Automatic Verification for a Class of Proof Obligations with SMT-Solvers

David Déharbe

UFRN / DIMAp / ForAll
Universidade Federal do Rio Grande do Norte
Departamento de Informática e Matemática Aplicada
Formal Methods and Languages Research Laboratory
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 therefore 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 generated in the Rodin platform in a language that can be addressed by state-of-the-art SMT solvers. The work presented in the paper handles proof obligations with Booleans, integer arithmetics and basic sets.

1 Introduction

Formal software development using frameworks such as B, Event-B and Z produces large quantities of proof obligations (POs), typically expressed in a first-order language including arithmetic and set-theoretic constructs. In the case of B and Event-B, platforms such as Atelier-B [1] and Rodin [2] 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. 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 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. Such interactions are a time-consuming activity and have a direct impact on the cost of software development. Progress in automatic theorem proving techniques for formal software development framework is therefore key to increase the application and dissemination of formal methods. This may be achieved with (1) more cost-effective

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 is successful to address many software-related problems. SMT-solvers 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 [3], and other optimizations, such as theory propagation [4]. Some solvers are also able to build a proof of their result, which is often as useful as the result for the users. The international SMT-LIB initiative provides a common input format [5] 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. Indeed, 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 [6]. This proposition considers many specification constructs such as sets, sequences and maps that are not yet fully handled by SMT-solving techniques.

The goal of the work presented in this paper is to present an approach to apply *existing* SMT-solvers to POs, restricted to a subset of the specification constructs of B and Event-B: booleans, linear integer arithmetics and basic sets, plus operators mapping sets to numbers such as cardinality. This experiment is based on a translation of the POs generated in the Rodin platform to the SMT-LIB language and, if successful, lays the basis for the implementation of a verification plug-in based on SMT-solvers for this platform. The scope and languages involved in this translation are presented in Section 2. The general principles of this translation are described in Section 3 and the detailed presentation of its formalization in Section 4. This translation was applied to a set of benchmarks supplied by an industrial partner and the result of their verification with an existing SMT-solver are presented in Section 5. The conclusions of this work are presented in Section 6.

2 The Source and Target Languages

2.1 Proof Obligations in Rodin

The Rodin platform [2] stores POs as XML files. The abstract syntax of the format is described hereafter. A PO file is structured in four sections:

$$\begin{array}{lcccc} L := & T^+ & E & H^* & G \\ & \textit{(theories)} & \textit{(typing environment)} & \textit{(hypothesis)} & \textit{(goal)} \end{array}$$

The theories are pre-defined names corresponding to different fragments of the specification logic. This work only considers theories that correspond to booleans,

integers and simple sets (i.e. no set of sets is allowed), but it could be straightforwardly extended to include real numbers. Another possible extension would be basic binary relations (i.e. binary relations over basic sets).

Since this work only considers booleans, integers and simple sets, the following grammar defines the typing environment:

$$\begin{array}{lll} E := & \textit{(typing environment)} & \\ & V^* & \text{variables} \end{array} \qquad \begin{array}{lll} S := & & \textit{(sort)} \\ & C & \text{carrier} \\ & \mid \mathbb{P}(C) & \text{set} \end{array}$$

$$\begin{array}{lll} V := & \textit{(variable)} & \\ & \textit{name} & \text{name} \\ & S & \text{sort} \end{array} \qquad \begin{array}{lll} C := & & \textit{(carrier set)} \\ & \mathbb{Z} & \text{integers} \\ & \mid \textit{name} & \text{user-defined} \end{array}$$

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 ϕ and τ stand respectively for formulas and terms. POs are well-typed and no type checking or type inference is needed.

$$\begin{array}{c} \phi := \neg\phi \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi \mid \phi \wedge \cdots \wedge \phi \mid \phi \vee \cdots \vee \phi \mid \exists x \bullet \phi \mid \forall x \bullet \phi \mid \mathbf{bool}(\phi) \mid \\ \tau = \tau \mid \tau \neq \tau \mid \tau < \tau \mid \tau > \tau \mid \tau \leq \tau \mid \tau \geq \tau \mid \tau \subseteq \tau \mid \tau \subset \tau \mid \tau \in \tau \end{array}$$

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

$$\begin{array}{l} \tau := \textit{name} \mid \mathbf{TRUE} \mid \mathbf{FALSE} \mid \textit{num} \mid \\ \quad -\tau \mid \tau - \tau \mid \tau \,\mathbf{div}\, \tau \mid \tau \,\mathbf{mod}\, \tau \mid \tau + \cdots + \tau \mid \tau \times \cdots \times \tau \mid \\ \quad \tau..\tau \mid \{\tau, \cdots, \tau\} \mid \{x \mid \phi\} \mid \tau \cup \cdots \cup \tau \mid \tau \cap \cdots \cap \tau \mid \tau \setminus \tau \end{array}$$

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

$$\begin{array}{l} \phi := \mathbf{finite}(\tau) \\ \tau := \mathbf{min}(\tau) \mid \mathbf{max}(\tau) \mid \mathbf{card}(\tau) \end{array}$$

2.2 The SMT-LIB Format

The SMT-LIB format [5] 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. Although there is an on-going discussion to extend the format, this paper considers version 1.2, which is the most recent official version.

A SMT benchmark is a sequence of declarations of different entities: logic, sorts, functions, predicates, assumptions, and goal. 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. Related to our work is the recent proposal [6] to include a logic for POs from the Vienna Development Method [7] in the SMT-LIB. This theory includes finite sets, lists and maps; it is currently not supported by existing SMT-solvers.

In general, a SMT benchmark also contains declarations for additional sorts, functions (including constants) and predicates (including atomic propositions). A sort is defined by its name. A function declaration is composed of the function symbol, and the list with the sorts of the parameters followed by the sort of the result. A predicate declaration is composed of the predicate symbol and the list of the sorts of its arguments. The semantics of such symbols may be defined axiomatically with first-order formulas as assumptions.

Finally, the assumptions and the goal are first-order multi-sorted logic formulas expressed using both the symbols of the declared logic and the additional symbols declared in the benchmark. In the general case, formulas may contain arbitrary combinations of quantifiers; the language also contains if-then-else constructs for both terms and formulas.

3 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 logic of Event-B and B contains terms of sort Boolean which is reflected in Rodin POs. The translation maps such terms to formulas, as this approach should take better advantage of the efficient handling of complex Boolean structures in SMT-solvers.

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 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.

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 [8, 6]). 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 [9] 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.

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 [10], 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 4):

```
union ≡ (lambda (p ('s Bool))(q ('s Bool))(lambda (?x 's) (or (p ?x) (q ?x))))
```

Here, union is declared as a macro with two arguments p and q. 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 hand side of the expression: it denotes a unary predicate with formal parameter ?x of sort 's, defined as disjunction of the appliction of p and q to ?x. veriT implements a type inference mechanism to handle macros with sort variables similar to that for "let polymorphism" [11]. The remaining macros are:

```
    empty set:      empty ≡ (lambda (?x 's) false)
    intersection:     inter ≡ (lambda (p ('s Bool))(q ('s Bool))
                              (lambda (?x 's) (and (p ?x) (q ?x))))
    difference:  setminus ≡ (lambda (p ('s Bool))(q ('s Bool))
                              (lambda (?x 's) (and (p ?x) (not (q ?x)))))
    membership:        in ≡ (lambda (x 's) (p ('s Bool))(p x))
    inclusion:   subseteq ≡ (lambda (p ('s Bool))(q ('s Bool))
                              (lambda (?x 's) (implies (p ?x) (q ?x))))
strict inclusion:  subset ≡ (lambda (p ('s Bool))(q ('s Bool))
                              (and (subseteq p q) (not (= p q))))
    range:          range ≡ (lambda (lo Int)(hi Int)
                              (lambda (?x Int) (and (<= lo ?x)(<= ?x hi))))
```

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:

$$(\mathsf{lambda}\ (?\mathsf{x}\ s)\ (\mathsf{or}\ (\mathsf{p_S}\ ?\mathsf{x})\ \mathsf{false})) = (\mathsf{lambda}\ (?\mathsf{x}\ s)\ (\mathsf{p_S}\ ?\mathsf{x})),$$

where $\mathsf{p_S}$ 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:

$$(=\ \mathsf{p}\ \mathsf{q}) \rightsquigarrow (\mathsf{forall}\ (?\mathsf{x}\ \mathsf{t})\ (\mathsf{iff}\ (\mathsf{p}\ ?\mathsf{x})\ (\mathsf{q}\ ?\mathsf{x})) \quad \text{if} \vdash_t \mathsf{p} : (\mathsf{t}\ \mathsf{Bool}) \text{ and} \vdash_t \mathsf{q} : (\mathsf{t}\ \mathsf{Bool}),$$

where $\vdash_t \mathsf{e} : \mathsf{s}$ is the typing judgement that s is the sort of expression e. Applying this rule followed by beta reduction to the example yields the first-order logic tautology: $(\mathsf{forall}\ (?\mathsf{x}\ \mathsf{s})\ (\mathsf{iff}\ (\mathsf{or}\ (\mathsf{p}\ ?\mathsf{x})\ \mathsf{false})(\mathsf{p}\ ?\mathsf{x})))$.

4 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.

4.1 Preliminary Definitions and Notations

The rules propagate an evaluation context Γ that gathers incrementally the contents of the different sections that compose the SMT-LIB format. They are: sorts, a set of SMT-LIB identifiers; preds (predicate symbols) and funs (function symbols), both maps SMT-LIB identifiers to a list of sorts; $\mathsf{assumptions}$, a set of formulas in SMT-LIB syntax; $\mathsf{formula}$, the goal, a formula in SMT-LIB syntax. In addition, the context maintains a mapping nm from Rodin symbols to SMT-LIB identifiers. Γ_I denotes the initial context and is such that:

$$\begin{aligned}
\Gamma_I = \{\, \mathsf{nm} \ &= \{\, \mathbb{Z} \mapsto \mathsf{Int}, \mathbf{BOOL} \mapsto \mathsf{Bool}, \\
&\quad \mathbf{TRUE} \mapsto \mathsf{true}, \mathbf{FALSE} \mapsto \mathsf{false} \\
&\quad \wedge \mapsto \mathsf{and}, \vee \mapsto \mathsf{or}, \Rightarrow \mapsto \mathsf{implies}, \Leftrightarrow \mapsto \mathsf{iff}, \\
&\quad \exists \mapsto \mathsf{exists}, \forall \mapsto \mathsf{forall}, \\
&\quad = \mapsto =, < \mapsto <, \leq \mapsto <=, > \mapsto >, \geq \mapsto >=, \\
&\quad \subset \mapsto \mathsf{subset}, \subseteq \mapsto \mathsf{subseteq}, \in \mapsto \mathsf{in} \\
&\quad + \mapsto +, \times \mapsto *, \mathbf{div} \mapsto /, \mathbf{mod} \mapsto \%, - \mapsto - \\
&\quad \cup \mapsto \mathsf{union}, \cap \mapsto \mathsf{inter}, \\
&\quad \setminus \mapsto \mathsf{setminus}, \emptyset \mapsto \mathsf{empty}\} \\
\mathsf{sorts} &= \{\mathsf{Int}, \mathsf{Bool}\}, \\
\mathsf{funs} \ &= \{\}, \mathsf{preds} = \{\}, \mathsf{assumptions} = \{\}, \mathsf{formula} = \{\} \quad \}
\end{aligned}$$

In the following, $\Gamma.\mathsf{sec}$ denotes the content of Section sec of Γ; when sec is a map, $\Gamma[\mathsf{sec} \oplus s]$ denotes update of sec with the map s, otherwise it denotes inclusion of s to the set sec; $\Gamma[\mathsf{sec} \ominus s]$ sec denotes removal of s from 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* denotes a predicate that tests if each element of a given set of SMT-LIB identifiers is fresh with respect to the current context.

4.2 Translation Rule for a PO

Rule 1 specifies how the evaluation of a PO L composed of the sections $T\ E\ H^*G$ results in a context Γ. The operators $[\![]\!]_L, [\![]\!]_E, [\![]\!]_H, [\![]\!]_G$ are responsible for translating a full PO, the typing environment, the hypothesis and the goal respectively. The resulting context Γ represents the components of SMT-LIB format for the Rodin PO L.

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

4.3 Translation Rules for the Typing Environment

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

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

The declarations of a Rodin lemma, grouped in the typing environment, are pairs n, t, where n is the name of the declared entity, and t is its basic type. When n and t are identical, then a new basic type is introduced in B, which is mapped to a new sort in SMT-LIB (Rule 4). When n and t differ, t may be a basic type, and then n is a value of type t, and a corresponding constant function n is added in the SMT-LIB (Rule 5). Another possibility is that the type be the powerset of some basic type t, then n is a set, and a corresponding unary predicate n is added in the SMT-LIB (Rule 6).

$$4\ \frac{n = t \qquad \Gamma' = \Gamma[\mathsf{nm} \oplus \{t \mapsto \mathsf{t}\} \mid \mathsf{sorts} \oplus \{\mathsf{t}\}]}{[\![n\ t; \Gamma]\!]_V = \Gamma'}\ \text{(basic type)}$$

$$5\ \frac{n \neq t \qquad \Gamma.\mathsf{nm}(t) = \mathsf{t} \qquad \mathsf{t} \in \Gamma.\mathsf{sort} \qquad \Gamma' = \Gamma[\mathsf{nm} \oplus \{n \mapsto \mathsf{n}\} \mid \mathsf{funs} \oplus \{\mathsf{n} \mapsto (\mathsf{t})\}]}{[\![n\ t; \Gamma]\!]_V = \Gamma'}\ \text{(set element)}$$

$$6\ \frac{n \neq t \qquad \Gamma(t) = \mathsf{t} \qquad \mathsf{t} \in \Gamma.\mathsf{sort} \qquad \Gamma' = \Gamma[\mathsf{nm} \oplus \{n \mapsto \mathsf{n}\} \mid \mathsf{preds} \oplus \{\mathsf{n} \mapsto (\mathsf{t})\}]}{[\![n\ \mathbb{P}(t); \Gamma]\!]_V = \Gamma'}\ \text{(set)}$$

4.4 Translation Rules for Hypothesis and Goal

Rules 7 and 8 specify that the hypothesis section is translated by a sequential traversal, applying the translation operator $[\![]\!]_\phi$ to each hypothesis, and yielding a formula f and a context Γ. The former is added to the latter as a result of the translation.

$$7\frac{}{[\![;\Gamma]\!]_H = \Gamma} \qquad 8\frac{[\![\phi^*;\Gamma]\!]_H = \Gamma_0 \quad [\![\phi;\Gamma_0]\!]_\phi = \mathsf{f};\Gamma_1}{[\![\phi^*\phi;\Gamma]\!]_H = \Gamma_1[\mathsf{assumptions} \oplus \{\mathsf{f}\}]}$$

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

$$9\frac{[\![G;\Gamma]\!]_\phi = \mathsf{g};\Gamma_1}{[\![G;\Gamma]\!]_G = \Gamma_1[\mathsf{formula} \oplus \{\mathsf{g}\}]}$$

4.5 Translation Rules for Formulas

The translation operator $[\![]\!]_\phi$ recurses over the structure of the formulas, up to the level of atoms. There are two classes of atoms: boolean constants, and applications of relational operators. The definition of the latter uses the term translation operator $[\![]\!]_\tau$ (defined in Section 4.6) to process the arguments. Their translation is specified by rules 10 and 11:

$$10\frac{\begin{array}{c}o \in \{=,<,>,\leq,\geq,\subset,\subseteq,\in\} \\ [\![\tau_1;\Gamma]\!]_\tau = \mathsf{t}_1;\Gamma_1 \quad [\![\tau_2;\Gamma_1]\!]_\tau = \mathsf{t}_2;\Gamma_2 \quad \mathsf{o} = \Gamma.\mathsf{nm}(o)\end{array}}{[\![\tau_1\ o\ \tau_2;\Gamma]\!]_\phi = (\mathsf{o}\ \mathsf{t}_1\ \mathsf{t}_2);\Gamma_2}$$

$$11\frac{\Gamma.\mathsf{nm}(name) = \mathsf{name}}{[\![name;\Gamma]\!]_\phi = \mathsf{name};\Gamma}$$

The rules for non-atomic formulas (12–14) are straightforward recursive applications to their arguments, propagating the context accordingly, and combining the results in the SMT-LIB syntax.

$$12\frac{[\![\phi;\Gamma]\!]_\phi = \mathsf{f};\Gamma'}{[\![\neg\phi;\Gamma]\!]_\phi = (\mathsf{not}\ \mathsf{f});\Gamma'}$$

$$13\frac{o \in \{\wedge,\vee\} \quad \mathsf{o} = \Gamma.\mathsf{nm}(o) \quad [\![\phi_1;\Gamma]\!]_\phi = \mathsf{f}_1;\Gamma_1\ \cdots\ [\![\phi_n;\Gamma_{n-1}]\!]_\phi = \mathsf{f}_\mathsf{n};\Gamma_n}{[\![\phi_1\, o\ \cdots\, o\ \phi_n;\Gamma]\!]_\phi = (\mathsf{o}\ \mathsf{f}_1\ \mathsf{f}_2 \cdots \mathsf{f}_\mathsf{n});\Gamma_n}$$

$$14\frac{o \in \{\Rightarrow,\Leftrightarrow\} \quad \mathsf{o} = \Gamma.\mathsf{nm}(o) \quad [\![\phi_1;\Gamma]\!]_\phi = \mathsf{f}_1;\Gamma_1 \quad [\![\phi_2;\Gamma_1]\!]_\phi = \mathsf{f}_2;\Gamma_2}{[\![\phi_1\, o\ \phi_2;\Gamma]\!]_\phi = (\mathsf{o}\ \mathsf{f}_1\ \mathsf{f}_2);\Gamma_2}$$

Rule 15 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 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.

$$15\ \frac{\begin{array}{cc} Q \in \{\exists, \forall\} & \mathsf{Q} = \Gamma.\mathsf{nm}(Q) \\ \Gamma.\mathsf{nm}(btype(x)) = \mathsf{s} & [\![\phi; \Gamma[\mathsf{nm} \oplus \{x \mapsto ?\mathsf{x}\}]]\!]_\phi = \mathsf{f}; \Gamma_1 \end{array}}{[\![Qx \bullet \phi; \Gamma]\!]_\phi = (\mathsf{Q}(?\mathsf{x}\ \mathsf{s})\ \mathsf{f}); \Gamma_2 = \Gamma_1[\mathsf{nm} \ominus x]}$$

4.6 Translation Rules for Terms

The operator $[\![]\!]_\tau$ recurses over the structure of terms found in Rodin lemmas, and builds terms according to the SMT-LIB syntax. The base cases of the recursion are identifiers (Rule 16) and numeric literals (Rule 17). Boolean conversion is also dealt with (Rule 18).

$$16\ \frac{\mathsf{id} = \Gamma.\mathsf{nm}(id)}{[\![id; \Gamma]\!]_\tau = \mathsf{id}; \Gamma} \quad 17\ \frac{}{[\![num; \Gamma]\!]_\tau = num; \Gamma} \quad 18\ \frac{[\![\phi; \Gamma]\!]_\phi = \mathsf{f}; \Gamma_1}{[\![\mathbf{bool}(\phi); \Gamma]\!]_\tau = \mathsf{f}; \Gamma_1}$$

Rules 19–21 specify the translation of arithmetic terms:

$$19\ \frac{o \in \{+, \times\} \quad \mathsf{o} = \Gamma.\mathsf{nm}(o) \quad [\![\tau_1, \Gamma]\!]_\tau = \mathsf{t_1}; \Gamma_1 \ \cdots\ [\![\tau_n, \Gamma_{n-1}]\!]_\tau = \mathsf{t_n}; \Gamma_n}{[\![\tau_1\ o \ldots o\ \tau_n; \Gamma]\!]_\tau = (\mathsf{o}\ \mathsf{t_1}\ \cdots\ \mathsf{t_n}); \Gamma_n}$$

$$20\ \frac{o \in \{\mathbf{div}, \mathbf{mod}, -\} \quad \mathsf{o} = \Gamma.\mathsf{nm}(o) \quad [\![\tau_1, \Gamma]\!]_\tau = \mathsf{t_1}; \Gamma_1 \quad [\![\tau_1, \Gamma_2]\!]_\tau = \mathsf{t_2}; \Gamma_2}{[\![\tau_1\ o\ \tau_2; \Gamma]\!]_\tau = (\mathsf{o}\ \mathsf{t_1}\ \mathsf{t_2}); \Gamma_2}$$

$$21\ \frac{[\![\tau, \Gamma]\!]_\tau = \mathsf{t}; \Gamma_1}{[\![-\tau; \Gamma]\!]_\tau = (\tilde{}\ \mathsf{t}); \Gamma_1}$$

The following rules specify the translation of set terms. The first group of rules deal with the base cases, i.e. the empty set (Rule 22), and sets defined in intention (Rule 23) and in extension (Rule 24).

$$22\ \frac{}{[\![\emptyset, \Gamma]\!] = \mathsf{empty}, \Gamma} \quad 23\ \frac{\Gamma.\mathsf{nm}(btype(x)) = \mathsf{s} \quad [\![\phi, \Gamma]\!]_\tau = \mathsf{f}; \Gamma_1}{[\![\{x \mid \phi\}; \Gamma]\!]_\tau = (\mathsf{lambda}\ (?\mathsf{x}\ \mathsf{s})\ \mathsf{f}); \Gamma_1}$$

$$24\ \frac{\Gamma.\mathsf{nm}(btype(\tau_1)) = \mathsf{s} \quad [\![\tau_1, \Gamma]\!]_\tau = \mathsf{t_1}; \Gamma_1 \quad \cdots \quad [\![\tau_n, \Gamma_{n-1}]\!]_\tau = \mathsf{t_n}; \Gamma_n}{[\![\{\tau_1, \cdots, \tau_n\}; \Gamma]\!]_\tau = (\mathsf{lambda}\ (?\mathsf{x}\ \mathsf{s})(\mathsf{or}\ (= ?\mathsf{x}\ \mathsf{t_1}) \cdots (= ?\mathsf{x}\ \mathsf{t_n}))); \Gamma_n}$$

The remaining classes of set expressions are translated according to Rules 25–27. Note that the translation of intersection and union needs to transform the variadic set connectives of Rodin to binary macros.

$$25\ \frac{o \in \{\cup, \cap\} \quad \mathsf{o} = \Gamma.\mathsf{nm}(o) \quad [\![\tau_1, \Gamma]\!]_\tau = \mathsf{t_1}, \Gamma_1 \quad [\![\tau_2, \Gamma_1]\!]_\tau = \mathsf{t_2}, \Gamma_2}{[\![\tau_1\ o\ \tau_2, \Gamma]\!] = (\mathsf{o}\ \mathsf{t_1}\ \mathsf{t_2}), \Gamma_2}$$

$$26\ \frac{o \in \{\cup, \cap\} \quad \mathsf{o} = \Gamma.\mathsf{nm}(o) \quad [\![\tau_1, \Gamma]\!]_\tau = \mathsf{t_1}, \Gamma_1 \quad [\![\tau_2\ o \cdots o\ \tau_n, \Gamma_1]\!]_\tau = \mathsf{t}, \Gamma_2}{[\![\tau_1\ o\ \tau_2\ o \cdots o\ \tau_n, \Gamma]\!] = (\mathsf{o}\ \mathsf{t_1}\ \mathsf{t}), \Gamma_2}$$

$$27\ \frac{o \in \{\setminus, ..\} \quad \mathsf{o} = \Gamma.\mathsf{nm}(o) \quad [\![\tau_1, \Gamma]\!]_\tau = \mathsf{t_1}, \Gamma_1 \quad [\![\tau_2, \Gamma_1]\!]_\tau = \mathsf{t_2}, \Gamma_2}{[\![\tau_1\ o\ \tau_2, \Gamma]\!] = (\mathsf{o}\ \mathsf{t_1}\ \mathsf{t_2}), \Gamma_2}$$

4.7 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:

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

Thus, (ismin m t) expands to a formula stating that m is equal to the smallest value in the set t. The translation of the operators **min** and **max** is specified by Rules 28 and 29. Each application of these rules add a fresh integer constant and an assumption to the context.

$$28\ \frac{[\![\tau, \Gamma]\!]_\tau = \mathsf{t}; \Gamma_1 \qquad \mathit{fresh}(\{\mathsf{m}\})\qquad \Gamma_2 = \Gamma_1[\mathsf{funs} \oplus \{\mathsf{m} \mapsto \mathsf{Int}\} \mid \mathsf{assumptions} \oplus \{(\mathsf{ismin}\ \mathsf{m}\ \mathsf{t})\}]}{[\![\mathbf{min}(\tau), \Gamma]\!]_\tau = \mathsf{m}; \Gamma_2}$$

$$29\ \frac{[\![\tau, \Gamma]\!]_\tau = \mathsf{t}; \Gamma_1 \qquad \mathit{fresh}(\{\mathsf{m}\})\qquad \Gamma_2 = \Gamma_1[\mathsf{funs} \oplus \{\mathsf{m} \mapsto \mathsf{Int}\} \mid \mathsf{assumptions} \oplus \{(\mathsf{ismax}\ \mathsf{m}\ \mathsf{t})\}]}{[\![\mathbf{max}(\tau), \Gamma]\!]_\tau = \mathsf{m}; \Gamma_2}$$

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.

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

Rule 30 specifies the translation of the predicate application $\mathbf{finite}(\tau)$: 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 τ. The context is also augmented with an assumption obtained by an expansion of macro finite.

$$30\ \frac{\begin{array}{c}[\![\tau, \Gamma]\!]_\tau = \mathsf{t}; \Gamma_1 \qquad \mathit{btype}(\tau) = \mathbb{P}(s) \qquad \Gamma.\mathsf{nm}(s) = \mathsf{s} \qquad \mathit{fresh}(\{\mathsf{p}, \mathsf{k}, \mathsf{f}\})\\ \Gamma_2 = \Gamma_1[\mathsf{preds} \oplus \{\mathsf{p} \mapsto ()\} \mid \mathsf{funs} \oplus \{\mathsf{k} \mapsto \mathsf{Int}, \mathsf{f} \mapsto (\mathsf{s}\ \mathsf{Int})\} \mid\\ \qquad\qquad \mathsf{assumptions} \oplus (\mathsf{finite}\ \mathsf{p}\ \mathsf{t}\ \mathsf{f}\ \mathsf{k})]\end{array}}{[\![\mathbf{finite}(\tau), \Gamma]\!]_\phi = \mathsf{p}; \Gamma_2}$$

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 $[1; \mathsf{k}]$.

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

Rule 31 specifies the translation of the application of operator **card** to a set τ; the context is enriched with a constant k and a function f, and an assumption that relates both new symbols to set τ using the macro card.

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

5 Experimental Results

An initial set of benchmarks was made available by an industrial partner. From this initial set were extracted the proof obligations that use only the constructs listed in Section 2.1: booleans, integers, and basic sets. 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.

The selected benchmarks were translated manually following the rules presented in Section 4. The resulting files were then processed with the SMT-solver veriT [10], that integrates the necessary pre-processing steps to handle the constructs introduced in these translation rules.

In the general case, an execution of veriT may yield four results. First, the execution may not halt within the resource bounds allocated (be it space or time). The second possible outcome is "unsat", i.e. unsatisfiability is detected; for some logics, veriT is then able to produce a trace of the reasoning steps that were applied to detect this unsatisfiability: this trace can be checked by a third-party tool and used as a certificate of the result. The last two possible results happen when veriT is not able to show the input formula is unsatisfiable. The input formula is then inspected and, if it belongs to a logic for which the solver is recorded to be complete, the result is "sat"; otherwise it is "unknown". In the case of the formulas addressed in this experiment, veriT is not complete and may only return the verdicts "unsat", "unknown" or timeout.

Table 1 summarizes the results. The first column identifies each lemma. The second column indicates what is the expected result. The third and fourth columns contain the result obtained with two versions of the SMT-solver veriT.

Name of RODIN lemma	logic	veriT 1	veriT 2	time
BepiColombo-thm15.smt	unsat	unsat	unsat	0.049s
BepiColombo-thm2.smt	unsat	unsat	unsat	0.010s
BoschSwitch-4.smt	unsat	unsat	unsat	0.030s
SSF_pilot-3.smt	unsat	unknown	unsat	0.031s
SSF_pilot-5.smt	unsat	unsat	unsat	0.011s
SimpleLyra-7.smt	sat	unknown	unknown	0.012s
ch4_other_file-1.smt	unsat	unsat	unsat	0.011s
ch7_conc-13.smt	unsat	unsat	unsat	0.011s
ch7_conc-21.smt	unsat	unsat	unsat	0.012s
ch7_conc-26.smt	unsat	unknown	unsat	0.033s
ch7_conc-28.smt	unsat	unknown	unsat	0.032s
ch910_ring-2.smt	unsat	unsat	unsat	0.010s
ch912_mobile-1.smt	unsat	unsat	unsat	0.031s
ch915_bin-2.smt	unsat	unknown	unsat	0.032s
ch915_maxi-7.smt	unsat	unsat	unsat	0.016s
ch915_mini-5.smt	unsat	unsat	unsat	0.015s
ch915_sort_other-3.smt	unsat	unsat	unsat	0.013s
gen_hotel_new-14.smt	unsat	unsat	unsat	0.035s
ssf-1.smt	unsat	unsat	unsat	0.016s
ssf-3.smt	unsat	unsat	unsat	0.016s
ssf-4.smt	unsat	unsat	unsat	0.014s
ssf-7.smt	unsat	unsat	unsat	0.013s

Table 1. Experimental results: verifying RODIN lemmas with veriT

The version labeled veriT 1 in the table, is the version that participated at SMT-COMP'2009, the yearly contest for SMT-solvers. For four lemmas, it is unable to prove that it is unsatisfiable. The reason was that all the corresponding formulas contained quantifiers, and the original instantiation heuristics of veriT failed to find the good instances. This motivated the extension of the quantifier instantation module of veriT to improve these results.

In the initial experiments, the quantifier instantiation in veriT was handled in two modules: an external theorem prover (namely, the E prover) to reason on purely equational first-order logic using the superposition calculus, and a component implementing custom quantifier instantation heuristics based on equalities and atom polarities in the quantified formulas. The latter module was then extended to implement a new heuristic that instantiates quantified variables using information from congruence closure, an internal module responsible for reasoning about equalities. Congruence closure maintains equivalence classes between the terms that are present in the solver; each such equivalence class has a representative term. In this heuristic, a quantified variable is instantiated with all the representative terms that have the same sort as the the variable. After including the new heuristics, no performance regression was observed compared to the results that were obtained with the initial version.

This new version, labeled veriT 2 in the table, includes the heuristics that uses the representative terms computed by congruence closure. With this new version, all unsatisfiable lemmas were successfully discharged. As expected, veriT reports an "unknown" verdict on the only lemma that cannot be proved.

Note that the verification time is negligible: the total time to prove all lemmas is less than one second. Even though the benchmarks are indeed small formulas with few hypothesis, these results are promising. Even in the case of the proof obligation that cannot be proved unsatisfiable, the result is returned very quickly.

6 Conclusions

This paper addresses the verification of proof obligations generated in formal systems and software developments using SMT-solvers. A fragment of the XML-based format for proof obligations in the RODIN platform was chosen: essentially it combines basic sets, 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. A 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 set of benchmark proof obligations, supplied by an industrial partner, 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. The initial version of the solver was already fast but, in some cases, was unable to produce the expected result. After extending the solver with a new (and simple) quantifier instantiation heuristics, all valid proof obligations were proved almost instantaneously. This experiment validate 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.

Future Work. The translation presented in this paper will thus be used as the specification for the implementation of a verification plug-in for the Rodin platform targeting SMT-solvers. The translation will then be extended to handle a larger number of specification constructs, namely basic binary relations and arithmetics for real numbers.

Further, the capabilities of SMT-solvers can be used to improve the workflow in the development platform. 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 [12]. 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, other translations can be defined to consider other classes of proof obligations. For instance, as suggested in [8, 6], sets can be mapped to arrays of booleans, for which efficient SMT-solving techniques are available.

Acknowledgements. The author thanks Laurent Voisin from Systerel for providing examples and Carine Pascal, also from Systerel, for sharing information 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.

References

1. ClearSy: Atelier B User Manual Version 4.0. Clearsy System Engineering. (2009) http://www.atelierb.eu.
2. Coleman, J., Jones, C., Oliver, I., Romanovsky, A., E.Troubitsyna: RODIN (rigorous open development environment for complex systems). In: Fifth European Dependable Computing Conference: EDCC-5 supplementary volume. (2005) 23–26
3. Nelson, G., Oppen, D.: Simplification by cooperating decision procedures. ACM Transactions on Programming Languages and Systems **1**(2) (1979) 245–257
4. Nieuwenhuis, R., Oliveras, A.: 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). Volume 3576 of Lecture Notes in Computer Science., Springer (2005) 321–334
5. Ranise, S., Tinelli, C.: The SMT-LIB standard : Version 1.2 (August 2006)
6. Kröning, D., Rümmer, P., Weissenbacher, G.: 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. (2009)
7. Bruun, H., Damm, F., Dawes, J., Hansen, B., Larsen, P., Parkin, G., Plat, N., Toetenel, H.: A formal definition of VDM-SL. Technical Report Technical Report 1998/9, University of Leicester (1998)
8. Couchot, J.F., Déharbe, D., Giorgetti, A., Ranise, S.: Scalable automated proving and debugging of set-based specifications. Journal of the Brazilian Computer Society **9**(2) (2003) 17–36
9. Hurlin, C., Chaieb, A., Fontaine, P., Merz, S., Weber, T.: Practical proof reconstruction for first-order logic and set-theoretical constructions. In Dixon, L., Johansson, M., eds.: The Isabelle Workshop 2007, Bremen, 16/07/07. (July 2007)
10. Bouton, T., de Oliveira, D.C.B., Déharbe, D., Fontaine, P.: verit: An open, trustable and efficient smt-solver. In Schmidt, R.A., ed.: CADE. Volume 5663 of Lecture Notes in Computer Science., Springer (2009) 151–156
11. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
12. Fontaine, P.: Combinations of theories for decidable fragments of first-order logic. In: 7th International Symposium on Frontiers of Combining Systems (FroCoS'09). Volume 5749 of Lecture Notes in Computer Science., Springer (2009)