theoretical results such as those reported in [20]. In those cases where the solver is complete, counter-models can be reported to the user, when a proof obligation cannot be verified. Also, SMT-solvers may identify the subset of hypothesis that was actually useful to verify a proof obligation. This information can be used by the

environment platform to reduce the number of generated proof obligations when the user modifies a definition. Finally, as some SMT-solvers include incremental assertion and retraction of sets of hypothesis, a tighter integration with the proof obligation generators could be designed to apply this feature to incrementally add or remove hypothesis that correspond to a definition context or control flow conditions, which would likely lead to faster verification times.

Also, a thorough experimental evaluation of the performances of SMT-solvers when dealing with the proof obligations generated in formal development in B and Event-B is now possible. The results of such evaluation could be used to establish a research agenda to drive further developments in SMT-solving techniques.

Finally, other translations can be defined to consider other classes of proof obligations. For instance, as suggested in [12, 22], sets can be mapped to arrays of Booleans, for which efficient SMT-solving techniques are available.

*Acknowledgements.*. The author thanks Systerel's Laurent Voisin, Carine Pascal, Yoann Fages and Yoann Guyot for providing initial examples and assistance on plug-in development for Rodin. The initial implementation of macro expansion, beta reduction and lifting of equalities to formulas in veriT is due to Pascal Fontaine. He also proposed to use them as a means to handle basic set connectors. Vítor Alcântara de Almeida is the author of the Rodin plug-in implementing the translation.

## References

[1] Abrial, J.-R., 1996. The B-book: assigning programs to meanings. Cambridge University Press.
URL http://portal.acm.org/citation.cfm?id=236705

[2] Abrial, J.-R., 2010. Modeling in Event-B: System and Software Engineering. Cambridge University Press.

[3] B-Core Ltd, 2002. The B-Toolkit. \url{http://www.b-core.com/btoolkit.html}.

[4] Barrett, C., Stump, A., Tinelli, C., 2010. The SMT-LIB Standard Version 2.0.
URL http://goedel.cs.uiowa.edu/smtlib/papers/smt-lib-reference-v2.0-r10.03.30.pdf

[5] Barrett, C., Tinelli, C., July 2007. CVC3. In: Damm, W., Hermanns, H. (Eds.), Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07). Vol. 4590 of Lecture Notes in Computer Science. Springer-Verlag, pp. 298–302.

[6] Bobot, F., Conchon, S., Contejean, E., Lescuyer, S., 2008. Implementing Polymorphism in SMT solvers. In: Barrett, C., de Moura, L. (Eds.), SMT 2008: 6th International Workshop on Satisfiability Modulo Theories.

[7] Bofill, M., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A., 2008. The Barcelogic SMT Solver. In: Gupta, A., Malik, S. (Eds.), Proceedings of the 20th international conference on Computer Aided Verification. Vol. 5123 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 294–298.
URL http://www.springerlink.com/index/10.1007/978-3-540-70545-1

[8] Bouton, T., de Oliveira, D. C. B., Déharbe, D., Fontaine, P., 2009. veriT: An Open, Trustable and Efficient SMT-Solver. In: Automated Deduction - CADE-22. Vol. 5663 of Lecture Notes in Computer Science. Springer Verlag, pp. 151–156.
URL http://www.springerlink.com/content/f33m4615152325x3

[9] Bruun, H., Damm, F., Dawes, J., Hansen, B., Larsen, P., Parkin, G., Plat, N., Toetenel, H., 1998. A formal definition of {VDM-SL}. Tech. Rep. Technical Report 1998/9, University of Leicester.

[10] ClearSy, 2009. Atelier B User Manual Version 4.0. Clearsy System Engineering.
URL http://www.atelierb.eu

[11] Coleman, J., Jones, C., Oliver, I., Romanovsky, A., E.Troubitsyna, 2005. {RODIN} (Rigorous open Development Environment for Complex Systems). In: Fifth European Dependable Computing Conference: EDCC-5 supplementary volume. pp. 23–26.

[12] Couchot, J.-F., Déharbe, D., Giorgetti, A., Ranise, S., 2003. Scalable Automated Proving and Debugging of Set-Based Specifications. Journal of the Brazilian Computer Society 9, 17–36.

[13] de Moura, L., Bjø rner, N., 2008. Z3: An Efficient SMT Solver. In: Ramakrishnan, C. R., Rehof, J. (Eds.), Proc. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Vol. 4963 of Lecture Notes in Computer Science. Springer, pp. 337–340.

[14] de Moura, L., Bjø rner, N., 2009. Satisfiability Modulo Theories: An Appetizer. In: Oliveira, M. V. M., Woodcock, J. (Eds.), Formal Methods: Foundations and Applications. Vol. 5902 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 23–36.
URL http://www.springerlink.com/index/10.1007/978-3-642-10452-7

[15] de Moura, L., Dutertre, B., Shankar, N., 2007. A Tutorial on Satisfiability Modulo Theories. In: Damm, W., Hermanns, H. (Eds.), Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings. Vol. 4590 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 20–36.
URL http://www.springerlink.com/content/11r20jq883677834

[16] Déharbe, D., 2010. Automatic Verification for a Class of Proof Obligations with SMT-Solvers. In: Frappier, M., Uwe, G., Sarfraz, K., Laleau, R., Reeves, S. (Eds.), Abstract State Machines, Alloy, B and Z, Second International Conference, ABZ 2010, Orford, QC, Canada, February 22-25, 2010. Proceedings. No. 1 in Lecture Notes in Computer Science. Springer Verlag, pp. 217–230.
URL http://dx.doi.org/10.1007/978-3-642-11811-1_17

[17] Déharbe, D., Fontaine, P., Ranise, S., Ringeissen, C., 2006. Decision Procedures for the Formal Analysis of Software. In: Theoretical Aspects of Computing—ICTAC 2006. Lecture Notes in Computer Science. Springer Verlag, pp. 366–370.
URL http://dx.doi.org/10.1007/11921240_26

[18] Dutertre, B., de Moura, L., August 2006. The Yices SMT solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf.

[19] El Ghazi, A. A., Taghdiri, M., 2010. Analyzing Alloy Constraints using an SMT Solver: A Case Study. In: Dutertre, B., Saïdi, H. (Eds.), AFM10 (Automated Formal Methods).

[20] Fontaine, P., 2009. Combinations of theories for decidable fragments of first-order logic. In: 7th International Symposium on Frontiers of Combining Systems (FroCoS'09). Vol. 5749 of Lecture Notes in Computer Science. Springer, pp. 263–278.

[21] Hurlin, C., Chaieb, A., Fontaine, P., Merz, S., Weber, T., July 2007. Practical Proof Reconstruction for First-order Logic and Set-Theoretical Constructions. In: Dixon, L., Johansson, M. (Eds.), Proceedings of the Isabelle Workshop 2007. pp. 2–13.

[22] Kröning, D., Rümmer, P., Weissenbacher, G., 2009. A Proposal for a Theory of Finite Sets, Lists, and Maps for the {SMT-LIB} Standard. In: Informal proceedings, 7th International Workshop on Satisfiability Modulo Theories at CADE 22.

[23] Leuschel, M., Butler, M., 2008. ProB: An Automated Analysis Toolset for the B Method. Journal Software Tools for Technology Transfer 10 (2), 185–.

[24] Nelson, G., Oppen, D., 1979. Simplification by Cooperating Decision Procedures. ACM Transactions on Programming Languages and Systems 1 (2), 245–257.

[25] Nelson, G., Oppen, D., 1980. Fast Decision Procedures Based on Congruence Closure. Journal of the ACM 27 (2), 356–364.

[26] Nieuwenhuis, R., Oliveras, A., 2005. {DPLL(T)} with Exhaustive Theory Propagation and its Application to Difference Logic. In: Etessami, K., Rajamani, S. (Eds.), Proc. 17th Int'l Conf. on Computer Aided Verification (CAV). Vol. 3576 of Lecture Notes in Computer Science. Springer, pp. 321–334.

[27] Pierce, B. C., 2002. Types and Programming Languages. MIT Press.

[28] Plagge, D., Leuschel, M., Lopatkin, I., Romanovsky, A., 2009. SAL, Kodkod, and BDDs for Validation of B Models. Lessons and Outlook. In: Saïdi, H., Shankar, N. (Eds.), Automated Formal Methods (AFM09). pp. 16–22.

[29] Ranise, S., Tinelli, C., August 2006. The SMT-LIB Standard : Version 1.2.

[30] Spivey, J., 1992. The Z Notation: a Reference Manual, 2nd Edition. Prentice-Hall International Series in Computer Science. Prentice Hall.

[31] The Eclipse Foundation, 2009. Eclipse SDK.
URL http://www.eclipse.org

| Project | Proof obligations | | |
|---|---|---|---|
| | Total | PP | PP+SMT |
| AccessControl | 11 | 11 | 11 |
| Celebrity | 46 | 25 | 25 |
| ch2_car | 253 | 141 | 167 |
| ch3_pattern | 126 | 68 | 104 |
| ch4_file_1 | 52 | 19 | 26 |
| ch4_file_2 | 41 | 2 | 15 |
| ch6_brp | 149 | 103 | 103 |
| ch7_conc | 248 | 109 | 127 |
| ch8_circ_arbiter | 153 | 110 | 110 |
| ch8_circ_light | 89 | 89 | 89 |
| ch8_circ_pulser | 94 | 63 | 69 |
| ch8_circ_road | 37 | 22 | 37 |
| ch910_ring | 24 | 8 | 8 |
| ch911_tree | 81 | 37 | 37 |
| ch912_mobile | 153 | 81 | 81 |
| ch913_ieee | 67 | 52 | 52 |
| ch915_bin | 37 | 0 | 10 |
| ch915_inv | 32 | 0 | 12 |
| ch915_mini | 25 | 0 | 7 |
| ch915_search | 17 | 0 | 5 |
| ch915_sqrt | 17 | 0 | 6 |
| ch916_doors | 103 | 100 | 100 |
| ch917_train | 133 | 92 | 92 |
| Closure | 4 | 0 | 0 |
| DecertExamples | 39 | 12 | 25 |
| Doors | 108 | 105 | 105 |
| Galois | 1 | 1 | 1 |
| ParkingLot | 10 | 2 | 9 |
| RingLeader | 33 | 7 | 7 |
| search | 31 | 0 | 8 |
| search_backward | 31 | 0 | 8 |
| search_proved | 31 | 0 | 8 |
| society | 7 | 5 | 5 |
| society_sol | 9 | 5 | 5 |
| Total | 2359 | 1273 | 1494 |

Table 1: Experimental results: verifying RODIN lemmas with veriT. Column 1 contains the name of the Event-B project. Columns 2 displays the number of proof obligations in that project. Column 3 shows how many POs were discharged automatically with the Predicate Prover, a tool included in the Roding platform. Column 4 contains the number of POs proved when applying the approach described in the paper.