1st International Workshop about Sets and Tools (SETS 2014)

Affiliated to ABZ 2014

Toulouse (France)

Preface

This volume contains the papers presented at SETS 2014: 1st International Workshop about Sets and Tools held on June 1, 2014 in Toulouse (France).

There were 6 submissions. Each submission was reviewed by at least 3, and on the average 3, program committee members. The committee decided to accept 6 papers. The program also includes 2 invited talks.

In the preparation of these proceedings and in managing the whole discussion process, Andrei Voronkov's EasyChair conference management system proved itself an excellent tool.

May 26, 2014 — David Delahaye
Paris, France — Catherine Dubois

Table of Contents

Program Committee

Maximiliano Cristia	CIFASIS-UNR
David Deharbe	Universidade Federal do Rio Grande do Norte
David Delahaye	Cedric/Cnam/Inria, Paris
Catherine Dubois	ENSIIE-CEDRIC
Mamoun Filali-Amine	IRIT
Michael Leuschel	University of Düsseldorf
Stephan Merz	INRIA Lorraine
Dominique Pastre	LIPADE - Universite Paris Descartes
Gianfranco Rossi	Universita' di Parma
Mark Utting	The University of Waikato
Benjamin Werner	INRIA
Freek Wiedijk	Radboud University Nijmegen
Wolfgang Windsteiger	RISC Institute, JKU Linz, Austria

Additional Reviewers

A

Arthan, Rob

K

Krings, Sebastian

Proof verification within Set Theory: Exploiting a new way of modeling graphs

Eugenio G. Omodeo

University of Trieste (Italy), DMG/DMI

This talk illustrates proof-verification technology based on set theory, also reporting on experiments carried out with ÆtnaNova, aka Ref (see [6, 4]).

The said verifier processes script files consisting of definitions, theorem statements and proofs of the theorems. Its underlying deductive system—mainly first-order, but with an important second-order construct enabling one to package definitions and theorems into reusable proofware components—is a variant of the Zermelo-Fraenkel set theory, ZFC, with axioms of regularity and global choice. This is apparent from the very syntax of the language, borrowing from the set-theoretic tradition many constructs, e.g. abstraction terms. Much of Ref's naturalness, comprehensiveness, and readability, stems from this foundation; much of its effectiveness, from the fifteen or so built-in mechanisms, tailored on ZFC, which constitute its inferential armory. Rather peculiar aspects of Ref, in comparison to other alike proof-assistants (cf., e.g., [2, 1]), are that Ref relies only marginally on predicate calculus and that types play no prominent role, in it, as a foundation.

The selection of examples, mainly referred to graphs, to be discusses in this talk, reflects today's tendency [5] to bring Ref's use closer to algorithm-correctness verification. To achieve relatively short, formally checked, proofs of properties enjoyed by claw-free graphs, we took advantage of novel results [3] about representing their (undirected) edges via membership.

Acknowledgements. *Partial funding was granted by the INdAM/GNCS 2013 project "*Specifica e verifica di algoritmi tramite strumenti basati sulla teoria degli insiemi*".*

References

1. C. E. Brown. Combining type theory and untyped set theory. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 205–219. Springer, 2006.
2. R. Matuszewski and P. Rudnicki. Mizar: the first 30 years. *Mechanized Mathematics and its Applications*, 4(1):3–24, 2005.
3. M. Milanič and A. I. Tomescu. Set graphs. I. Hereditarily finite sets and extensional acyclic orientations. *Discrete Applied Mathematics*, 161(4-5):677–690, 2013.
4. E. G. Omodeo. The Ref proof-checker and its "common shared scenario". In Martin Davis and Ed Schonberg, editors, *From Linear Operators to Computational Biology: Essays in Memory of Jacob T. Schwartz*, pages 121–167. Springer, 2012. With an appendix, *Claw-free graphs as sets*, co-authored by A. I. Tomescu.

5. E. G. Omodeo and A. I. Tomescu. Set graphs. III. Proof Pearl: Claw-free graphs mirrored into transitive hereditarily finite sets. *Journal of Automated Reasoning*, 52(1):1–29, 2014.
6. J.T. Schwartz, D. Cantone, and E.G. Omodeo. *Computational Logic and Set Theory - Applying Formalized Logic to Analysis*. Springer, 2011.

Mathematical theorem proving, from Muscadet0 to Muscadet4, why and how ?

Dominique Pastre

Universite Paris Descartes, Paris, France
pastre@math-info.univ-paris5.fr

We will present the ideas and the choices which have been made throughout the development of the Muscadet theorem prover. We will first see the principles and main ideas which lead to a first prover in the context of the time, influenced by a famous paper by Woody Bledsoe. This program used natural methods and was applied to set theory. It was then rewritten as a knowledge based-system where an inference engine applied rules, given or automatically built by metarules which expressed general or specific mathematical knowledge. It has been applied to some difficult problems. In order to allow more flexibility for expressing knowledge, it has been rewritten in Prolog, allowing the knowledge to be more or less declarative or procedural. To work with the TPTP Library the system had to work without knowing anything about mathematics except predicate calculus. All mathematical concepts had to be defined with mathematical statements, and the belonging relation handled as any other binary relation. To avoid translating knowledge to TPTP syntax, TPTP syntax has been used (this unfortunately forbade the use of some abbreviations mathematicians are comfortable with). Last but not least, the relevant trace has been extracted to give a proof easily read by anyone, except in the case of failure, when all steps may be displayed to understand (manually) the reasons for the failure. Muscadet has participated to CASC competitions. The results show its complementarity with regard to resolution-based provers.

Rapid Prototyping and Animation of Z Specifications Using {*log*}

Maximiliano Cristiá[1] and Gianfranco Rossi[2]

[1] CIFASIS and UNR, Rosario, Argentina
cristia@cifasis-conicet.gov.ar
[2] Università degli Studi di Parma, Parma, Italy
gianfranco.rossi@unipr.it

Abstract. Prolog has been proposed as the programming language on which animation and prototyping of Z specifications should be based. However, we believe there is still room for improvements. In this paper, we want to revisit this issue in the light of a powerful, set-oriented constraint programming language like {*log*} (pronounced 'setlog'). In particular, we pay attention to three points that we think are crucial: finding solutions to complex state predicates; defining formal criteria for guiding an evaluation process based on prototypes; and automatically adding graphical user interfaces to prototypes generated from Z specifications. Three examples of information systems prototypes are available online.

1 Running Example

Assume you are eliciting the user requirements for an ERP system. In a first meeting, your customer asks the following:

(R1) Keep a record of client accounts where the company registers all the credits and debits of its clients.
(R2) Get a listing of all the accounts whose balance is in a given range.

Given that you have experience in requirement engineering, you know that probably your customer has told you (R1) and (R2) without giving them too much thought. You would like to be sure that these are the real, final requirements before engaging your team in programming unstable, unvalidated requirements. Following the software engineering literature you know that it would be of great help if you could show your customer a prototype of these requirements.

Now, suppose you know of a tool that automatically generates *functional* prototypes of some kind of Z specifications, provided they are annotated with user interface directives. Further, let us say that in your team there is an engineer who is knowledgeable in the Z notation. Would it be cost-effective to write a Z specification of (R1) and (R2) and use that tool to generate a prototype so your customer can validate them? In this way, your programmers will implement validated requirements; moreover, they will do it from a formal specification. All this would amount to a much better product with less iterations to improve its

quality. Even more, assume that this prototyping tool records all the interactions of users. Then, later, these interactions can be used as *functional* test cases. So the tool would also help you during the testing process.

After considering all these gains, you decide to ask your team to write the Z specification shown in Figure 1 and the GUI specification shown in 3 which, when fed into the prototyping tool, become the Prolog+{*log*}+XPCE program [25, 11, 36] partially shown in Figure 4. When this program is executed the user can play with an application presenting an acceptable GUI like the one shown in Figure 2—download this and two more examples from [6].

$$[UID, NAME]$$

$$ERP == [clients : UID \pfun NAME;\ accounts : UID \pfun \mathbb{Z} \mid \mathrm{dom}\ clients = \mathrm{dom}\ accounts]$$

$$InitERP == [ERP \mid clients = \emptyset \land accounts = \emptyset]$$

NewClientOk

$$\Delta ERP;\ u? : UID;\ n? : NAME$$

$$u? \notin \mathrm{dom}\ clients$$
$$clients' = clients \cup \{u? \mapsto n?\}$$
$$accounts' = accounts \cup \{u? \mapsto 0\}$$

TransactionOk

$$\Delta ERP;\ n? : UID;\ m? : \mathbb{Z}$$

$$n? \in \mathrm{dom}\ accounts \land m? \neq 0$$
$$accounts' = accounts \oplus \{n? \mapsto accounts\ n? + m?\}$$
$$clients' = clients$$

$$ClientAlreadyExists == [\Xi ERP;\ u? : UID \mid u? \in \mathrm{dom}\ clients]$$
$$NewClient == NewClientOk \lor ClientAlreadyExists$$
$$AccountNotExists == [\Xi ERP;\ n? : UID \mid n? \notin \mathrm{dom}\ accounts]$$
$$NullAmount == [\Xi ERP;\ m? : \mathbb{Z} \mid m? = 0]$$
$$Transaction == TransactionOk \lor AccountNotExists \lor NullAmount$$
$$AccountsInRangeOk == [\Xi ERP;\ a?, b? : \mathbb{Z};\ r! : \mathbb{P}\ UID \mid a? \leq b? \land r! = \mathrm{dom}(accounts \rhd (a? \mathinner{..} b?))]$$
$$NotARange == [\Xi ERP;\ a?, b? : \mathbb{Z};\ r! : RET \mid a? > b?]$$
$$AccountsInRange == AccountsInRangeOk \lor NotARange$$

Fig. 1: Excerpt of the Z specification of (R1) and (R2)

Z and {*log*} will be introduced in sections 2 and 3, respectively, and how prototypes would be automatically generated in Sect. 4. Now we want to show

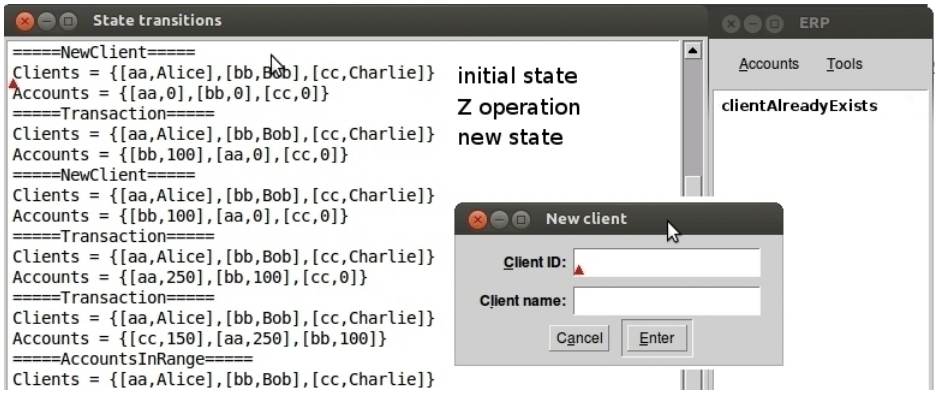


Fig. 2: Screenshot of the prototype

what users and software engineers can do with these prototypes. Note that the GUI is divided into two panels. The one on the left, STATE TRANSITIONS, displays all the state transitions executed while the application is used. It displays the initial state by giving the value for each state variable, the name of the Z operation (transition) executed and the resulting new state—input and output variables are not displayed to keep the example manageable but they can be trivially added. The panel on the right, named ERP in this case, is the actual prototype which presents a menu bar. This menu bar contains as many pull-down menus as indicated in the GUI specification plus the TOOLS menu automatically added to all prototypes. Although not shown in Figure 2, the ACCOUNTS menu contains three options (NEW CLIENT, INFORM TRANSACTION and ACCOUNTS IN RANGE) as indicated by the GUI specification given in Figure 3. The TOOLS menu contains only one option called SET STATE. When users click on a given menu option a dialog box opens, as, for instance, the one shown in the figure which corresponds to the NEW CLIENT option. Again, dialog boxes display GUI widgets according to the GUI specification, plus default Cancel and Enter buttons. Once a dialog box is filled in and the Enter button is pressed, the dialog calls the corresponding Z operation which in turn updates the state and prints some output. Output can be printed in the same dialog-box or in the application main window, as can be seen in the figure. In summary, the prototype executes all the specified operations through a reasonably real GUI.

Allowing end-users to play with the prototype as with a final application is important but they should do it trying to see if you (the requirement engineer) have correctly understood what they asked you for. For instance, executing *AccountsInRange* from the initial state of the system (Figure 1) says little about whether requirement (R2) has been correctly understood or not, because the answer will be the empty set regardless of the values of $a?$ and $b?$. Therefore, it would be better if the operation is executed when the state of the system contains some client accounts with different balances. One possible way of doing this is entering some clients and transactions by using the different menus and options.

Clearly, this is annoying, time consuming and error prone. Alternatively, you can think of a state predicate such that every possible solution would enable satisfactory executions of the operation. For example, solutions to $\#\,\mathrm{ran}\,accounts > 3$ are partial functions that contain at least four elements in the range, which implies at least three elements in the domain. Then, if the state from which *AccountsInRange* is executed satisfies that formula, $\mathrm{dom}(accounts \rhd (a? \,..\, b?))$ (see *AccountsInRangeOk*) can be a set with different numbers of elements depending on the values set for $a?$ and $b?$.

The SET STATE option in the TOOLS menu allows you to enter a Z predicate depending on the state variables of the system. $\{log\}$ tries to find a solution for the goal and, if it succeeds, this solution is bound to the state variables. In other words, the system transitions to a state verifying that goal. Most of Z predicates can be translated into $\{log\}$ goals [7]. Hence, if you call SET STATE, enter the $\{log\}$ representation of $\#\,\mathrm{ran}\,accounts > 3$, and call *AccountsInRange* there will be several accounts with different balances making it possible to set interesting values for $a?$ and $b?$, thus generating different listings. This would allow your end-users to validate requirement (R2) through the prototype by evaluating whether its answers are correct with respect to their expectations. Finding solutions to state predicates is further discussed in Sect. 5.

However, why $\#\,\mathrm{ran}\,accounts > 3$ is the proper condition? Are there other conditions that should be sought? How can they be discovered? Model-based testing (MBT) methods face a similar problem. In Sect. 6 we propose to apply the Test Template Framework (TTF) [32], a MBT method tailored to the Z notation, to discover the key conditions to validate user requirements.

Positive side effects. Say that your customer approves the prototype. This implies that all the transitions recorded by the prototype (cf. panel STATE TRANSITIONS) are correct. You can use them to conduct system and acceptance testing because they represent the runs that your customer has validated. If the final system behaves the same with respect to these executions, your customer should be satisfied with the product. Note that these executions can be replayed in the final product because they are actual values for the important variables—however, some semi-automatic *reification* or *refinement* will be needed, see [5] for an approach to this problem. Furthermore, if the TTF is applied to guide the validation process, these runs can be used for unit testing too. Even more: you have a formal specification of the requirements ready to be handed to the programming team. All this should account for a much better product at the price of writing a Z specification.

Warning. The prototype generation tool that we described in this introduction is not fully implemented yet, but we think it is feasible because: translations from Z to Prolog have already been proposed; $\{log\}$ can solve many complex set predicates [7] making it possible to implement the SET STATE feature; and the TTF has been implemented by the Fastest tool which has been applied to several Z specifications to discover complex testing specifications [4]. We believe that the problem is putting all these techniques and tools to work together.

2 Brief Introduction to the Z notation

Z is a formal specification language based on first-order logic and set theory [30]. It can be used to specify software systems in many different ways. However, in this paper we will consider that specifications take the form of state machines, which is how the notation is widely used. This way of writing specifications is based on a set of interpretations and conventions. Then, all the results presented in this paper hold if specifiers follow them. We will introduce some details of Z by describing the meaning of the specification presented in Figure 1, which formalizes part of (R1) and (R2). *UID* and *NAME* are two uninterpreted sorts (called basic or given types in Z). *ERP* is a so-called schema declaring two variables, *clients* and *accounts*. *clients* is a partial function from *UID* into *NAMES*. *ERP* also restricts the possible values of *clients* and *accounts* to those that have equal domains. *clients* represents the set of agents who are currently clients of the company; *accounts* holds the current balance of each client's account. In this way, *ERP* is the state space of the state machine. In Z, ordered pairs are written as $a \mapsto b$, and partial functions are sets of ordered pairs. Then, *InitERP* simply states that in the initial state both partial functions are empty.

NewClientOk is an operation schema because it defines a state transition. This can be noted by the ΔERP expression. ΔERP declares *clients*, *accounts*, $clients'$ and $accounts'$ where $clients'$ and $accounts'$ represent the value of *clients* and *accounts* in the after-state. In Z state transitions can also consume inputs and produce outputs. Inputs variables are decorated with '?' while output ones with '!'. The predicate part of schema *NewClientOk* states the pre-conditions and post-conditions that define the operation. In this case, the agent for whom the account is being opened must be new to the company; this is formalized by asking that his/her ID must not belong to the domain of *clients*. If this is the case, his/her ID and name are added to the client "database"; and a new account whose balance is zero is added. Note how set expressions are used even for partial functions and how the new state is defined. For instance, $accounts' = accounts \cup \{u? \mapsto 0\}$ states that the value of *accounts* in the after-state must be equal to its value in the before-state plus the ordered pair $u? \mapsto 0$.

Schema *ClientAlreadyExists* states that the system must remain the same when the client already exists. This is formalized by the expression ΞERP which implicitly cojoins $clients' = clients \land balances' = balances$ to the schema. The final operation, *NewClient*, is specified as the disjunction of the two previous schemas. *NewClient* is also a schema defined by a schema expression.

Since the specification of the other operations follows a similar structure, we will only comment about the new expressions. $\oplus$ is a relational operator defined as $f \oplus g = (\mathrm{dom}\, g \lhd f) \cup g$ where $\lhd$ is the domain anti-restriction operator. That is, $A \lhd f = \{p : f \mid p.1 \notin A\}$. In turn, $\rhd$, used in *AccountsInRangeOk*, is the domain restriction operator defined as $A \rhd f = \{p : f \mid p.1 \in A\}$. These operators are part of the Z mathematical toolkit (ZMT) [26], which is a rich and expressive collection of mathematical operators supported by Z. In particular, we would like to emphasize that the notion of set is essential in Z because it is used to define binary relations, partial and total functions, sequences and bags.

Therefore, it is tantamount to the success of an animation or rapid prototyping tool for Z specifications to be based on a powerful, efficient set constraint solver.

3 Brief Introduction to {*log*}

{*log*} [9, 10, 25] is a constraint logic programming (CLP) language that extends Prolog with general forms of sets and basic set-theoretic operations in the form of primitive constraints. Sets are primarily designated by *set terms*, that is, terms of one of the forms: {}, whose interpretation is the empty set, or $\{t_1, \ldots, t_n \mid s\}$, where s is a set term, whose interpretation is the set $\{t_1\} \cup \{t_2\} \cup \cdots \cup \{t_n\} \cup s$. The kind of sets that can be constructed in {*log*} are the so-called *hereditarily finite sets*, that is finitely nested sets that are finite at each level of nesting. Note that similarly to Prolog's lists, a set $\{t_1, \ldots, t_n \mid s\}$ can be partially specified, in that either some of its elements $t_1, \ldots, t_n$ or the remaining part s can contain unbound variables (hence "unknowns"). Sets can be also denoted intentionally by set formers of the form $\{\mathsf{X} : \mathsf{exists}([\mathsf{Y_1}, \ldots, \mathsf{Y_n}], \mathsf{G})\}$, where G is a {*log*}-goal (see below) and $\mathsf{X}, \mathsf{Y_1}, \ldots, \mathsf{Y_n}$ are variables occurring in G. Finally, sets can be denoted by *interval terms*, that is terms of the form $\mathsf{int}(\mathsf{a}, \mathsf{b})$, where a and b are integer terms, whose interpretation is the integer interval $[a, b]$.

Basic set-theoretic operations are provided in {*log*} as predefined predicates, and dealt with as constraints. For example, the predicates in and nin are used to represent membership and not membership, respectively; the predicate subset represents set inclusion (i.e., $\mathsf{subset}(\mathsf{r}, \mathsf{s})$ holds if and only if $r \subseteq s$ holds); while inters represents the intersection relation (i.e., $\mathsf{inters}(\mathsf{r}, \mathsf{s}, \mathsf{t})$ holds if and only if $t = r \cap s$). Basically, a {*log*}-*constraint* is a conjunction of such atomic predicates. For example,

$$1 \text{ in } \mathsf{R} \;\&\; 1 \text{ nin } \mathsf{S} \;\&\; \mathsf{inters}(\mathsf{R}, \mathsf{S}, \mathsf{T}) \;\&\; \mathsf{T} = \{\mathsf{X}\}$$

where R, S, T and X are variables, is an admissible {*log*}-constraint, whose interpretation states that set T is the intersection between sets R and S, R must contain 1 and S must not, and T must be a singleton set.

The original collection of set-based primitive constraints has been extended in [22] to include simple integer arithmetic constraints over Finite Domains as provided by CLP(FD) systems (cf. e.g. [24]).

The {*log*}-interpreter includes a *constraint solver* that is able to check satisfiability of {*log*}-constraints with respect to the underlying set and integer arithmetic theories. Moreover, when a constraint c holds, the constraint solver is able to compute, one after the other, all its solutions (i.e., all viable assignments of values to variables occurring in c). In particular, automatic labeling is called at the end of the computation to force the assignment of values from their domains to all integer variables occurring in the constraint, leading to a chronological backtracking search of the solution space. Possibly, remaining irreducible constraints are also returned as part of the computed answer.

Clauses, goals, and programs in {*log*} are defined as usual in CLP. In particular, a {*log*}-*goal* is a formula of the form $\mathsf{B_1} \;\&\; \mathsf{B_2} \;\&\; \cdots \;\&\; \mathsf{B_k}$, where $\mathsf{B_1}, \ldots, \mathsf{B_k}$

are either user-defined atomic predicates, or atomic $\{log\}$-constraints, or disjunctions of either user-defined or predefined predicates, or Restricted Universal Quantifiers (RUQs). Disjunctions have the form $\mathsf{G_1}$ or $\mathsf{G_2}$, where $\mathsf{G_1}$ and $\mathsf{G_2}$ are $\{log\}$-goals, and are dealt with through non-determinism: if $\mathsf{G_1}$ fails then the computation backtracks, and $\mathsf{G_2}$ is considered instead. RUQs are atoms of the form $\mathsf{forall(X\ in\ s, exists([Y_1, \ldots, Y_n], G))}$, where s denotes a set and G is a $\{log\}$-goal containing $\mathsf{X, Y_1, \ldots, Y_n}$. The logical meaning of this atom is $\forall X(X \in s \Rightarrow \exists Y_1, \ldots, Y_n : G)$, that is G represents a property that all elements of s are required to satisfy. When s has a known value, the RUQ can be used to iterate over s, whereas, when s is unbound, the RUQ allows s to be non-deterministically bound to each set satisfying the property G.

The following is an example of a $\{log\}$ program:

$\mathsf{is\_rel(R)}$:- $\mathsf{forall(P\ in\ R, exists([X, Y], P = [X, Y]))}$.
$\mathsf{dom(\{\}, \{\})}$.
$\mathsf{dom(\{[X, Y]/Rel\}, Dom)}$:- $\mathsf{dom(Rel, D)}\ \&\ \mathsf{Dom = \{X/D\}}\ \&\ \mathsf{X\ nin\ D}$.

This program defines two predicates, is_rel and dom. $\mathsf{is\_rel(R)}$ is true if R is a binary relation, that is a set of pairs of the form $\mathsf{[X, Y]}$. $\mathsf{dom(R, D)}$ is true if D is the domain of the relation R. The following is a goal for the above program:

$$\mathsf{R = \{[1, 5], [2, 7]\}}\ \&\ \mathsf{is\_rel(R)}\ \&\ \mathsf{dom(R, D)}$$

and the computed solution for D is $\mathsf{D} = \{1, 2\}$. It is important to note that $\mathsf{is\_rel(R)}$ can be used both to test and to compute R; similarly, $\mathsf{dom(R, D)}$ can be used both to compute D from R, and to compute R from D, or simply to test whether the relation represented by dom holds or not.

As can be seen, is_rel and dom are $\{log\}$ implementations of the corresponding concepts in the ZMT. They are part of a $\{log\}$ library implementing almost all the ZMT [7]. Some elements of the ZMT are not supported yet because they are beyond of set theory and, in some cases, would require some extensions to $\{log\}$ or a more complex translation. For example, generic and axiomatic definitions and recursive types are not fully yet supported.

$\{log\}$ has important advantages compared to other Prolog-based tools that can deal with sets. With respect to the simpler scheme which constructs set abstractions on top of an existing Prolog system, typically by using lists (cf. e.g. [21]), $\{log\}$ demonstrates its superiority whenever one has to deal with partially specified sets. For example, even the simple problem of comparing two sets, such as $\{a \mid X\}$ and $\{b \mid Y\}$, may lead to an infinite collection of answers when sets and set operations are implemented using lists, whereas in $\{log\}$ it is dealt as a set unification problem [12] which admits the single more general solution $X = \{b \mid S\}$ and $Y = \{a \mid S\}$ where S is a new variable. Similar considerations hold also when the negative counterparts of the basic set-theoretic operations, such as *'not equal'* and *'not member'*, are taken into account. The use of the list-based implementation of sets, in conjunction with the *Negation as Failure* rule provided by most implementations of Prolog, leads to well-known problems whenever non-ground atoms are involved.

Viewing sets as first-class entities and operations on sets as constraints, as done in $\{log\}$, provides a much more convenient solution. With respect to other proposals which deal with set constraints, such as [13], the main advantage of $\{log\}$ is its generality: sets in $\{log\}$ can contain elements of any type, can be nested and partially specified, whereas sets in the so-called set-constraint languages are restricted to flat sets of known integer elements. Moreover these proposals usually require a finite domain (i.e. a set of sets) is specified for each set variable occurring in a constraint, whereas this is not the case for $\{log\}$. On the other hand, the (incomplete) solvers of the set-constraint languages turn out to be in many cases more efficient than the general solver of $\{log\}$.

$\{log\}$ is fully implemented in Prolog and can be downloaded from [25].

4 Automatic Prototype Generation from Z Specifications

Generating a Prolog prototype from a Z specification involves the automatic translation from Z into Prolog. There are several proposals [8, 31, 3, 17, 14, 29, 33, 18, 37, 15, 23, 35, 7] that confirm that it is possible to implement such a *compiler*. There are also similar works for the B method [34, 1, 16] which defines a mathematical library like the ZMT. However, most of these works are aimed at animating the specification rather than using it as a means to validate user requirements. Then, they do not pay attention to equip the prototype with an acceptable user interface. Animation is thought to be a verification activity carried out by software engineers who want to analyze if a specification verifies some properties; in this sense animation is a complement or an alternative to proof. Instead, we think that if the resulting (Prolog) program is augmented with a GUI, animation becomes rapid prototyping and can be used (also) for functional requirements validation. That is, the prototype can be used by end-users to validate whether or not requirement engineers have understood what the system is supposed to do. Sterling et al. [31] aim at a similar target but they think in a Z-to-Prolog compiler that generates a command-line-like program that users are supposed to play with. We think that this is impractical. Besides, only a handful of these works [1, 23, 16] rely on a powerful, set-oriented constraint language implementing set constraint satisfaction, and even in these cases the underlying mathematics are not described in detail. B-Motion and Brama [28, 19] are two tools based on the B method which go in the same direction proposed here. B-Motion is based on ProB [20]. Both tools seem to be oriented to graphically represent software-controlled physical systems rather than information systems as the one described in the motivation example and the two other examples available on-line [6].

Therefore, we propose to define a simple, GUI specification language (GEL) of which Figure 3 is an example. This language should allow engineers to define the *structure* of a GUI and how it connects with the (Prolog) program generated from the Z specification. This GEL would not need to express the *behavior* of the GUI (for instance, specify when part of the GUI should be disabled).

GUI specification for ERP
Pull-down menu ACCOUNTS has
  Option NEW CLIENT opens ADD NEW CLIENT
  Option INFORM TRANSACTION opens ADD A CREDIT OR DEBIT
  Option ACCOUNTS IN RANGE opens LIST ALL ACCOUNTS IN RANGE
end of menu

. . . . . .

Dialog-box ADD A CREDIT OR DEBIT calls *Transaction* where
  n? is read by dom *clients*
  n? is a `list` labeled as SELECT ID
  m? is a `int` labeled as AMOUNT
end of dialog box

Dialog-box LIST ALL ACCOUNTS IN RANGE calls *AccountsInRange* where
  a? is a `int` labeled as MIN VALUE
  b? is a `int` labeled as MAX VALUE
  r! is a `text` labeled as ACCOUNTS ID IN RANGE
end of dialog box

end of GUI specification

Fig. 3: GUI specification of the Z specification of Figure 1

Then, after a meeting with the client, requirement engineers write Z and GUI specifications and use the prototype generation tool to get the first prototype. In this way, they can go to the next meeting with a prototype over which end-users can validate the requirements [27]. If users propose changes, the Z and GUI specifications are changed accordingly and the tool is run again. This allows an iterative, prototype-based process of requirement elicitation and validation.

The Z specification can be compiled into a Prolog program by a combination between the proposals by Sterling, Ciancarini and Turnidge [31] and by Cristiá, Rossi and Frydman [7]. The main contribution here is to use $\{log\}$ as the constraint solver instead of the list-based implementation of sets proposed by Sterling and his colleagues because this enables the possibility of implementing the SET STATE command discussed in Sect. 5 and the validation method introduced in Sect. 6. Besides, the tool must take into account also the GUI specification accompanying the Z specification. Here XPCE, the SWI-Prolog native GUI library [36], comes handy because it allows a seamless integration of the functional (Z) and the GUI specifications.

The main rules for compiling GUI specifications are the following (follow figures 3 and 4 as an aid to understand these rules):

1. The GUI is implemented inside a XPCE frame object.
2. The menu-bar is implemented as a XPCE menu_bar object.
3. The GUI specification must contain at least one 'pull-down menu' directive.
4. Each 'pull-down menu' directive is compiled as a XPCE popup object.
5. Each 'option' directive in a 'pull-down menu' must open a 'dialog-box'.
6. Each 'option' directive in a 'pull-down menu' is compiled to:

menu_item(option, message(@prolog, dialogBox))

```
initERP :-
    saveState({}, {}), nb_setval(clients, {}), nb_setval(accounts, {}), mainMenu.

mainMenu :-
    Builds a menu bar according to the GUI specification. Each option
    of the pull-down menus calls a predicate like readAccountsInRange below

readAccountsInRange :-
    new(D, dialog('List all accounts in range')),
    send_list(D, append,
              [new(M, int_item(min_value)), new(N, int_item(max_value)),
               text('Accounts ID in range'), new(B, browser),
               button(enter, message(@prolog, accountsInRange, B,
                                     M?selection, N?selection))]),
    send(D, open).

accountsInRange(B, M, N) :- accountsInRangeOk(B, M, N); notARange(B, M, N).

accountsInRangeOk(B, M, N) :-
    b_getval(clients, C), b_getval(accounts, A),
    setlog(M ein int(0, 10000) & N ein int(0, 10000) & M =< N &
           Y = int(M, N) & rres(Y, A, Y1) & dom(Y1, Y2) & set_to_list(Y2, L)),
    writeOutput(B, L), saveNewState(C, A).
```

Fig. 4: Code excerpt of the prototype of the Z specification shown in Figure 1

where dialogBox is a Prolog predicate implementing the dialog-box that the option must open.

7. Each dialog-box is implemented as a XPCE dialog object.
8. Each dialog-box must call an existing Z operation.
9. Each dialog-box must indicate what kind of widget must be used for each and every input and output variable declared in the corresponding Z operation.
10. `text` is implemented as a XPCE text_item object for inputs, and as a browser for outputs; `int` as int_item; `list` as list_browser; etc.
11. A 'is read by' directive in a 'dialog-box' directive simply states that the input variable is read by means of another expression. The selection made through the expression is passed as the value for the input variable.
12. When the enter button in a dialog-box is pressed the following is executed:

    button(enter, message(@prolog, Operation, Input))

    where Operation is a Prolog predicate implementing the corresponding Z operation (this code was generated when the Z specification was compiled) and Input are the input variables waited by the Z operation where each actual value corresponds to one of the values read by the widgets of the dialog-box.

Two more examples of this translation can be downloaded from [6]. They include the Z and GUI specifications and the Prolog+{*log*}+XPCE program.

5 Solving State Predicates for Requirement Validation

In this section we discuss the SET STATE feature presented in the introduction. The inclusion of this feature is based on the following observation: *validating some functional requirements involves taking the prototype to a state such that state variables satisfy a (complex) predicate.* In turn, this can be achieved in three ways: a) use the prototype's GUI to put itself in the desired state; b) write a simple Prolog predicate where each state variable is bound to the desired value and then execute the prototype from this state; and c) write the state predicate, give it to a constraint solver, and use its answer as the prototype's new state.

The first option is annoying, time consuming and error prone; this is the one discussed in [31]. The second one, is error prone because the values may be wrong. It can be improved by: writing the state predicate as in c), proposing a solution to it as in b), and asking the constraint solver whether this solution indeed satisfies the predicate. If it answers "no", try another value. This may take some time but it is safe. However, the best option is c) because the constraint solver does all the work—except proposing the state predicate, see Sect. 6.

The key condition for c) to be feasible is to have a constraint solver capable of solving (complex) Z predicates. Since Z predicates are first-order predicates over the set theory, only a constraint solver for such set theory can deliver the desired results. {*log*} may be such a constraint solver because, according to the results we have obtained when we used it as a test case generator for Fastest [7], it is able to solve many complex Z predicates. In effect, when {*log*} is used as Fastest's test case generator it needs to find solutions to complex Z predicates describing equally complex test conditions. However, not all possible Z predicates can be solved by {*log*} in its current state, since many Z facilities are implemented as user-defined predicates and the {*log*} interpreter can not guarantee efficient processing or even termination for all the considered formulas.

Therefore, the SET STATE menu option of the prototypes generated by our method would wait for either a Z predicate or a {*log*} goal, translate it into {*log*} in the first case, call it and use its answer to set the new state of the prototype. The following considerations must be taken into account:

1. If a Z predicate is entered it would be automatically translated into a {*log*} goal [7]. However, there are cases in which an automatic translation may not yield the best {*log*} code in the sense of {*log*} being able to find a solution for it in a reasonable time. An alternative could be to let users to edit the resulting {*log*} goal to help the tool to find a solution.
2. Typing information about the state variables must be entered along the goal; this has been discussed in [7].
3. Z basic types (i.e. *UID* and *NAME* in Figure 1) pose a problem because their elements have no structure and so {*log*} generates variables instead of constants. This can be solved by post-processing {*log*}'s answer.
4. If some {*log*} goal entered by the user is too complex to be solved, (s)he can write special-purpose {*log*} predicates to help the tool to find a solution.

For example, $\#\,\mathrm{ran}\,accounts > 3$, i.e. the state predicate discussed in the introduction, can be translated as follows:

$$\mathsf{pfunFromRan}(\mathsf{D},\mathsf{R},\mathsf{Accounts})\,\&\tag{1}$$
$$\mathsf{solve}(\mathsf{size}(\mathsf{R},\mathsf{SR})\ \&\ \mathsf{SR}>3)\,\&\tag{2}$$
$$\mathsf{subset}(\mathsf{R},\mathsf{int}(-10000,10000)\ \&\ \mathsf{is\_pfun}(\mathsf{Clients})\,\&\tag{3}$$
$$\mathsf{dom}(\mathsf{Clients},\mathsf{D}).\tag{4}$$

where: (1) builds Accounts from its range, R, and with domain included in D (note that the solution must be a value for Accounts and not only for its range as might be suggested by the original Z predicate); (2) sets the size of R to something greater than 3; (3) gives type information of R and Clients; and (4) is the invariant. {*log*} immediately returns the following:

$$\mathsf{Accounts} = \{[\mathsf{g45},-10000],[\mathsf{g30},-9999],[\mathsf{g15},-9998],[\mathsf{g00},-9997]\}$$
$$\mathsf{Clients} = \{[\mathsf{g45},\mathsf{g630}],[\mathsf{g30},\mathsf{g573}],[\mathsf{g15},\mathsf{g516}],[\mathsf{g00},\mathsf{g459}]\}$$

6 Rigorous Requirement Validation Based on Prototypes

In this section we discuss how scenarios for requirement validation can be automatically generated from the Z specification. These scenarios are a guide for users so they can chose which are worth to be explored and which are not. Each of these scenarios will be given by a Z predicate in terms of the state and input variables of a given Z operation. Then, each scenario will contribute to validate the requirements formalized by the corresponding Z operation. In order to perform the validation, the software engineer must enter each of them in the Set state function discussed in Sect. 5 and then execute the implementation of the corresponding Z operation. End-users must observe the output generated by the prototype and classify it as correct or incorrect.

Our proposal for a rigorous requirement validation process based on prototypes generated from Z specifications (of the requirements themselves) is to apply the Test Template Framework (TTF) [32, 4]. The TTF is a model-based testing (MBT) method tailored to the Z notation. It proceeds by dividing the input space of each Z operation (of a given specification) into so-called *test conditions*. In this context we will use, instead, the term *validation condition*. Each validation condition is given by a Z predicate on the input and state variables of the corresponding Z operation. Each of them describes the conditions for a test case. Precisely, we propose to use these conditions for requirement validation (and, obviously, to later test the implementation). In this sense, validating a requirement consists of:

1. Put the prototype in the state described by the validation condition;
2. Execute the implementation of the Z operation with the input values given by the test condition; and
3. Observe the prototype's behavior to determine if it satisfies end-users.

Fastest is a MBT tool that provides tool support for the TTF [4]. It can automatically generate validation conditions by applying so-called testing tactics. A testing tactic is a systematic and general rule to divide the input space of Z operations. The same testing tactics can be applied for requirement validation. For example, a classical testing tactic is Disjunctive Normal Form (DNF), which rewrites the Z operation in DNF, takes the precondition of each disjunct, and divides its input space with these predicates.

For example, the input space of *Transaction* (Figure 1) is:

$$Transaction^{IS} == \\ [clients : UID \pfun NAME;\ accounts : UID \pfun \mathbb{Z};\ n? : UID;\ m? : \mathbb{Z}]$$

So, when DNF is applied to it the following validation conditions are generated:

$$Transaction_1^{DNF} == [Transaction^{IS} \mid n? \in \mathrm{dom}\ accounts \land m? \neq 0] \\ Transaction_2^{DNF} == [Transaction^{IS} \mid n? \notin \mathrm{dom}\ accounts] \\ Transaction_3^{DNF} == [Transaction^{IS} \mid m? = 0]$$

Therefore, if the prototype is used in each of these conditions, end-users will be validating whether requirement engineers correctly understood (at least part of) what it means to record a transaction of a client. For instance, they will try to record a proper transaction ($Transaction_1^{DNF}$), a transaction of a nonexistent client ($Transaction_2^{DNF}$), etc. As suggested by the TTF, more testing tactics can be applied to further divide the validation conditions obtained so far, thus, getting more revealing ones. Therefore, if engineers and users find that some validation condition should be verified more deeply they can apply more testing tactics to this validation condition to further divide it in more complex cases.

In summary, if Fastest is used to generate the validation conditions, and then each of them is successively entered in SET STATE (cf. Sect. 5), software engineers will have a tool that will assist them in generating and validating a prototype (besides making early progress with testing). {*log*} has already proved to solve many of these conditions [7].

7 Conclusions

We have shown that it would be possible to develop a prototype generator for Z specifications based on the {*log*} constraint solver. These prototypes could include rich GUIs that would make end-users to be able to use them for requirement validation. The TTF could assist software engineers in conducting a systematic requirement validation process, and {*log*} could help them in automating some fundamental activities. This technique would also help during the testing stage. The proposal would have a number of advantages that would pay-off the effort of writing a Z specification.

However, many issues need to be discussed and some problems need to be solved. For example, how prototypes would interface with external systems? How this can be described in Z? Should it be described in Z? Should it be prototyped?

What if a validation condition asks for a state that cannot be reached by a sequence of the operations available in the specification? Should this condition be validated or not? What is a precise characterization of the class of systems within the scope of this proposal? Finding an answer to all these questions, however, seems feasible and is left for future work.

References

1. Bouquet, F., Legeard, B., Peureux, F.: CLPS-B - A constraint solver to animate a B specification. STTT 6(2), 143–157 (2004)
2. Bowen, J.P., Hall, J.A. (eds.): Z User Workshop, Cambridge, UK, 29-30 June 1994, Proceedings. Workshops in Computing, Springer/BCS (1994)
3. Breuer, P.T., Bowen, J.P.: Towards correct executable semantics for Z. In: Bowen and Hall [2], pp. 185–209
4. Cristiá, M., Albertengo, P., Frydman, C.S., Plüss, B., Rodríguez Monetti, P.: Tool support for the Test Template Framework. Softw. Test., Verif. Reliab. 24(1), 3–37 (2014)
5. Cristiá, M., Hollmann, D., Albertengo, P., Frydman, C.S., Monetti, P.R.: A language for test case refinement in the Test Template Framework. In: Qin, S., Qiu, Z. (eds.) ICFEM. Lecture Notes in Computer Science, vol. 6991, pp. 601–616. Springer (2011)
6. Cristiá, M., Rossi, G.: Protoype examples for SETS 2014, `https://www.dropbox.com/s/rgub9d3i10coht8/sets2014-examples.tar.gz`
7. Cristiá, M., Rossi, G., Frydman, C.S.: {log} as a test case generator for the Test Template Framework. In: Hierons, R.M., Merayo, M.G., Bravetti, M. (eds.) SEFM. Lecture Notes in Computer Science, vol. 8137, pp. 229–243. Springer (2013)
8. Doma, V., Nicholl, R.A.: EZ: A system for automatic prototyping of Z specifications. In: Prehn, S., Toetenel, W.J. (eds.) VDM Europe (1). Lecture Notes in Computer Science, vol. 551, pp. 189–203. Springer (1991)
9. Dovier, A., Omodeo, E.G., Pontelli, E., Rossi, G.: A language for programming in logic with finite sets. J. Log. Program. 28(1), 1–44 (1996)
10. Dovier, A., Piazza, C., Pontelli, E., Rossi, G.: Sets and constraint logic programming. ACM Trans. Program. Lang. Syst. 22(5), 861–931 (2000)
11. Dovier, A., Piazza, C., Rossi, G.: A uniform approach to constraint-solving for lists, multisets, compact lists, and sets. ACM Trans. Comput. Log. 9(3) (2008)
12. Dovier, A., Pontelli, E., Rossi, G.: Set unification. Theory Pract. Log. Program. 6(6), 645–701 (Nov 2006), `http://dx.doi.org/10.1017/S1471068406002730`
13. Gervet, C.: Conjunto: Constraint propagation over set constraints with finite set domain variables. In: Hentenryck, P.V. (ed.) ICLP. p. 733. MIT Press (1994)
14. Goodman, H.S.: The Z-into-Haskell tool-kit: An illustrative case study. In: Bowen, J.P., Hinchey, M.G. (eds.) ZUM. Lecture Notes in Computer Science, vol. 967, pp. 374–388. Springer (1995)
15. Grieskamp, W.: A computation model for Z based on concurrent constraint resolution. In: Bowen, J.P., Dunne, S., Galloway, A., King, S. (eds.) ZB. Lecture Notes in Computer Science, vol. 1878, pp. 414–432. Springer (2000)
16. Hallerstede, S., Leuschel, M., Plagge, D.: Validation of formal models by refinement animation. Sci. Comput. Program. 78(3), 272–292 (2013)
17. Hasselbring, W.: Animation of Object-Z specifications with a set-oriented prototyping language. In: Bowen and Hall [2], pp. 337–356

18. Hewitt, M.A., O'Halloran, C., Sennett, C.T.: Experiences with PiZA, an animator for Z. In: Bowen, J.P., Hinchey, M.G., Till, D. (eds.) ZUM. Lecture Notes in Computer Science, vol. 1212, pp. 37–51. Springer (1997)
19. Ladenberger, L., Bendisposto, J., Leuschel, M.: Visualising Event-B models with B-Motion Studio. In: Alpuente, M., Cook, B., Joubert, C. (eds.) Formal Methods for Industrial Critical Systems, Lecture Notes in Computer Science, vol. 5825, pp. 202–204. Springer Berlin Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-04570-7_17
20. Leuschel, M., Butler, M.: ProB: A model checker for B. In: Keijiro, A., Gnesi, S., Mandrioli, D. (eds.) FME. Lecture Notes in Computer Science, vol. 2805, pp. 855–874. Springer-Verlag (2003)
21. Munakata, T.: Notes on implementing sets in prolog. Commun. ACM 35(3), 112–120 (Mar 1992), http://doi.acm.org/10.1145/131295.131300
22. Palù, A.D., Dovier, A., Pontelli, E., Rossi, G.: Integrating finite domain constraints and CLP with sets. In: PPDP. pp. 219–229. ACM (2003)
23. Plagge, D., Leuschel, M.: Validating Z specifications using the ProB animator and model checker. In: Davies, J., Gibbons, J. (eds.) Integrated Formal Methods. Lecture Notes in Computer Science, vol. 4591, pp. 480–500. Springer-Verlag (2007)
24. Rossi, F., Beek, P.v., Walsh, T.: Handbook of Constraint Programming (Foundations of Artificial Intelligence). Elsevier Science Inc., New York, NY, USA (2006)
25. Rossi, G.: {*log*} (2008), http://www.math.unipr.it/~gianfr/setlog.Home.html, last access: December 2013
26. Saaltink, M.: The Z/EVES mathematical toolkit version 2.2 for Z/EVES version 1.5. Tech. rep., ORA Canada (1997)
27. Schrage, M.: Never go to a client meeting without a prototype. IEEE Software 21(2), 42–45 (2004)
28. Servat, T.: BRAMA: A new graphic animation tool for B models. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007: Formal Specification and Development in B, Lecture Notes in Computer Science, vol. 4355, pp. 274–276. Springer Berlin Heidelberg (2006), http://dx.doi.org/10.1007/11955757_28
29. Sherrell, L.B., Carver, D.L.: FunZ: An intermediate specification language. Comput. J. 38(3), 193–206 (1995)
30. Spivey, J.M.: The Z notation: a reference manual. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK (1992)
31. Sterling, L., Ciancarini, P., Turnidge, T.: On the animation of "not executable" specifications by Prolog. International Journal of Software Engineering and Knowledge Engineering 6(1), 63–87 (1996)
32. Stocks, P., Carrington, D.: A Framework for Specification-Based Testing. IEEE Transactions on Software Engineering 22(11), 777–793 (Nov 1996)
33. Valentine, S.H.: The programming language Z−. Information & Software Technology 37(5-6), 293–301 (1995)
34. Waeselynck, H., Behnia, S.: B model animation for external verification. In: ICFEM. pp. 36–45 (1998)
35. West, M.M.: The use of a logic programming language in the animation of Z specifications. In: Dahl, V., Niemelä, I. (eds.) ICLP. Lecture Notes in Computer Science, vol. 4670, pp. 451–452. Springer (2007)
36. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. TPLP 12(1-2), 67–96 (2012)
37. Winikoff, M., Dart, P., Kazmierczak, E.: Rapid prototyping using formal specifications. In: In Proceedings of the 21st Australasian Computer Science Conference. pp. 279–294. Springer-Verlag (1998)

Introduction to the Integration of SMT-Solvers in Rodin*

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

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

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

Keywords: Event-B · SMT · Rodin

1 Introduction

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

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

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

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

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

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

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

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

2 SMT Solvers

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

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

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

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

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

3 Event-B

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

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

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

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

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

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

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

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

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

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

4 Translating Event-B to SMT

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

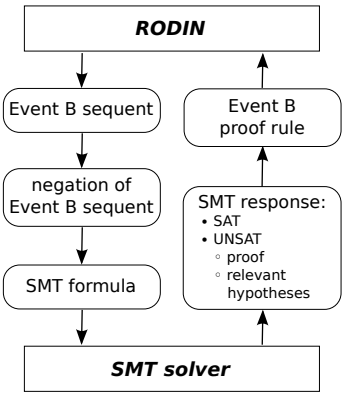


Fig. 1. Schematic view of the interaction between Rodin and SMT solvers.

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

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

The main issue in the translation of proof obligations to SMT-LIB is the representation of the set-theoretic constructs. Different approaches are possible, especially considering the expressiveness of the set operators found in the proof obligation. For instance, the approach presented in [10] encodes sets as predi-

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

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

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

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

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

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

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

The pre-processing behaves as follows:

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

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

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

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

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

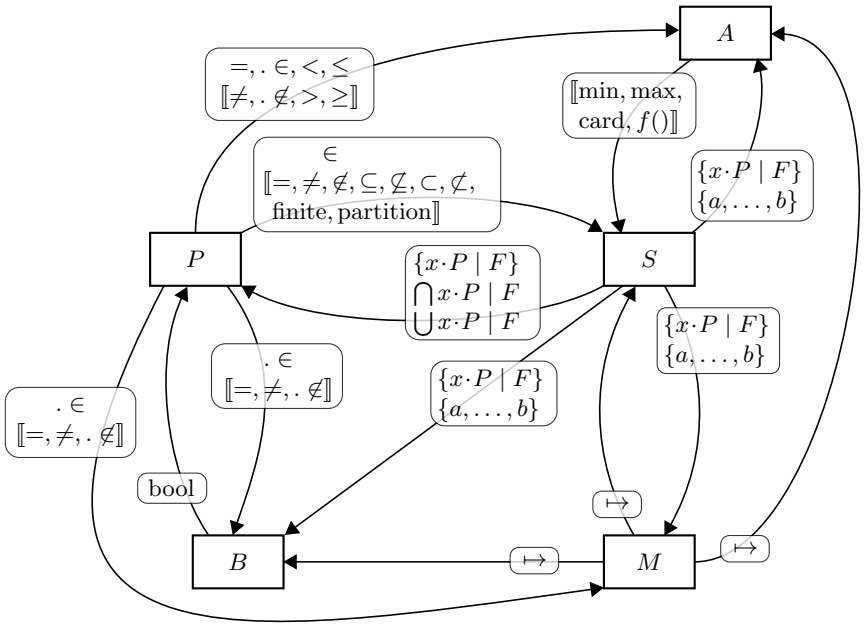


Fig. 3. The different syntactic categories and the symbols relating them: A for arithmetic expressions, P for predicates, S for set expressions, B for Boolean expressions and M for maplet expressions. *ppTrans* removes all occurrences of the constructs delimited by double-brackets.

$$x \neq y \rightsquigarrow \neg(x = y) \quad (2) \qquad x \notin s \rightsquigarrow \neg(x \in s) \quad (4)$$

$$s \subseteq t \rightsquigarrow s \in \mathbb{P}(t) \quad (3) \qquad \text{finite}(s) \rightsquigarrow \forall a \cdot \exists b, f \cdot f \in s \rightarrowtail a..b \quad (5)$$

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

$$x_1 \mapsto x_2 = y_1 \mapsto y_2 \rightsquigarrow x_1 = y_1 \wedge x_2 = y_2 \quad (6) \qquad x = f(y) \rightsquigarrow y \mapsto x \in f \quad (9)$$

$$\text{bool}(P) = \text{bool}(Q) \rightsquigarrow P \Leftrightarrow Q \quad (7) \qquad x = \text{FALSE} \rightsquigarrow \neg(x = \text{TRUE}) \quad (10)$$

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

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

$$n = \text{card}(s) \rightsquigarrow \exists f \cdot f \in s \rightarrowtail\!\!\!\!\!\rightarrow 1..n \quad (11) \qquad a \succ b \rightsquigarrow b \prec a \quad (13)$$

$$n = \max(s) \rightsquigarrow n \in s \wedge \max(s) \leq n \quad (12) \qquad a \prec \max(s) \rightsquigarrow \exists x \cdot x \in s \wedge a \prec x \quad (14)$$

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

$$e \in s \twoheadleftarrow\!\!\!\!\!\leftarrow t \rightsquigarrow e \in s \twoheadleftarrow\!\!\!\!\!\leftarrow t \wedge t \subseteq \text{ran}(e) \quad (15)$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5 Experimental results

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

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

```
1 (set-logic AUFLIA)
2 (declare-sort JOBS 0)
3 (declare-sort PJ 0)
4 (declare-fun MS (JOBS PJ) Bool)
5 (declare-fun active () PJ)
6 (declare-fun j () JOBS)
7 (declare-fun queue () PJ)
8 (assert (! (forall ((x JOBS))
9              (not (and (MS x active) (MS x queue)))) :named inv3))
10 (assert (! (MS j queue) :named grd1))
11 (assert (! (not (forall ((x0 JOBS))
12                    (not (and (or (MS x0 active) (= x0 j))
13                              (MS x0 queue)
14                              (not (= x0 j)))))) :named goal))
15 (check-sat)
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6 Conclusion

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

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

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

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

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

References

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

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

Using Deduction Modulo in Set Theory *

Pierre Halmagrand

Cedric/Cnam/Inria, Paris, France
Pierre.Halmagrand@inria.fr

Abstract. We present some improvements of Zenon Modulo and the application of this tool to sets of problems coming from set theory. Zenon Modulo is an extension of the tableau-based first order automated theorem prover Zenon to deduction modulo. Deduction modulo is an extension of predicate calculus, which allows us to rewrite terms as well as propositions, and which is well-suited for proof-search in axiomatic theories, as it turns axioms into rewrite rules. The improvements discussed here consist in a better heuristic to automatically build rewrite systems given a set of axioms, and some optimizations in the rewriting process used during the proof search. We also present some updated results obtained on benchmarks provided by the TPTP library for set theory categories. Finally, we discuss some recent work about the application of our tool to the B method set theory, in particular the way we treat equality and the comprehension scheme.

Keywords: Automated Theorem Proving, Tableaux, Deduction Modulo, Rewriting, Set Theory, B Method, Zenon Modulo.

1 Introduction

The development of large-scale industrial projects based on formal software is constrained by the efficiency of verification tools that themselves rely on automated theorem provers to mechanize a maximal part of the formalization process. Therefore, to allow a wider dissemination of these techniques, a particular attention must be paid to the development of automated theorem prover. Since many formal developments may be based on specific theories, like the widely used B method [1] which relies on a particular typed set theory, we must pay attention to reason within axiomatic theories. A solution to improve verification tools is to combine different approaches in automated deduction and specialize them for a specific theory, like SMT solvers.

The BWare project [17] aims to provide a generic platform relying on different theorem provers (first order provers and SMT solvers) to verify proof obligations coming from the development of industrial applications using the B method.

* This work is supported by the BWare project [17] (ANR-12-INSE-0010) funded by the INS programme of the French National Research Agency (ANR).

As part of this project, the development of Zenon Modulo [10] is driven by its application to set theory.

Our approach is to extend the tableau-based first order automated theorem prover Zenon [6] to deduction modulo [11]. Deduction modulo is an extension of predicate calculus, which allows to rewrite terms as well as propositions. This is well suited for proof-search in axiomatic theories, as it turns axioms into rewrite rules. For instance, we can express Zermelo set theory without any axiom [12], and turn non-deterministic proof-search among axioms into computations, which reduces the proof-search space. Moreover, Zenon Modulo, like Zenon, adopts a certifying approach and produces proof certificates that can be checked by external proof checkers. Since the proofs produced by Zenon Modulo should keep the advantage of conciseness of deduction modulo, we do not want to record all the computational steps of rewriting [9]. As a result, in order to verify such proofs, we use Dedukti [4], a simple proof checker based on the $\lambda\Pi$-calculus modulo which can deal with rewriting.

We also present an updated heuristic, used by Zenon Modulo to automatically transform axioms into rewrite rules. The new heuristic captures more axioms and guarantees to have a more meaningful rewrite system and impacts both rewrite rules over terms and over propositions. We should notice here that this heuristic has been developed to be used as a preprocessing tool that allows a fully automatic use of Zenon Modulo on problems in the TPTP format. A second option is to give manually the rewrite system to Zenon Modulo. This second option is the one chosen for the B method set theory since there is a limited number of axioms and definitions, and we can afford to design a customized rewrite system.

To assess our approach, we apply this new heuristic to the SET and SEU categories of the first-order problems of the TPTP library [16] which deal with set theory, and compare results obtained by Zenon and Zenon Modulo with both heuristics. In particular, we exhibit some examples of difficult problems that can be proved by our tool using the new heuristic.

Finally, we discuss the application of Zenon Modulo to the B method set theory [1]. We will especially focus on equality and the comprehension scheme in first order logic.

This paper is organized as follows: in Sec. 2, we first introduce the basic notions of deduction modulo and its application to Zenon Modulo; we then present in Sec. 3 the new heuristic for Zenon Modulo; we expose in Sec. 4 the experimental results obtained on the benchmarks provided by the TPTP library; finally, in Sec. 5, we discuss the ongoing work on the B set theory.

2 Zenon with Deduction Modulo

Zenon Modulo [10] is an extension of the tableau-based first order automated theorem prover Zenon [6] to deduction modulo [11]. This extension is partially inspired by the presentation of tableaux modulo in [5]. The proof-search rules

are applied with the usual tableau method: starting from the negation of the goal, apply the rules in a top-down fashion to build a tree. When all branches can be closed by applying a closure rule, the tree itself is said closed, and this closed tree is a proof of the goal.

Deduction modulo [11] extends the predicate calculus by introducing a congruence relation over propositions and the ability to perform conversion between propositions that are equivalent modulo the congruence. Given a set of axioms, the congruence is generated by the rewrite system coming from the transformation of axioms into rewrite rules. One of the major interests of deduction modulo lies in reasoning modulo this congruence, since it reduces the search space by removal of axioms from the context. For example, considering the following definition of inclusion in set theory: $\forall X, Y\ (X \subseteq Y \Leftrightarrow \forall x\ (x \in X \Rightarrow x \in Y))$

The proof of $A \subseteq A$ produced by Zenon has the following shape:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\neg(A \subseteq A),\ \forall X, Y\ (X \subseteq Y \Leftrightarrow \forall x\ (x \in X \Rightarrow x \in Y))}{\forall Y\ (A \subseteq Y \Leftrightarrow \forall x\ (x \in A \Rightarrow x \in Y))}\,\gamma_{\forall\text{inst}}}{A \subseteq A \Leftrightarrow \forall x\ (x \in A \Rightarrow x \in A)}\,\gamma_{\forall\text{inst}}}{\neg\forall x\ (x \in A \Rightarrow x \in A)}\,\beta_{\Leftrightarrow} \quad \dfrac{A \subseteq A}{\odot}\,\odot}{\neg(\tau \in A \Rightarrow \tau \in A)}\,\delta_{\neg\forall}}{\tau \in A,\ \neg(\tau \in A)}\,\alpha_{\neg\Rightarrow}}{\odot}\,\odot$$

where $\tau = \epsilon x.\neg(x \in A \Rightarrow x \in A)$ is a Hilbert epsilon-term formed with a variable x and the formula $\neg(x \in A \Rightarrow x \in A)$, which is the formula existentially quantified over. The Hilbert epsilon-term denotes the term that satisfies the formula, seen as a predicate on x, if it exists [18]. It can be seen as a Skolem term that records the formula itself.

Deduction modulo replaces the axiom by the rewrite rule, where X and Y are variables: $X \subseteq Y \longrightarrow \forall x\ (x \in X \Rightarrow x \in Y)$

The proof produced by Zenon Modulo is then:

$$\dfrac{\dfrac{\dfrac{\neg(A \subseteq A)}{\neg(\tau \in A \Rightarrow \tau \in A)}\,\delta_{\neg\forall},\ A \subseteq A \longrightarrow \forall x\ (x \in A \Rightarrow x \in A)}{\tau \in A,\ \neg(\tau \in A)}\,\alpha_{\neg\Rightarrow}}{\odot}\,\odot$$

where $\tau = \epsilon x.\neg(x \in A \Rightarrow x \in A)$.

It can be seen that computations are interleaved with the deduction rules. It can be noticed that the proof is much simpler than the one produced by Zenon. In addition to simplicity, deduction modulo also allows for unbounded proof size reduction [7].

3 Building Theories Modulo

Dealing with large sets of axioms is quite difficult when building theories modulo. In the TPTP library [16], the first order problems category has almost 7,000,000 axioms and the subset of set theory problems has more than 170,000 axioms. Since we cannot manually build rewrite systems for such large sets of axioms, we develop a heuristic to automatically turn axioms into rewrite rules.

The heuristic described below is based on the following guidelines. We never introduce a free variable that was not bound at the head of the formula by a universal quantifier. For rewrite rules over terms, we want to capture equality properties of terms, except axioms expressing commutativity for obvious termination reasons. We also excluded to rewrite a variable to a more complicated term for the same reason. For rewrite rules over propositions, we select axioms expressing equivalence properties of propositions. The left hand side of the rewrite rule must be an atom, or its negation, as required by deduction modulo [11], and the right hand side may be any proposition. We also handle cases where the atomic proposition is an equality, in this case we require that at least one side of the equality is a function symbol. Moreover, axioms with just an atomic proposition, or its negation, are turned to rewrite to true, and false respectively. Finally, conjunctions of propositions that satisfy one of the previous conditions, are separated into multiple rewrite rules, accordingly to the logical equivalence of $\forall \bar{x}\ \varphi \wedge \psi$ and $\forall \bar{x}\ \varphi \wedge \forall \bar{x}\ \psi$, where φ and ψ are two arbitrary formulas.

In the following, t_1 and t_2 are terms, P is an atomic formula that is not an equation, and φ is an arbitrary formula. $\mathrm{FV}(\varphi)$ refers to the set of free variables of φ. Here are the shapes of axioms that are handled by our heuristic and the corresponding rewrite rule that is generated:

- $\forall \bar{x}\ t_1 = t_2$
  - If $\mathrm{FV}(t_2) \subseteq \mathrm{FV}(t_1)$ and t_1 is not a variable, then the term rewrite rule $t_1 \longrightarrow t_2$ is generated;
  - Otherwise, if $\mathrm{FV}(t_1) \subseteq \mathrm{FV}(t_2)$ and t_2 is not a variable, then the term rewrite rule $t_2 \longrightarrow t_1$ is generated;
  - In addition, all axioms expressing the commutativity of a given symbol are excluded.

- $\forall \bar{x}\ P$ (resp. $\forall \bar{x}\ \neg P$)
  - The proposition rewrite rule $P \longrightarrow \top$ is generated (resp. $P \longrightarrow \bot$).

- $\forall \bar{x}\ P \Leftrightarrow \varphi$ (resp. $\forall \bar{x}\ \neg P \Leftrightarrow \varphi$)
  - If $\mathrm{FV}(\varphi) \subseteq \mathrm{FV}(P)$, then the proposition rewrite rule $P \longrightarrow \varphi$ is generated (resp. $P \longrightarrow \neg\varphi$);
  - Otherwise, if φ is a literal and $\mathrm{FV}(P) \subseteq \mathrm{FV}(\varphi)$, then we apply the heuristic to the formula $\forall \bar{x}\ \varphi \Leftrightarrow P$ (resp. $\forall \bar{x}\ \varphi \Leftrightarrow \neg P$).

- $\forall \bar{x}\ (t_1 = t_2) \Leftrightarrow \varphi$ (resp. $\forall \bar{x}\ \neg(t_1 = t_2) \Leftrightarrow \varphi$)
  - If $\mathrm{FV}(\varphi) \subseteq \mathrm{FV}(t_1) \cup \mathrm{FV}(t_2)$ and at least one of the two terms t_1 and t_2 is not a variable, then the proposition rewrite rule $(t_1 = t_2) \longrightarrow \varphi$ is generated (resp. $(t_1 = t_2) \longrightarrow \neg\varphi$).

The main difference between this new heuristic and the previous one presented in [10] is the last item of the list above. With this new rule, we can transform into rewrite rules axioms which express an equivalence between equality of two terms and a proposition. An example of a proof using such rewrite rule is given in Sec. 4. In addition, we improve the generation of rewrite rules over terms in order to exclude to rewrite variables, and we deal with conjunctions of propositions that satisfy one the conditions listed above.

4 Experimental Results

4.1 TPTP Benchmarks

The TPTP library (v6.0.0) provides a large set of standard benchmark examples for automated theorem proving systems [16]. To assess our approach, we perform experiments over all the first order problems coming from the TPTP library and dealing with set theory. This leads to select the SET and SEU categories that have respectively 462 and 900 problems. We compare Zenon, the old version of Zenon Modulo and the new version of Zenon Modulo which uses the heuristic described in Sec. 3. This experiment was done on an Intel Core i7-4770 3.40GHz computer, with a memory limit of 1GB and a timeout of 300s.

The results are summarized in Tab. 1. This table has three columns: the first one provides the number of problems proved by Zenon for each category, while the other two show the results for the old and new versions of Zenon Modulo. In addition, for both old and new versions of Zenon Modulo, we provide some detailed results with profit and loss between provers.

From the results of Tab. 1, we observe that the new version of Zenon Modulo proves more problems than Zenon and the old version of Zenon Modulo for both SET and SEU categories. The total profit of 5 problems in SET category (resulting from the gain of 12 problems and the loss of 7) may seem low, regarding to the difference of 73 problems between Zenon and the old version of Zenon Modulo, but we have gained some very difficult problems, as shown below. The improvement is more significant than for the SEU category. We notice a net profit of 8 problems, coming from a gain of 26 problems and a loss of 18. We also show an example from the SEU category below.

The verification by Dedukti of proofs produced by Zenon Modulo is almost total. Over the 227 proofs in SET category, 224 are correctly verified. The three missing proofs are due to termination issues of the translation of the proof from Zenon Modulo to Dedukti by the backend of Zenon Modulo. In SEU category, all the 110 proofs are declared well typed by Dedukti.

TPTP Category	Zenon	Zenon Modulo (Old)		Zenon Modulo (New Version)		
	Total	Total	vs. Zenon	Total	vs. Zenon	vs. Old
SET 462 prob.	149	222	+86 -13	227	+91 -13	+12 -7
SEU 900 prob.	96	102	+14 -8	110	+32 -18	+26 -18

Table 1. Experimental Results over the TPTP Library

4.2 Analysis of Two Proofs From Set Theory

The TPTP library provides a ranking system to evaluate problems. This note, between 0 and 1, expresses the percentage of automated theorem provers that are not able to prove the problem. A note of 0 means that the problem is trivial (every prover solves it), and a note of 1 means that all provers fail.

According to the TPTP ranking, Zenon Modulo is able to prove some quite difficult problems. The hardest problem is SET817+4, with a ranking of 0.97 in TPTP v6.0.0 (solved only by Muscadet [15]), and which neither Zenon or the previous version of Zenon Modulo [10] was able to prove. It states that the intersection of all elements of a non-empty set of ordinal numbers is an ordinal. Among the 8 axioms needed by Zenon Modulo to solve this problem, 7 are transformed into rewrite rules by our heuristic. The proof of this problem is too big to be displayed here.

An other example coming from the set theory, and that bodes well for application to the B method set theory, is the problem SEU194+1. This problem, with a ranking of 0.70, and which neither Zenon or the previous version of Zenon Modulo [10] was able to prove, states that for any set s, if p is a relation, then the domain of the restriction of p to s is equal to the intersection of the domain of p and the set s. Among the twenty eight axioms provided with the conjecture, Zenon Modulo needs only two axioms to solve it, of which one is turned into a rewrite rule. In the following, s, t and u are sets, p is a relation, a is an element of a set, rel is a predicate for relations, dom a function that returns the domain of a relation, and rest a function that returns the restriction of a relation.

- Conjecture:
  $\forall s, p\ (\mathsf{rel}(p) \Rightarrow \mathsf{dom}(\mathsf{rest}(p, s)) = \mathsf{dom}(p) \cap s)$
- Axiom:
  $\forall a, s, p\ (\mathsf{rel}(p) \Rightarrow (a \in \mathsf{dom}(\mathsf{rest}(p, s)) \Leftrightarrow (a \in s) \wedge (a \in \mathsf{dom}(p))))$
- Rewrite rule:
  $u = s \cap t \longrightarrow \forall a\ (a \in u \Leftrightarrow ((a \in s) \wedge (a \in t)))$

Zenon Modulo produces the proof of Fig. 1, presented in a format combining deduction steps in solid line and rewriting steps in dashed line. In addition, we omit some unnecessary formulas resulting from the application of $\beta_{\Leftrightarrow}$ rules to lighten the presentation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\neg(\forall s,p\ (\mathsf{rel}(p) \Rightarrow \mathsf{dom}(\mathsf{rest}(p,s)) = \mathsf{dom}(p) \cap s)),}{\forall a,s,p\ (\mathsf{rel}(p) \Rightarrow (a \in \mathsf{dom}(\mathsf{rest}(p,s)) \Leftrightarrow a \in s \wedge a \in \mathsf{dom}(p)))}\,\delta_{\neg\forall}\times 2}{\neg(\mathsf{rel}(\tau_2) \Rightarrow \mathsf{dom}(\mathsf{rest}(\tau_2,\tau_1)) = \mathsf{dom}(\tau_2) \cap \tau_1)}\,\alpha_{\neg\Rightarrow}}{\mathsf{rel}(\tau_2),\ \neg(\mathsf{dom}(\mathsf{rest}(\tau_2,\tau_1)) = \mathsf{dom}(\tau_2) \cap \tau_1)}\ \textit{rewrite}}{\neg(\forall a\ (a \in \mathsf{dom}(\mathsf{rest}(\tau_2,\tau_1)) \Leftrightarrow a \in \mathsf{dom}(\tau_2) \wedge a \in \tau_1))}\,\delta_{\neg\forall}}{\neg(\tau_3 \in \mathsf{dom}(\mathsf{rest}(\tau_2,\tau_1)) \Leftrightarrow \tau_3 \in \mathsf{dom}(\tau_2) \wedge \tau_3 \in \tau_1)}\,\beta_{\neg\Leftrightarrow}$$

$$\dfrac{\Pi_1 \qquad \dfrac{\tau_3 \in \mathsf{dom}(\mathsf{rest}(\tau_2,\tau_1)),\ \neg(\tau_3 \in \mathsf{dom}(\tau_2) \wedge \tau_3 \in \tau_1)}{\Pi_2 \qquad \Pi_3}\,\beta_{\neg\wedge}}{\neg(\tau_3 \in \mathsf{dom}(\mathsf{rest}(\tau_2,\tau_1))),\ \tau_3 \in \mathsf{dom}(\tau_2) \wedge \tau_3 \in \tau_1}\,\alpha_{\wedge}$$

$$\dfrac{\dfrac{\dfrac{\tau_3 \in \mathsf{dom}(\tau_2),\ \tau_3 \in \tau_1}{\mathsf{rel}(\tau_2) \Rightarrow (\tau_3 \in \mathsf{dom}(\mathsf{rest}(\tau_2,\tau_1)) \Leftrightarrow \tau_3 \in \tau_1 \wedge \tau_3 \in \mathsf{dom}(\tau_2))}\,\gamma_{\forall}\times 3}{\dfrac{\neg(\mathsf{rel}(\tau_2))}{\odot}\,\odot \qquad \dfrac{\tau_3 \in \mathsf{dom}(\mathsf{rest}(\tau_2,\tau_1)) \Leftrightarrow \tau_3 \in \tau_1 \wedge \tau_3 \in \mathsf{dom}(\tau_2)}{\dfrac{\neg(\tau_3 \in \tau_1 \wedge \tau_3 \in \mathsf{dom}(\tau_2))}{\dfrac{\neg(\tau_3 \in \tau_1)}{\odot}\,\odot \qquad \dfrac{\neg(\tau_3 \in \mathsf{dom}(\tau_2))}{\odot}\,\odot}\,\beta_{\neg\wedge} \qquad \dfrac{\tau_3 \in \mathsf{dom}(\mathsf{rest}(\tau_2,\tau_1))}{\odot}\,\odot}\,\beta_{\Leftrightarrow}}\,\beta_{\Rightarrow}$$

$$\dfrac{\dfrac{\dfrac{\neg(\tau_3 \in \mathsf{dom}(\tau_2))}{\mathsf{rel}(\tau_2) \Rightarrow (\tau_3 \in \mathsf{dom}(\mathsf{rest}(\tau_2,\tau_1)) \Leftrightarrow \tau_3 \in \tau_1 \wedge \tau_3 \in \mathsf{dom}(\tau_2))}\,\gamma_{\forall}\times 3}{\dfrac{\neg(\mathsf{rel}(\tau_2))}{\odot}\,\odot \qquad \dfrac{\tau_3 \in \mathsf{dom}(\mathsf{rest}(\tau_2,\tau_1)) \Leftrightarrow \tau_3 \in \tau_1 \wedge \tau_3 \in \mathsf{dom}(\tau_2)}{\dfrac{\neg(\tau_3 \in \mathsf{dom}(\mathsf{rest}(\tau_2,\tau_1)))}{\odot}\,\odot \qquad \dfrac{\tau_3 \in \tau_1 \wedge \tau_3 \in \mathsf{dom}(\tau_2)}{\dfrac{\tau_3 \in \tau_1,\ \tau_3 \in \mathsf{dom}(\tau_2)}{\odot}\,\odot}\,\alpha_{\wedge}}\,\beta_{\Leftrightarrow}}\,\beta_{\Rightarrow}$$

$$\dfrac{\dfrac{\dfrac{\neg(\tau_3 \in \tau_1)}{\mathsf{rel}(\tau_2) \Rightarrow (\tau_3 \in \mathsf{dom}(\mathsf{rest}(\tau_2,\tau_1)) \Leftrightarrow \tau_3 \in \tau_1 \wedge \tau_3 \in \mathsf{dom}(\tau_2))}\,\gamma_{\forall}\times 3}{\dfrac{\neg(\mathsf{rel}(\tau_2))}{\odot}\,\odot \qquad \dfrac{\tau_3 \in \mathsf{dom}(\mathsf{rest}(\tau_2,\tau_1)) \Leftrightarrow \tau_3 \in \tau_1 \wedge \tau_3 \in \mathsf{dom}(\tau_2)}{\dfrac{\neg(\tau_3 \in \mathsf{dom}(\mathsf{rest}(\tau_2,\tau_1)))}{\odot}\,\odot \qquad \dfrac{\tau_3 \in \tau_1 \wedge \tau_3 \in \mathsf{dom}(\tau_2)}{\dfrac{\tau_3 \in \tau_1,\ \tau_3 \in \mathsf{dom}(\tau_2)}{\odot}\,\odot}\,\alpha_{\wedge}}\,\beta_{\Leftrightarrow}}\,\beta_{\Rightarrow}$$

where :
$\tau_1 = \epsilon(s).\neg(\mathsf{rel}(p) \Rightarrow \mathsf{dom}(\mathsf{rest}(p,s)) = \mathsf{dom}(p) \cap s)$
$\tau_2 = \epsilon(p).\neg(\mathsf{rel}(p) \Rightarrow \mathsf{dom}(\mathsf{rest}(p,\tau_1)) = \mathsf{dom}(p) \cap \tau_1)$
$\tau_3 = \epsilon(a).\neg(a \in \mathsf{dom}(\mathsf{rest}(\tau_2,\tau_1) \Leftrightarrow a \in \mathsf{dom}(\tau_2) \wedge a \in \tau_1))$

Fig. 1. Proof of Problem SEU194+1

5 Application to the B Method Set Theory

The BWare project [17] aims to provide a generic platform based on Why3 [3] relying on different deduction tools, such as Alt-Ergo [2], iProver Modulo [8], Super Zenon [13] and Zenon Modulo [10], in order to verify proof obligations coming from the development of industrial applications using the B method. Since B proof obligations are translated into the input language of Why3 [14], the B method set theory has been axiomatized in the WhyML language.

Building the B set theory modulo consists mainly in turning six axioms and many derived constructs into rewrite rules. The first two axioms, dealing with membership of an ordered pair in a cartesian product and the membership of a set in the power-set, can be easily turned into rewrite rules. The third axiom, defining the comprehension scheme, is removed due to its high-order definition, we present below how to deal with derived constructs defined with the comprehension scheme. The fourth axiom is the extensionality axiom and states that two sets s and t are equal if being a member of s is equivalent to be a member of t. This is not the only property of the equality symbol, as we will see below. Finally, the last two axioms, the axiom of choice and the existence of an infinite set are easy to deal with.

Following the notations of the B-Book, s and t are sets, E and F some expressions, x a variable, $\mathbb{P}(t)$ the power-set of the set t and `BIG` the constant infinite set. Here is the set of rewrite rules generated from the axioms:

$$\begin{cases} (E,F) \in (s \times t) \longrightarrow (E \in s \wedge F \in t) & \text{(pair)} \\ s \in \mathbb{P}(t) \longrightarrow \forall x\,(x \in s \Rightarrow x \in t) & \text{(power)} \\ s = t \longrightarrow \forall x\,(x \in s \Leftrightarrow x \in t) & \text{(extensionality)} \\ \texttt{choice}(s) \in s \longrightarrow \exists x\,(x \in s) & \text{(choice)} \\ \texttt{infinite}(\texttt{BIG}) \longrightarrow \top & \text{(infinite)} \end{cases}$$

Fig. 2. Expression of the Axioms of the B Set Theory as a Rewrite System

5.1 Removal of the Comprehension Scheme

In the B-book, the comprehension scheme is used to define non-primitive symbols. For instance, the union of two sets is defined as follows:

$$s \cup t \;:=\; \{a \mid a \in u \wedge (a \in s \vee a \in t)\}$$

where u is a set, and s and t and two subsets of u.

Since we have dismissed the comprehension scheme, we expand the above definition by directly defining membership to the union, thereby removing the use of comprehension:

$x \in s \cup t \longrightarrow x \in s \vee x \in t \qquad \text{(union)}$

This rewrite rule, combined with extensionality, is equivalent to the previous definition of the union. Handling the other non-primitive symbols, like intersection of sets, inverse of a relation or also the identity relation, in this systematic way, allows a total removal of the comprehension scheme used to define derived constructs in the B-Book. Unfortunately, this method do not permit us to manage user-defined sets using the comprehension scheme for the moment.

Here is, as an example, the proof produced by Zenon Modulo for the commutativity of union:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\neg\forall A, B\ (A \cup B = B \cup A)}{\neg\forall B\ (\tau_1 \cup B = B \cup \tau_1)}\,\delta_{\neg\forall}}{\tau_1 \cup \tau_2 = \tau_2 \cup \tau_1}\,\delta_{\neg\forall}}{\neg\forall X\ (X \in \tau_1 \cup \tau_2 \Leftrightarrow X \in \tau_2 \cup \tau_1)}\,\text{extensionality}}{\neg(\tau_3 \in \tau_1 \cup \tau_2 \Leftrightarrow \tau_3 \in \tau_2 \cup \tau_1)}\,\delta_{\neg\forall}}{\Pi_1 \qquad \Pi_2}\,\beta_{\neg\Leftrightarrow}$$

where Π_1 and Π_2 are the following subtrees:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Pi_1}{\neg(\tau_3 \in \tau_1 \cup \tau_2),\ \tau_3 \in \tau_2 \cup \tau_1}}{\neg(\tau_3 \in \tau_1 \vee \tau_3 \in \tau_2),\ \tau_3 \in \tau_2 \vee \tau_3 \in \tau_1}\,\text{union} \times 2}{\neg(\tau_3 \in \tau_1),\ \neg(\tau_3 \in \tau_2)}\,\alpha_{\neg\vee}}{\dfrac{\tau_3 \in \tau_2}{\odot}\,\odot \qquad \dfrac{\tau_3 \in \tau_1}{\odot}\,\odot}\,\beta_{\vee}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\Pi_2}{\tau_3 \in \tau_1 \cup \tau_2,\ \neg(\tau_3 \in \tau_2 \cup \tau_1)}}{\tau_3 \in \tau_1 \vee \tau_3 \in \tau_2,\ \neg(\tau_3 \in \tau_2 \vee \tau_3 \in \tau_1)}\,\text{union} \times 2}{\neg(\tau_3 \in \tau_2),\ \neg(\tau_3 \in \tau_1)}\,\alpha_{\neg\vee}}{\dfrac{\tau_3 \in \tau_1}{\odot}\,\odot \qquad \dfrac{\tau_3 \in \tau_2}{\odot}\,\odot}\,\beta_{\vee}$$

and where:

$$\tau_1 = \epsilon A.\neg(A \cup B = B \cup A)$$
$$\tau_2 = \epsilon B.\neg(\tau_1 \cup B = B \cup \tau_1)$$
$$\tau_3 = \epsilon X.\neg(X \in \tau_1 \cup \tau_2 \Leftrightarrow X \in \tau_2 \cup \tau_1)$$

We notice that the subtrees Π_1 and Π_2 are symmetric, the proof of the commutativity of union resulting of the commutativity of the disjunction.

5.2 Dealing with Equality

The set theory of the B-book relies heavily on a primitive notion of equality, introduced before axioms for the set theory [1]. The main property of the equality is substitutivity (*i.e.* an expression can be replaced by another one in a proposition provided they are equal). Starting from substitutivity and reflexivity of equality, other properties like symmetry and transitivity can be derived. Zenon already have rules that deal with this equational reasoning [6].

The second property of equality, introduced much later, and that has to deal directly with set theory is extensionality as shown in Fig. 2. Turning the axiom of extensionality into a rewrite rule allows Zenon Modulo to extend the equality symbol into membership equivalence at each step of the proof-search.

Choosing between equational reasoning and the extensionality properties of equality during proof-search may be decisive to find a proof. There are different solutions to deal with this problem. A first idea is to try both equational reasoning and extensionality every time we meet an equality symbol, but this solution may not be efficient since we duplicate the work for each equality symbol. The solution we are working on is to implement a heuristic into Zenon Modulo that decides to apply the extensionality rewrite rule, or to use the equational reasoning of Zenon, based on the shape of terms. For instance, if both sides of the equality are variables, we do not apply the extensionality rewrite rule, otherwise we use it.

6 Conclusion

We have presented some improvements of Zenon Modulo, in particular a new heuristic to transform axioms into rewrite rules over terms and propositions. This heuristic is used as a preprocessing tool to automatically build theories modulo given sets of axioms. We have also presented results obtained on set theory problems, coming from the benchmarks provided by the TPTP library. This experiment was performed using the new heuristic. In particular, we have shown that this new version proves some new difficult problems according to the TPTP ranking.

We have also discussed some recent work about the application of our tool to the B method set theory. Since we want to use Zenon Modulo to verify proof obligations coming from the development of industrial applications using the B method, we have to build a B set theory modulo. We have described a method to remove the comprehension scheme used to define non-primitive symbols like union of sets. Finally, we have presented some ideas to handle equality in the B set theory.

As future work, we first aim to implement a heuristic into Zenon Modulo that handles the equality of B set theory. In particular, this heuristic should decide whether to apply the extensionality rewrite rule, or to use the equational reasoning of Zenon. To assess our work, we will first try to prove a large part of derived lemmas coming from the B-Book. Finally, we will apply Zenon Modulo to the set of proof obligations provided by the benchmarks of the BWare project.

Acknowledgement. Many thanks to O. Hermant, and D. Doligez for their detailed comments on this paper, and to the Deducteam Inria research team for the many interactions.

References

1. J.-R. Abrial. *The B-Book, Assigning Programs to Meanings*. Cambridge University Press, Cambridge (UK), 1996. ISBN 0521496195.
2. F. Bobot, S. Conchon, E. Contejean, and S. Lescuyer. Implementing Polymorphism in SMT solvers. In C. Barrett and L. de Moura, editors, *SMT 2008: 6th International Workshop on Satisfiability Modulo*, volume 367 of *ACM International Conference Proceedings Series*, pages 1–5, 2008.
3. F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011.
4. M. Boespflug, Q. Carbonneaux, and O. Hermant. The $\lambda\Pi$-Calculus Modulo as a Universal Proof Language. In *Proof Exchange for Theorem Proving (PxTP)*, pages 28–43, Manchester (UK), June 2012.
5. R. Bonichon. TaMeD: A Tableau Method for Deduction Modulo. In *International Joint Conference on Automated Reasoning (IJCAR)*, volume 3097 of *LNCS*, pages 445–459, Cork (Ireland), July 2004. Springer.
6. R. Bonichon, D. Delahaye, and D. Doligez. Zenon: An Extensible Automated Theorem Prover Producing Checkable Proofs. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 4790 of *LNCS/LNAI*, pages 151–165, Yerevan (Armenia), Oct. 2007. Springer.
7. G. Burel. Efficiently Simulating Higher-Order Arithmetic by a First-Order Theory Modulo. *Logical Methods in Computer Science (LMCS)*, 7(1):1–31, Mar. 2011.
8. G. Burel. Experimenting with Deduction Modulo. In *Conference on Automated Deduction (CADE)*, volume 6803 of *LNCS/LNAI*, pages 162–176, Wrocław (Poland), July 2011. Springer.
9. D. Delahaye, D. Doligez, F. Gilbert, P. Halmagrand, and O. Hermant. Proof Certification in Zenon Modulo: When Achilles Uses Deduction Modulo to Outrun the Tortoise with Shorter Steps. In K. McMillan, A. Middeldorp, and A. Voronkov, editors, *International Workshop on the Implementation of Logics (IWIL)*, Stellenbosch (South Africa), Dec. 2013. EasyChair.
10. D. Delahaye, D. Doligez, F. Gilbert, P. Halmagrand, and O. Hermant. Zenon Modulo: When Achilles Outruns the Tortoise using Deduction Modulo. In K. McMillan, A. Middeldorp, and A. Voronkov, editors, *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 8312 of *LNCS/ARCoSS*, pages 274–290, Stellenbosch (South Africa), Dec. 2013. Springer.
11. G. Dowek, T. Hardin, and C. Kirchner. Theorem Proving Modulo. *Journal of Automated Reasoning (JAR)*, 31(1):33–72, Sept. 2003.
12. G. Dowek and A. Miquel. Cut Elimination for Zermelo Set Theory. In *Archive for Mathematical Logic*. Springer. Submitted.
13. M. Jacquel, K. Berkani, D. Delahaye, and C. Dubois. Tableaux Modulo Theories using Superdeduction: An Application to the Verification of B Proof Rules with the Zenon Automated Theorem Prover. In *International Joint Conference on Automated Reasoning (IJCAR)*, volume 7364 of *LNCS*, pages 332–338, Manchester (UK), June 2012. Springer.

14. D. Mentré, C. Marché, J.-C. Filliâtre, and M. Asuka. Discharging proof obligations from Atelier B using multiple automated provers. In S. Reeves and E. Riccobene, editors, *ABZ'2012 - 3rd International Conference on Abstract State Machines, Alloy, B and Z*, volume 7316 of *Lecture Notes in Computer Science*, pages 238–251, Pisa, Italy, June 2012. Springer. `http://hal.inria.fr/hal-00681781/en/`.
15. D. Pastre. Muscadet 2.3: A knowledge-based theorem prover based on natural deduction. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Automated Reasoning*, volume 2083 of *Lecture Notes in Computer Science*, pages 685–689. Springer Berlin Heidelberg, 2001.
16. G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning (JAR)*, 43(4):337–362, Dec. 2009.
17. The BWare Project, 2012. `http://bware.lri.fr/`.
18. C.-P. Wirth. Hilbert's epsilon as an operator of indefinite committed choice. *Journal of Applied Logic*, 6(3):287 – 317, 2008.

Turning Failure into Proof: Evaluating the ProB Disprover

Sebastian Krings, Jens Bendisposto and Michael Leuschel

Institut für Informatik, Universität Düsseldorf**
Universitätsstr. 1, D-40225 Düsseldorf
{krings,bendisposto,leuschel}@cs.uni-duesseldorf.de

Abstract. The ProB disprover uses constraint solving to try and find counter examples to proof obligations. As the ProB kernel is now capable of determining whether a search was exhaustive, one can also use the disprover as a prover. In this paper, we compare the ProB Prover with the standard automatic provers for B and Event-B, such as ml, pp and the Rodin SMT plug-in. We demonstrate that ProB is able to deal with classes of proof obligations that are not easily discharged by other provers. As benchmarks we use medium sized specifications such as solutions to the ABZ 2014 case study, a CAN bus specification and a railway system.

1 Introduction and Motivation

Both the B-method and its successor Event-B [1] are state-based formal methods rooted in set theory. They are used for the formal development of software and systems that are correct by construction. This usually involves formal proofs of different properties of the specification. The proof obligations often include set theoretic theorems and claims. Many provers such as "ml" of Atelier-B are able to discharge certain proof obligations automatically. In former work [13] we already described a disprover based on using ProB's constraint solver to automatically find counter-examples for given proof obligations and thus saving the user from spending time in a futile interactive proof attempt.

We made the observation that in some cases, namely if we never encounter infinite sets nor deferred sets[1] whose cardinality remains unbounded, the absence of a counter example is actually a proof. In [13] we thus suggested as future work to implement an analysis that checks if the absence of a counter example is a valid proof. This work has been finalized in the recent months: ProB now keeps track of infinite set enumeration, in particular the scope in which an infinite enumeration has occurred and whether a solution has been found or not. This enables our technique to detect if the search for a counter-example was exhaustive, i.e., we can now use ProB as a prover.

** Part of this research has been sponsored by the EU funded FP7 project 287563 (ADVANCE).

[1] Deferred sets are sets which are not given upfront by enumerating their elements. They are unbound sets which can become bounded by further constraints.

In [13] we have also identified the need to empirically evaluate the disprover. In this paper we will focus on this research goal: the empirical evaluation of our constraint-based approach to checking proof obligations, in particular when compared to the existing provers available for B.

2 Technique

When working on a proof obligation, Rodin keeps track of two sets of hypotheses: the set of all hypothesis available to proof the target goal and a user-selected subset. The idea behind this is to be able to shrink the search space of automatic provers by omitting hypothesis that should not be used in a proof attempt. In the case of the ProB prover we could, for instance, get rid of hypotheses that are irrelevant for the proof but contain variables over infinite domains, deferred sets or complicated constraints. This approach can not lead to false positives because limiting the number of available hypothesis can not render a formerly unprovable sequent provable.

However, disproving while omitting hypotheses can lead to false negatives if the hypotheses are too weak for a proof. In order not to confuse the user with invalid counter-examples, we only try to disprove a sequent using all hypotheses.

Figure 1 outlines how we proceed:

1. We try to solve the predicate $H_1 \wedge ... \wedge H_m \wedge \neg G$, i.e. the negated goal together with all available hypotheses. If we find a solution, we report a counter-example to Rodin and show it inside the proof tree as shown in Figure 2. If a contradiction is detected, either by analyzing the predicate or by enumerating exhaustively without finding a solution, the initial sequent is proven, because no counter-example exists.
2. If the constraint solver is unable to prove or disprove the predicate, we reduce the number of hypotheses to the user-selected hypotheses. After reducing to a subset of the hypotheses we will not report counter examples to avoid false negatives as discussed:
   - A contradiction detected with the reduced set of hypotheses is still a valid proof as reducing the number of hypotheses might introduce further counter-examples but not remove them.
   - If we find a solution, we report a possible counter-example. However, we do not prevent a following proof effort.
   - Otherwise we return without a result.

The ProB constraint solver supports sets in different ways. First of all, all set theoretic features of the B language are available to formulate constraints. This includes, among others, subset, strict subset, membership, union and intersection as well as cardinality of sets.

The solver is based on constraint-propagation and resorts to enumeration if no further propagation is possible. While doing so, the solver tracks where and why the elements of a set have to be enumerated. It is able to distinguish

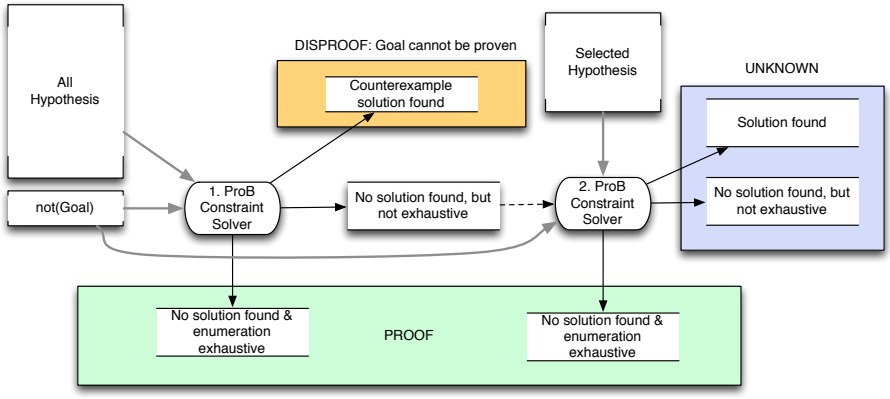


Fig. 1. Disproving Algorithm

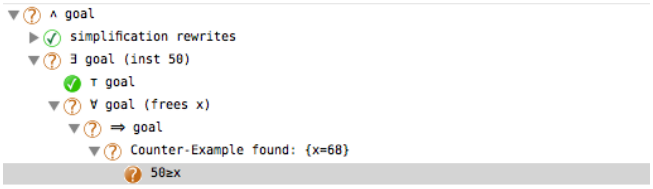


Fig. 2. Counter-Example inside the Rodin Proof Tree

between safe and unsafe enumerations, i.e. if all possible values of a variable have to be tried out or if a single solution is sufficient. This is done by observing the context[2] in which an enumeration occurs. Exhaustive enumeration can then be detected individually for each variable and later be transferred to the whole constraint if possible. Let us look at a few examples, where we suppose all free variables to be existentially quantified:

- $i \in \{1, 2, 1024, 2048\} \wedge i > 2 \ \wedge \ \neg(i \ mod \ 2 \neq 0)$:
  ProB finds two solutions ($i = 1024$ and $i = 2048$) and no infinite enumeration has occurred as ProB has narrowed down the interval of i to 3..2048 before enumeration has started. As such, we can conclude that $G = i \ mod \ 2 \neq 0$ is *not* a logical consequence of the hypotheses $H_1 = i \in \{1, 2, 1024, 2048\}$ and $H_2 = i > 2$. The same solutions could be found by a CLP(FD) query.[3]
- $i > 20 \ \wedge \ \neg(i \ mod \ 2 \neq 0)$:
  ProB finds a solution ($i = 22$), but infinite enumeration has occurred in

[2] This includes quantification, negation and arbitrarily nested combinations of them.
[3] SICStus Prolog: `list_to_fdset([1,2,1024,2048],FDSet), I in_set FDSet, I #>2, I mod 2 #\= 0 #<=> 0, labeling([],[I]).`

the sense that the possible values of i lie in the interval $22..\infty$. However, in this context this is not an issue, as a solution has been found. As such, we can conclude that $i \ mod \ 2 \neq 0$ is not a logical consequence of $i > 20$. This time there is no CLP(FD) query that returns a solution. As there is no finite domain attached to i, labeling can not be performed. In contrast, ProB is able to (partially) enumerate the infinite domain of i in order to find a solution.

- $i \in \{1, 2, 1024, 2048\} \wedge i > 2 \ \wedge \ \neg(i \ mod \ 2 = 0)$:
  ProB finds no solution and no infinite enumeration has occurred. As such, we have proven that $i \ mod \ 2 = 0$ follows logically from $i \in \{1, 2, 1024, 2048\} \wedge i > 2$. A CLP(FD) query also confirms, that there is no solution.
- $i > 20 \ \wedge \ \neg(i \ mod \ 2 = 0 \vee i \ mod \ 1001 \neq 800)$:
  Here ProB finds no solution, but an "enumeration warning" is produced. Indeed, the constraint solver has narrowed down the possible solutions for i to the interval $801..\infty$, but with the default search settings no solution has been found. Here, we cannot conclude that $i \ mod \ 2 = 0 \vee i \ mod \ 1001 \neq 800$ is a logical consequence of $i > 20$. Indeed, $i = 1801$ is a counter example. [4] Again, CLP(FD) is unable to solve the query due to the infinite domain of i.

As mentioned in the introduction, we will not go into further technical details in this paper.

3 Empirical Evaluation and Comparison

For our empirical evaluation we compare ProB to several other provers available for the Rodin platform [2], i.e., Rodin's automatic tactic and the SMT plug-in [9,10]. Our comparison shows the benefit gained from using ProB as a prover. Each additional obligation that is discharged in this comparison actually saves time and money.

3.1 Experimental Setup

- The automatic tactic applies a number of rewriting rules and decision procedures to the proof tree. For instance, a decision procedure checks if the goal is listed in the set of hypotheses and thus discharged. It also uses the PP and ML provers from AtelierB. The automatic tactic is applied until a fixpoint is reached.
- The SMT plug-in [9,10] applies two different SMT solvers (veriT [8] and CVC3 [5]) to the original goal, after some pre-processing.
- The disprover tactic applies three trivial decision procedures (check if goal is $\top$, check if the hypotheses contain $\bot$ and if the goal appears in the list of hypotheses). Afterwards the disprover is applied to the goal.

[4] Which ProB can find if you enlarge the default search space, e.g., by adding $i < 10000$ as additional constraint.

For our experiments, we have used Rodin 2.8, version 2.0.1 of the Atelier B provers plugin and version 1.1.0.e126305 of the SMT Solvers Plugin, with the bundled version 2.4.1 of CVC3 and the bundled development version of veriT. We have used a timeout of 1 second for each SMT solver, run in succession. ProB was used in version 1.3.7-beta10, connected through the disprover plugin version 2.4.4.201403152244. Again, a timeout of 1 second was used for each constraint solving attempt with a maximum of two attempts per proof obligation (see Figure 1). Both the CLP(FD) and the CHR-based solvers of ProB were activated. All benchmarks were run on a MacBook Pro featuring a 2,6 GHz Intel Core i7 CPU and 8 GB 1600 MHz DDR3 memory. The CPU includes 4 cores, yet we ran at most two proof attempts at once. We used a plugin[5] for the Rodin platform that applies the user- or pre-defined proof tactics to selected proof obligations.

As models for our benchmarks we used the following models:

- Answers to the ABZ-2014 case study [7]. The case study models a landing gear system. Beside our own version [12], we also used the three models by Su and Abrial [17], a model by André, Attiogbé and Lanoix [3] as well as a model by Mammar and Laleau [14].
- A model of the Stuttgart 21 Railway station interlocking by Wiegard, derived from the interlocking model in chapter 17 of [1] with added timing and performance modeling.
- A model of a controller area network (CAN) bus. A CAN Bus is used in vehicles for direct communication between components without a central processor. The model was developed by Colley.
- A formal development of a graph coloring algorithm by Andriamiarina and Méry. The graphs to be colored are finite, but unbounded and not fixed in the model.
- A model of a pacemaker by Neeraj Kumar Singh [15].
- A model formalizing a number of set theoretical laws; generated for regression tests.

3.2 Results and Analysis

The results of the benchmarks are shown in Table 1 and Figures 4 and 5. Table 1 shows the total number of proof obligations discharged, as well as a column showing the percentage of proof obligations discharged using ML/PP together with SMT and in the last column the percentage discharged by using these two proof tactics together with the ProB disprover. Each Venn diagram shows how many proof obligations are discharged by which prover. Except for the graph coloring algorithm and the set laws example ProB performs surprisingly well.

The graph coloring algorithm uses unbounded sets, that means that some of the proof obligations cannot be proven using constraint solving and enumeration.

[5] The source code of the plugin can be found at `https://github.com/wysiib/ProverEvaluationPlugin`. An update site for installation inside Rodin is available at `http://nightly.cobra.cs.uni-duesseldorf.de/rodin_provereval/`.

Table 1. Benchmark results: proof obligations discharged for various developments

Model	# POs	ML/PP	SMT	ProB	% excl. ProB	% incl. ProB
Landing Gear System 1, Su, et. al.	2328	2171	2312	2275	99.57	99.79
Landing Gear System 2, Su, et. al.	1188	845	1140	1165	97.22	99.49
Landing Gear System 3, Su, et. al.	341	201	187	251	74.78	85.63
CAN Bus, Colley	542	501	490	320	95.02	95.2
Graph Coloring, Andriamiarina, et. al.	254	226	116	19	96.06	96.06
Landing Gear System, Hansen, et. al.	74	72	63	74	100	100
Landing Gear System, Mammar, et. al.	433	347	385	334	95.15	98.15
Landing Gear System, Andre, et. al.	619	466	400	459	79.81	90.63
Pacemaker, Neeraj Kumar Singh	370	360	358	351	98.38	100
Stuttgart 21 interlocking, Wiegard	202	57	94	184	55.45	93.56
Set laws, Leuschel	67	67	67	62	100	100

ProB is only able to prove some very trivial invariants, such as $\forall n \cdot n \in S \Leftrightarrow n \in dom(R)$ for the initialization $S := \emptyset \parallel R := \emptyset$. The other unfavorable case, the set laws, is very similar. The model contains five invariants that contain infinite sets and cannot be proven using ProB.

As can be seen in the last two columns of Table 1, ProB improves the results of automatic proving in all other developments. In some cases, such as the cases shown in Figure 5(b), 4(c) and 4(e) the improvement is rather big. The reason for the big improvement is that these models only use enumerated sets and integers. In these cases ProB can produce elaborate case distinctions, combined with constraint solving to narrow down the search space. This type of proof is not supported by the classical provers ML and PP. Generally, the proof obligations that pose problems to the ProB disprover are well-definedness proof obligations.

It is also interesting to note that, on their own, the ML and PP provers do not fare quite so well as in Table 1: they require specific pre-processing to be effective. In Table 2 are the results for two models without any pre-processing (except for collecting hypotheses using the lasso tool):

Table 2. Results without pre-processing by Rodin

Model	# POs	Provers			SMT	ProB
		ML	PP	ML/PP		
Landing Gear System, Mammar et al	433	284	127	284	385	341
Landing Gear System, Andre et al	619	560	81	567	400	511
Pacemaker, Neeraj Kumar Singh	370	344	187	352	328	350

As can be seen, for the first model ML on its own discharges just 284 (45.9 %) proof obligations. PP discharges just 127 (20.5 %) of the proof obligations. The SMT solver also benefits considerably from pre-processing: without it, it discharges "just" 385 (62.2 %) of the proof obligations. The third model shows

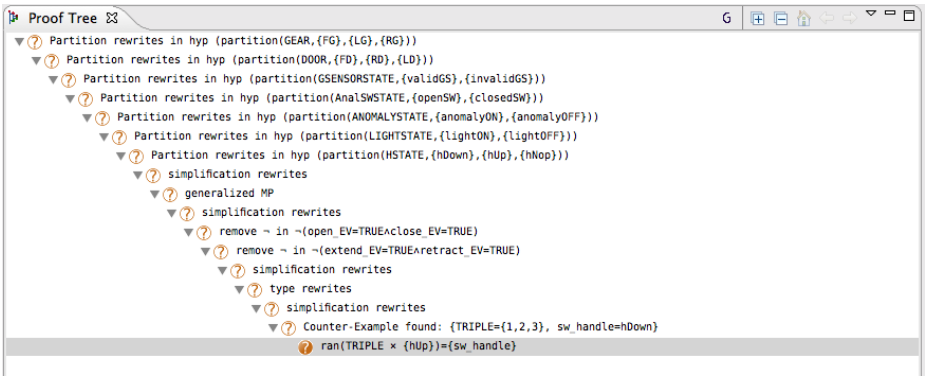


Fig. 3. Counter-Example for proof obligation of Landing Gear System by Andre et al.

quite similar declines if pre-processing is omitted. However, the second model shows the opposite behavior: without pre-processing, more proof obligations can be discharged. This is due to the timeouts leaving less time for the actual prover, if we include a pre-processing phase. In future, we want to examine whether better pre-processing can improve the performance of the ProB disprover.

The first landing gear system by Andre et al. contains unprovable proof obligations, where the disprover finds counter examples (e.g., the proof obligation `cockP_handleUp/onHand/INV` in the model `LandingSysDP_SWITCH_A`). This is very useful feedback to the developer of the model, and the initial purpose of the ProB disprover. Figure 3 shows the counter-example inside the Rodin proof tree.

Finally, for the Landing Gear System by Mammar et al., the developers had trouble discharging a few proof obligations using the other provers (including the SMT plugin). The ProB disprover was able to discharge them; one of the proof obligations was a well-definedness proof obligation.

4 Discussion and Conclusion

A secondary motivation for the experiments conducted in this paper was the empirical evaluation of our constraint solver, more precisely its capability to detect inconsistencies (a successful proof with the disprover requires finding an inconsistency without enumerating unbounded variables; see Fig. 1). Finding inconsistencies is important for detecting disabled events during animation, and more importantly for constraint-based validation, such as constraint-based deadlock checking [11]: it avoids the constraint solver exploring unsuccessful alternatives. In the context of model-based testing, it enables one to detect uncoverable alternatives, and not spending time trying to find test cases to cover them.

One important issue is the soundness of the ProB disprover. In [6] we have presented the various measures we are taking to validate ProB's results. In

addition, we have developed a SMT-LIB [4] importer for ProB and have applied our disprover to a large number of SMT-LIB benchmarks, checking that no "false theorems" are proven. For this paper, we have also double checked many of the proof obligations which were only provable by ProB, to ensure that they are indeed provable. As the Venn diagrams in Figures 4 and 5 show, a large number of proof obligations can be proven by two or even three different provers. As the three provers rely on completely different technologies and have been developed by independent teams, we can have a very high confidence that those proof obligations are indeed provable.

We have demonstrated that ProB is capable to discharge proof obligations that currently cannot be proven using Rodin's auto tactic and the SMT solvers. Our prover typically deals well with a different kind of proof obligations than the other provers, and is thus an orthogonal extension rather than a replacement. Rodin's auto tactic performs well in the realm of set theoretic constructs and relational expressions, some of which cannot be easily represented in the SMT syntax. SMT on the other hand performs well on arithmetic expressions, where the auto tactics often fail. ProB finally covers predicates over enumerated sets, explicit data and explicit computations and has a good support for integer arithmetic over finite domains.

However, for models which make heavy use of deferred sets, such as the graph colouring algorithm model (see Table 1), the ProB disprover can currently mainly play its role as disprover. More precisely, for any proof obligation which involves deferred sets and where no precise value of the cardinality of the deferred set is known, the disprover can only return either a counter example or the result "unknown". In future, we plan to improve the treatment of deferred sets in ProB, and to have the constraint solver determine the cardinalities of those sets while solving. This should also enable the disprover to act as a prover for more proof obligations involving deferred sets.

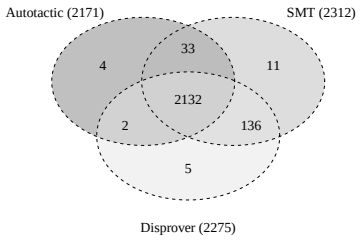
We think that the ProB Disprover is a valuable extension to Rodin's set of provers, because it can increase the number of proof obligations that are automatically discharged, thus saving time and money. Overall, the outcome of the empirical evaluation was a positive surprise, as ProB's main domain of application is finding concrete counter examples, not discharging proof obligations. In particular, the fact that the number of discharged proof obligations, for the models under consideration in Table 1, is comparable to that of the SMT plugin with its two SMT solvers was unexpected. In future, we also plan to use our SAT backend [16] for the ProB disprover, and evaluate its performance.

Acknowledgements We would like to thank the various developers for giving us access to their Event-B models, and for discussions and feedback: Jean-Raymond Abrial, Andre, Attiogbe, John Colley, Régine Laleau, Lanoix, Amel Mammar, Dominique Méry, Neeraj Kumar Singh, Wen Su.

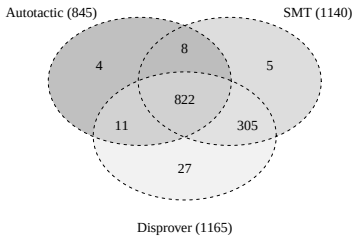
References

1. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.

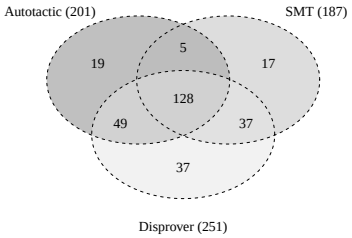
2. J.-R. Abrial, M. Butler, and S. Hallerstede. An open extensible tool environment for Event-B. In Z. Liu and J. He, editors, *Proceedings ICFEM'06*, LNCS 4260, pages 588–605. Springer-Verlag, 2006.
3. André, Attiogbé, and Lanoix. Modelling and Analysing the Landing Gear System: a Solution with Event-B/Rodin. `http://www.lina.sciences.univ-nantes.fr/aelos/softwares/LGS-ABZ2014/index.php`. Solution to ABZ-2014, Accessed: 2014-03-17.
4. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.
5. C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.
6. J. Bendisposto, S. Krings, and M. Leuschel. Who watches the watchers: Validating the prob validation tool. In *Proceedings of the 1st Workshop on Formal-IDE*, EPTCS XYZ, 2014. Electronic Proceedings in Theoretical Computer Science, 2014.
7. Boniol and Wiels. Landing gear system. `http://www.irit.fr/ABZ2014/landing_system.pdf`. Accessed: 2014-03-17.
8. T. Bouton, D. C. B. de Oliveira, D. Déharbe, and P. Fontaine. verit: an open, trustable and efficient smt-solver. In R. A. Schmidt, editor, *Proc. Conference on Automated Deduction (CADE)*, Lecture Notes in Computer Science, pages 151–156. Springer-Verlag, 2009.
9. D. Déharbe. Automatic Verification for a Class of Proof Obligations with SMT-Solvers. In M. Frappier, U. Glässer, S. Khurshid, R. Laleau, and S. Reeves, editors, *Proceedings ASM 2010*, LNCS 5977, pages 217–230. Springer, 2010.
10. D. Deharbe, P. Fontaine, Y. Guyot, and L. Voisin. SMT solvers for Rodin. In *Proceedings ABZ'2012*, LNCS 7316, pages 194–207. Springer, 2012.
11. S. Hallerstede and M. Leuschel. Constraint-based deadlock checking of high-level specifications. *TPLP*, 11(4–5):767–782, 2011.
12. Hansen, Ladenberger, Wiegard, Bendisposto, and Leuschel. Validation of the ABZ Landing Gear System using ProB. `http://www.stups.uni-duesseldorf.de/ProB/index.php5/ABZ14`. Solution to ABZ-2014 case study, Accessed: 2014-03-17.
13. O. Ligot, J. Bendisposto, and M. Leuschel. Debugging Event-B Models using the ProB Disprover Plug-in. *Proceedings AFADL'07*, Juni 2007.
14. Mammar and Laleau. Modeling a Landing Gear System in Event-B. `http://www-public.it-sudparis.eu/~mammar_a/LandingGearsSystem.html`. Solution to the ABZ-2014 Case Study, Accessed: 2014-03-17.
15. D. Méry and N. K. Singh. Formal specification of medical systems by proof-based refinement. *ACM Trans. Embed. Comput. Syst.*, 12(1):15:1–15:25, Jan. 2013.
16. D. Plagge and M. Leuschel. Validating B, Z and TLA+ using ProB and Kodkod. In D. Giannakopoulou and D. Méry, editors, *Proceedings FM'2012*, LNCS 7436, pages 372–386. Springer, 2012.
17. Su and Abrial. Aircraft Landing Gear System: Approaches with Event-B to the Modeling of an Industrial System. `http://www.lab205.org/home/#!/case-landing`. Solution to the ABZ-2014 Case Study.



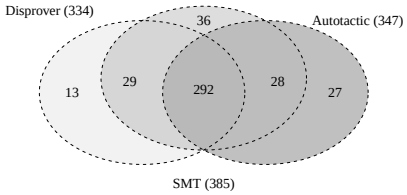
(a) Su and Abrial, version 1



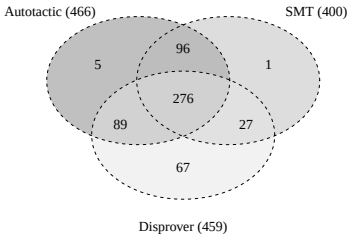
(b) Su and Abrial, version 2



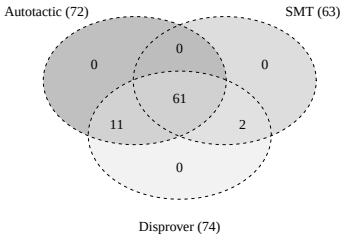
(c) Su and Abrial, version 3



(d) Mammar and Laleau

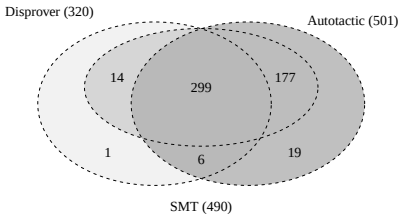


(e) André, Attiogbé and Lanoix

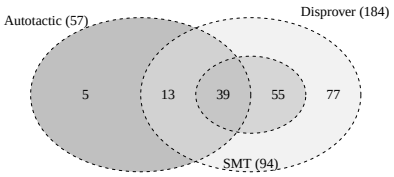


(f) Hansen, Ladenberger, Wiegard, Bendisposto and Leuschel

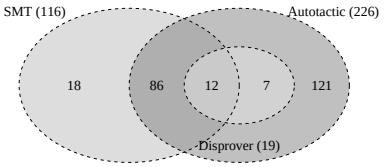
Fig. 4. Visualization of the benchmark results. Part 1: Landing gear system



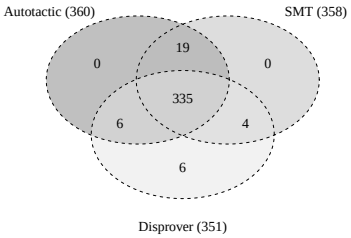
(a) Colley, CAN Bus



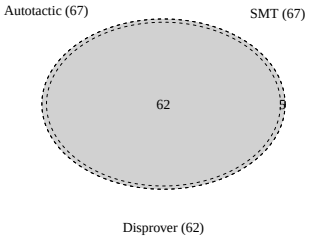
(b) Wiegard, Stuttgart 21



(c) Andriamiarina and Mèry, Graph Coloring Algorithm



(d) Singh, Pacemaker



(e) Leuschel, Set Laws

Fig. 5. Visualization of the benchmark results. Part 2: Miscellaneous models

Return of Experience on Automating Refinement in B

Thierry Lecomte[1]

[1] ClearSy,
Aix en Provence, France
Thierry.lecomte@clearsy.com

Abstract. Refining a B specification into an implementation can be a complex and time consuming process. This process can usually be separated in two distinct parts: the specification part, where refinement is used to introduce new properties and specification details, and the implementation, where refinement is used to convert a detailed B specification into a B0 implementation. This article presents experience on the development and use of a refiner tool that automates the production of implementable models, in a number of industrial applications.

Keywords: B formal method, refinement, deployment, industry

1 Introduction

Historically, the B Method [1] was introduced in the late 80's to design correct by construction, safe software. B and Atelier B[1], the tool implementing it, have been successfully applied to the industry of transportation with the development of large size embedded software [2], [3], [4], [10] and to a lesser extent to other application domains [5], [7], [11], [12], [13], [15], [16]. Based on the notion of refinement, the B method allows writing software specification and implementation by using the same formal language. Specification is progressively transformed (refined) into implementation by adding algorithmic details and/or transforming abstract data types into concrete (implementable) ones. If the specification is the result of a purely human activity, the refinement process could be automated as it consists in a sequence of transformations applied to the specification in order to obtain an implementable model.

In this article, we expose our return of experience on applying automatic refinement in industry, through a number of industrial applications. In section 2, an overview of refinement in B is given. Automatic refinement key concepts and tooling are presented in section 3. Section 4 exposes industrial applications of automatic refinement. Finally section 5 concludes and discusses further work.

[1] *http://www.atelierb.eu/*

2 Refining in B

The B method is mainly aimed at software development. B models are structured in components: a component is usually[2] made of a specification model and an implementation model (the implementation being a refinement of the specification).

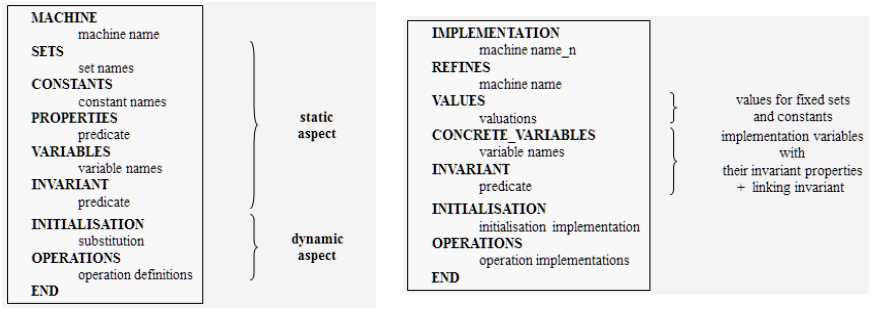


Figure 1: structure of B specification and implementation

An implementation may use other specification models by calling their operations (related models are imported) or by having a read access to the value of variables and constants (related models are seen). A B model is made of variables, constants and operations that modify the variables[3]. Variables and constants have types and constraints linking some variables and constants can be expressed (called invariant for variables and properties for constants). Variables and constants can be either abstract or concrete. Operations are defined with substitutions [8] that can either be deterministic or non-deterministic. Depending of the nature of the model, substitutions used to describe operations are restricted: for example, sequencing is forbidden in specifications, as well as loops, and non-determinism is forbidden in implementations.

Major restrictions on B modelling are:

- [***R1***] The obligation to have a concrete variable implemented in one and only one implementation (it couldn't be modified at two different places). This constraint has a major impact on the structure of B projects: dataflow has to be considered first before defining the components.
- [***R2***] The imports graph must be a tree (see Figure 2): each concrete module except the tree root must be imported in the project to insure that the properties proved locally still hold at global level.

[2] Other cases are: (1) Basic machines have no B implementation as they are used to connect B models with third party software (2) Several successive refinements might be necessary to obtain implementation.

[3] Model initialisation is also considered as an operation. However it can be executed only once and variables properties have to be set after this execution.

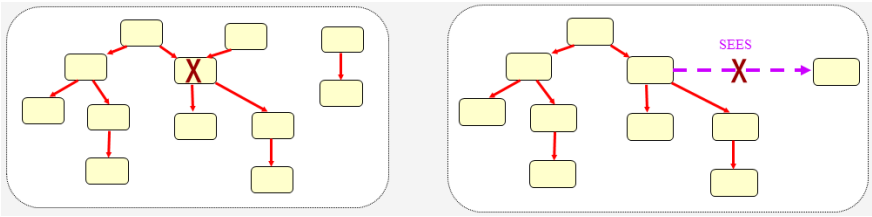


Figure 2: incorrect imports graphs

- [***R3***] The dependency graph must not have any cycle. In this case, it would not have any valid order of initialisation.

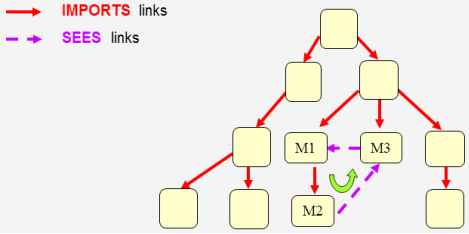


Figure 3: incorrect dependency graph

- [***R4***] An operation defined in the specification of a component can't be used in the implementation of this component. It has to be called in another implementation that imports this component. This constraint requires adding extra-layers in the B project.

On the other hand, this modular decomposition based on specification importation allows proving the models more easily as it breaks down proof complexity and factorizes the proof of operations. Models are also considered as easier to read and to understand.

As an example, let us consider the example shown in Figure 4. The specification (machine M) declares a variable *setv*, typed as a subset of 1..10. This variable is initialized as empty. Obviously this variable can't be implemented as it is. Sets are often refined by using a table (a function) that associates to each element of the domain (here the interval 1..10) a Boolean value indicating if the element belongs to the set. In this example, the new variable *bitv* is declared as a total function from the interval 1..10 to BOOL.

The gluing invariant

$$setv = bitv^{-1}[\{TRUE\}]$$

establishes a direct relation between the variable *setv* (that doesn't exist anymore in refinement M_r) and the variable *bitv*: all elements for which *bitv* is valued as TRUE belong to *setv*. Initialisation has also to comply with this gluing invariant: *setv* being initialised as empty, *bitv* has to be initialised with all elements associated to FALSE.

```
MACHINE
    M
ABSTRACT_VARIABLES
    setv
INVARIANT
    setv ⊆ 1..10
INITIALISATION
    setv := ∅
…
END
```

```
REFINEMENT
    M_r
REFINES
    M
ABSTRACT_VARIABLES
    bitv
INVARIANT
    bitv ∈ 1..10 → BOOL ∧
    setv = bitv⁻¹[{TRUE}]
INITIALISATION
    bitv := (1..10) × {FALSE}
…
END
```

Figure 4: example of specification and refinement

3 B Automatic Refinement

Rationale. Automatic refinement has been initially imagined [6], developed and put into existence by Matra Transport with the automated "Canarsie line" metro in New-York [9]. By abstract model (see Figure 5), we do not mean the collection of all specification models of the project but a selection of related top level components, including specification, refinement and implementation models, that capture the specification of the software. This specification is abstract but contains all details: it just needs to be transformed into an implementable model.

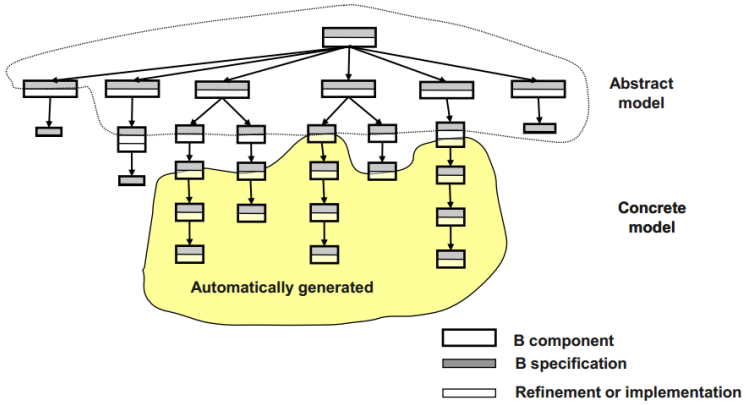


Figure 5: separation between abstract model and concrete model

Automatic refinement is aimed at automatically generating the concrete model (or part of it) from the abstract model, with the objective to dramatically reduce development costs. Indeed, safety critical software usually requires twice the workload because of additional testing, verification and validation. Automatic

refinement is aimed at reducing the workload by up to half as the concrete model is automatically generated and the model is easier to validate. Finally safety critical software would only require the budget of a non-safety critical software development to complete.

Concepts. The main concept is the automatic building of the concrete model by applying refinement patterns step by step, the refinement patterns being expressed as refinement rules. The data (variables) are refined first. The substitutions are then refined.
Abstract model is usually constructed with abstract data typed as set, partial function, relation, etc. As B0 model only allows concrete data type as scalar, total function, etc., data refinement consists in replacing abstract data with more concrete one : this needs to introduce a gluing invariant describing how abstract variable becomes concrete. Data refinement can be done in several steps: not directly from abstract to concrete but with intermediate data representation. Data can already be concrete but be refined in order, for instance, to reduce the memory footprint. In principle, the choice of a refinement for a variable is done regarding its initialisation and operations manipulating this variable. In practice, the choice is done regarding the type of the variable. However types for concrete variables are limited, B0 initialisation is limited too. To automate the data refinement, refinement patterns are given describing the constraints that have to be fulfilled by refined data, the introduced refining variable, and gluing invariant. Then the automatic refiner chooses a refinement pattern regarding the properties of the variables. This choice can be modified by the user who can add new refinement pattern into the tool.

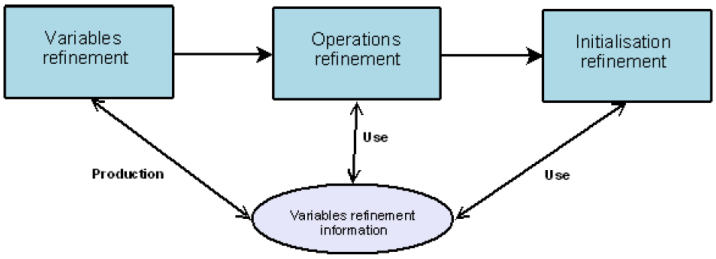


Figure 6: refinement process order

For substitution refinement, treatments on abstract data have to be refined to become translatable treatment on concrete data. B0 allowed substitutions are : assignment, IF, CASE, and WHILE. Similarly to data refinement, substitution refinement is based on refinement patterns that should make explicit under which constraints they can be applied, what are the resulting substitutions, and either it is an implementation or it should be refined one more time.
The refinement of a component consists in the 3 successive actions (see Figure 6): determining refinement pattern for variables of a component, applying refinement patterns for all the operations of a component, and producing the resulting B component. When a variable is refined, related information is stored (variable

refinement information) in order to be reused later on and to speed up the process (variable refinement is elaborated once for all). The recursive application of these 3 steps should lead to a B project fully implemented, if the refinement patterns can be applied at each step of the process. If it is not true, the refinement process stops and some interaction with the user are required to modify existing refinement patterns and/or add new ones.

Tools. In 1997, Matra Transport International (now Siemens Transportation Systems) developed internally a tool called edithB that was used for the development of the Automatic Train Protection System of Canersie Line in New York. This tool was developed in Ada.
In 2008, with Siemens agreement, ClearSy developed a similar tool, called BART (B Automatic Refinement Tool), in order to allow the Community to benefit from automatic refinement tools. The development of the tool has been partly supported by grant No ANR-06-SETI-015-03 awarded by "Agence Nationale de la Recherche" during the R&D project RIMEL (Incremental refinement of event models). In 2009, BART[4] is integrated to the first open-source version of Atelier B (4.0). BART is made more generic than EdithB, in particular the support of the vital coded-monoprocessor is removed. This tool is developed in C++/Qt.

Rule	Operation call	Generated operation
IMPORTED_OPERATION(name => add out => (@a), body => (@a := @e))	aa <-- add1(bb, cc)	out <-- add1(in1,in2) = BEGIN out <-- in1 + in2 END

Figure 7: example of an IMPORTED_OPERATION substitution

For both tools, refinement patterns are expressed using a rich language [9] that allows to specify guards/constraints like DECL_OPERATION (that requires for an operation to be defined), to tune the refinement process and to use advanced substitutions like TYPE_ITERATION (that generates while loops when iterating over a type) or IMPORTED_OPERATION (that generates operation definition and operation call when a treatment needs to be exported to another machine).

```
RULE assign_a_b_6
REFINES
    @a := @b
WHEN
    B0(@a) &
    DECL_OPERATION(@c <-- @d | BEGIN @c := @b END)
IMPLEMENTATION
    @a <-- @d
IMPLEMENT
    @a
END;
```

Figure 8: example of a rule using DECL_OPERATION. The valuation of @a by @b is replaced by the call of an operation (named @d) that returns @b.

[4] https://sourceforge.net/projects/bartrefiner/

These advanced substitutions allow mimicking human refinement by defining precisely how to refine variables and substitutions. These advanced substitutions are replaced by regular substitutions during the automatic refinement process.
The refinement engine manages a stack which contains information about the project, and how variables are refined. This stack also contains tags generated by the refinement rules when applied, these tags have no prior semantics but could be used by refinement rules to give a particular direction to the refinement process.

Metrics. Detailed metrics on edithB are not publically available. Concerning BART, the complete software is made of 264 kloc decomposed as follows:

- Core tool: 89 kloc.
- GUI: 12 kloc
- B Compiler : 154 kloc
- Bwidgets : 7 kloc.

The core tool is made of the refining engine (the biggest module – 7 kloc), the rules parser, the splitter, the namer and the pattern matcher. The complete development was completed in 10 months.

The tool was tested and validated using regular test bench. Indeed, even if the tool is directly contributing to the building of SIL4[5] software, the tool is not expected to be "correct-by-construction" – in case of design error, the generated model should be not provable. In this case, the Atelier B prover is our safety belt.

4 Industrial Applications

Several applications have been developed with edithB and BART.

EdithB has been used for an automatic train protection software (embedded software in charge of stopping a metro in case it is not able to comply with speed limit) in New York [10] and for a wayside control unit (in charge of avoiding train collision) in Roissy airport [2]. If we compare the figures from a manual development [3], [4] and from an automatic refinement development, for the same kind of application, it is apparent that automatic refinement is generating more lines of B models and target code (see Table 1). Meteor and Canarsie being similar applications, automatic refinement is able to double the size of the modelling. There are several reasons for that: every part of the concrete model is usually broken down into many intermediate levels, which produce many lines of B (hence many lines of target code), and code is not shared through the refinement process (for example, generic elements share a similar code, however the code is duplicated at each use). On the proof obligations side, Meteor, Canarsie and Val Roissy generate respectively around 27 000, 82 000 and 43 000 proof obligations. Even if it is difficult to compare proof just from these figures, we can notice automatic refinement generates more proof obligations. Most of them are located in the abstract model, demonstrating the effectiveness of automatic refinement. Moreover, while 1 400 mathematical rules

[5] Safety Integrity Level 4 (highest level)

were required to prove Meteor proof obligations, only 290 rules were required for Val Roissy (together with 61 generic demonstrations, 97% proof obligations were automatically demonstrated).

Table 1. Number of lines of B models per project.

Project	Abstract model	Concrete manual		Concrete automatic
Meteor	<-	115 000	->	None
Canarsie line	125 000		38 000	110 000
VAL Roissy	38 000		27 000	117 000

BART has been used for the development of several SIL4 T2[6] tools, mainly to avoid the development of two redundant software (B allows to develop single SIL4 proven software, non-proven software requires to be developed twice by two separate teams with distinct technologies). Figures similar to edithB have been measured from these developments:

- manual modelling represents one third of the total while automatic refinement is two third. Around 500 refinement rules have been added in every project.
- abstract modelling represents two third of development + proof cost while concrete modelling is one third. However 1 800 mathematical rules were added to the prover (compared to the 290 for Val Roissy). This can be explained by the fact that Atelier B main prover mathematical rules database has been enriched to cover the same kind of modelling (ATP software), so the new kind of modelling brought by these T2 tools requires to slightly bootstrap the prover.
- generated source code has twice the size of handwritten source code because of the many operation calls. Because of constraint [***R4***], every refinement step requires to add an extra-layer of imports, hence the number of lines.

Generated models keep track of the initial substitution and of the refinement rules used, with comments inserted in the model (see listing below).

```
Load_component_2(pcd, pid, pctd2) =
  VAR
    l_1
  IN
    l_1 :(l_1 : INT);
    /* Rule : specific.r14_exit */
    /*? l_1 := perm_component_cpt ?*/
    /* Rule : default.default */
    l_1 <-- Load_component_2_1 ;
    test_max_component_def(l_1) ;
```

[6] According to EN50128 standard, tools are categorized in three : T3 for tools directly contributing to source code, T2 for tools performing verification, T1 for all other tools (editors for example).

```
        /*? perm_component_r( bijection_component_def (perm_component_cpt
+ 1)) := pid ?*/
        /* Rule : default.default */
        Load_component_2_2(pid) ;
        …
```

Refinement rules are mainly gathered in a single file (rmf) shared in the whole project. This file has to be set up right at the beginning of the project by identifying the abstract data types that need to be taken into account and how they are implemented. This identification requires performing several interactive refinement sessions (see Figure 9). Then automatically refining the project is initiated, the components being refined in parallel by different users. New refinement rule is added in the shared rmf file if the rule is generic (reusable in another operation / component) or added to a rmf file specific otherwise. Shared rmf file is composed of:

- 400 rules for operations
- 45 rules for structure
- 30 rules for data
- 30 rules for initialisation

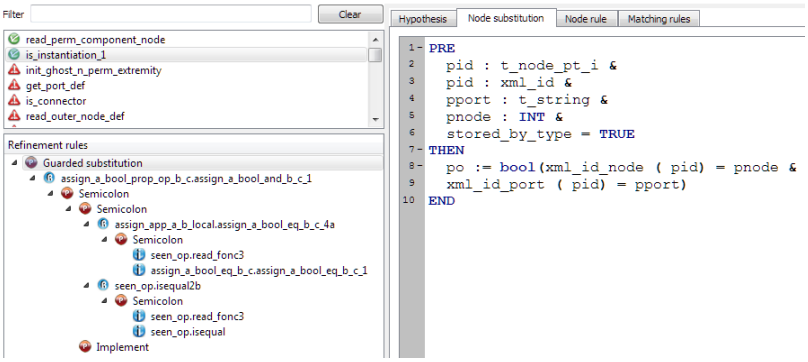


Figure 9: example of interactive refinement session. The refinement tree is displayed on bottom left

5 Return of Experience on Using Set Theory Modelling

The approach presented in this paper is aimed at developing abstract deterministic software specification, using B as a high level language, and leaving most of implementation details to a mechanical, semi-automated process. The specification has to be written using set theory and first order logic, in a way to avoid introducing programmatic gimmicks too early.

For example, in the case of the block controller in [2], blocks are either occupied by a train or free. This information should be safely computed by the functional module "Block Logic". It is formalized by the abstract variable occupied_blocks:

- Either as a subset of t_block (1): a block belongs to occupied_blocks if and only if it is considered to be occupied.
- Or as a total function from t_block to BOOL (2), associating to a block the value TRUE when the block is occupied and FALSE when it is not.

(1) $occupied\_blocks \subseteq t\_block$ (2) $occupied\_blocks \in t\_block \rightarrow \mathrm{BOOL}$

Actually this second choice suits less the proposed B Method, since it is more an encoding of the former (i.e., the use of a set characteristic function instead of the set directly). In this case using a subset of blocks is more abstract. Expressions, predicates and substitutions concerning this set are also more abstract and closest from informal specification. We prefer to use properties over sets when specifying instead of describing the intricate loops that are required to iterate over tables.
We also use relations, partial functions and total functions to type abstract data. For example, the abstract constant ctx_next_block_up associates to a block its next upward block. A block has at most one next block located in the upward direction. A terminal block in the upward direction has no next upward block. So next_block_up is a partial function from t_block to t_block. Finally property states are added to the model, such as this one (linking blocks states and sensors):

$$\begin{array}{l}\forall block \cdot (block \in t\_block \wedge \\ \quad ((ctx\_block\_bs\_up[\{block\}] \cup ctx\_block\_bs\_down[\{block\}]) \cap cut\_beam\_sensors \neq \varnothing \vee \\ \quad ctx\_block\_detector[\{block\}] \subseteq occupied\_block\_detectors) \\ \quad \Rightarrow \\ \quad block \in occupied\_blocks)\end{array}$$

With such predicate, we are able to capture a property in a compact form that is quite straightforward to check. Up to 16 properties have to be formalized in the abstract model. Although it seems to be a small number, some properties had to be cut into many actual properties. At the end, the size of the static and dynamic properties in the main abstract machine is more than 1,000 line long. With the mathematical proof, we do not have to verify the consistancy of these 1,000 lines but just to check that they correctly refine a unique property that is easier to check against the informal specification. In the case of xml engineering tools, xml models are represented as lists, lists of lists, etc. Data types are more elaborated with the notion of node pointers and references to be able to navigate the xml trees. Abstract iterators operate on (node) lists, so many of the existing refinement rules are still valid in this context.
However the development of software aimed at a very different domain induced important modelling effort:

- New data types and structures imply to completely revisit data type related refinement rules
- Existing structural refinement rules have to be specialized in order to take into account these new data types and structures

From this experience, we may consider that the first application of automatic refinement to a new domain is not expected to be economically sound as some effort is required to axiomatize refinement for this domain. However further applications

allow to reuse most of existing rules and to specialize only when required, for particular algorithms. Similarly automatic refinement was initially expected to help newcomers to apply refinement more efficiently. It appears that the profile required to efficiently use automatic refinement demands so many skills that finally only proficient practitioners and experts are able to deal with it. Indeed writing refinement rules could be seen as abstracting the refinement process and requires a strong habit of it.

5 Conclusion and perspectives

Automatic refinement is likely to improve productivity by automating tasks, leading to simpler proofs and simplifying the reuse of known refinement patterns. Similar types and treatments are always refined the same way. This process is not limited to a single type of application, as it has been applied to embedded software as well as to more classical applications (symbolic computation, SysML model analyser / transformer), while obtaining the same metrics (modelling, refinement and proof effort). The process is scalable.

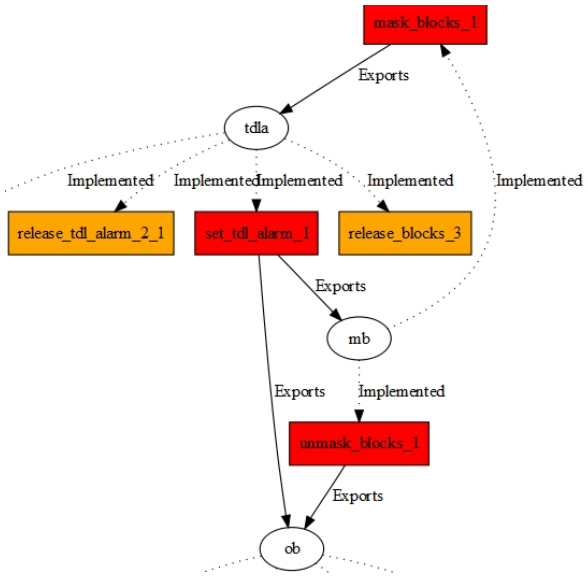


Figure 10: conflict depicted to the user

On the other hand, even if it was our objective when we decided to develop BART, an automatic refiner is not a tool for beginners as it requires understanding why the automatic refinement process stops and how to modify existing refinement rules. Having some experience in refining in B is mandatory to get the most out of it.

Constraint [***R1***] is posing a problem to existing refinement algorithms: in case of conflicting rules, a single variable could be implemented in different components at the same time that is rejected by Atelier B project checker. We are currently working on improving these algorithms in order to not apply conflicting rules and to provide some guidance to the user (interaction with user still remains to be decided). Current feedback is provided as an implementation graph exhibiting *exports* and *implemented* constraints. In Figure 10, we can clearly see a cycle (the upward arrow from variable *mb* to component *mask_blocks_1*). The art of the automatic refinement is today to understand from this graph how to modify refinement rules in order to solve these conflicts.

We are also investigating the possibility to "prove" refinement rules in order to consider that related proof obligations are "by construction" already proven. We are using an approach similar to the Atelier B main prover: the inference engine is based on transformation rules that have been formally proven, hence the successive application of these rules to solve a predicate is considered correct and the predicate is considered true.

References

1. Abrial, J.R. (1996) , *The B-book: Assigning programs to meanings,* Cambridge University Press
2. Amelot, A. & al (2005), *Using B as a High Level Programming Language in an Industrial Project: Roissy VAL*, ZB 2005
3. Behm, P. & al (1998), *METEOR: an Industrial Success in Formal Development*, B'98
4. Behm, P. & al (1999), METEOR: a Successful Application of B in a Large Project, FM'99
5. Benveniste, M. & al (2009), *A Proved "Correct by Construction" Realistic Digital Circuit*, RIAB, FMWeek 2009
6. Burdy, L.(1996) *, Automatic Refinement. In Proceedings of BUGM* at FM'99
7. Casset, L.*,(1999) A formal specification of the Java byte code verifier using the B method,* Lisbonne 99
8. ClearSy (2014), *Atelier B: B language Reference Manual*
9. ClearSy (2014), *BART: User Manual*
10. Essamé, D. & al (2007), *B in Large-Scale Projects: the Canarsie Line CBTC Experience*, B 2007
11. Hoffmann, S. & al (2007), *The B Method for the Construction of Micro-Kernel Based Systems*, ZB 2007
12. Lecomte, T. (2008), *Safe and Reliable Metro Platform Screen Doors Control/Command Systems*, FM 2008
13. Lecomte, T. & al (2007), *Formal Methods in Safety Critical Railway Systems*, SBMF 2007
14. Requet, A. & al (2008), *BART: A Tool for Automatic Refinement*, ABZ 2008
15. Sabatier, D. & al (2006), *Use of the Formal B Method for a SIL3 System Landing Door Commands for line 13 of the Paris subway*, Lambda Mu 15
16. Sabatier, D. & al (2008), *FDIR Strategy Validation with the B method*, DASIA 2008

Programming with Partially Specified Collections

Gianfranco Rossi

Dipartimento di Matematica e Informatica
Università degli Studi di Parma, Parma, Italy
gianfranco.rossi@unipr.it

Abstract. Enumerated collections (e.g. lists, sets, etc.) provided by programming languages are usually defined by completely and precisely enumerating all their constituent elements. Conversely, in (constraint) logic programming languages it is common to deal with *partially specified collections* where either some elements or some parts of the collection are left unknown. In this paper we claim that partially specified collections (in particular, lists and sets) can be conveniently exploited in a wider setting, even in the context of more conventional languages using a library-based approach. We prove this claim by showing a number of simple examples using Java and the Java library JSetL.

1 Introduction

Programming languages usually provide several kinds of collections, e.g. array, string, lists, sets, multi-sets, maps, etc, either as *primitive* abstractions or programmed by exploiting the data abstraction facilities of the language itself and made available to users as standard libraries. For example, sets are a primitive data abstraction in Pascal, SETL and Python, while they are provided as libraries in C++ and Java.

In this paper we will restrict our attention to *enumerated collections* only, that is collections designated by explicitly enumerating its constituting elements. Hereafter the term *collection* implicitly refers to *enumerated* collections.

In imperative programming languages collections need to be always *completely specified* whenever we operate on them. This means that all elements of a collection must be provided and they must have known values. For example, using the interface `Set` of `java.util`, the set $\{1, 3, 0\}$ can be constructed by executing the statements

```
s.add(0); s.add(3); s.add(1);
```

where `s` is an object denoting the empty set which can be defined as

```
Set s = new HashSet();
```

and `HashSet` is one of the implementations of the interface `Set` provided by `java.util`. Note that the requirement to be completely specified does not prevent a collection from containing variables, but whenever the collection needs to be evaluated all variables possibly occurring in it must be replaced by their actual values.

In contrast, collections in declarative programming languages, such as lists in Prolog, are allowed to contain unknown elements and to be only partially specified, and, nevertheless, they can be accessed and manipulated. For instance, the Prolog predicate `[a,b,c] = [X,Y|R]` compares the (completely specified) list of three elements, `a`, `b`, and `c`, with the partially specified list containing two unknown elements `X` and `Y` and an unknown remainder part `R` (i.e., any list containing at least two elements).

Usefulness of partially specified collections has been amply demonstrated in the context of declarative programming languages. In particular, programming with partially specified lists is a well-assessed technique in standard logic programming languages. Partially specified sets, instead, are supplied by extended logic programming languages with sets, such as [6, 7, 5], and are shown to be a powerful data abstraction. Usefulness of partially specified sets is advocated also in specific application areas such as deductive databases, computational linguistics, and knowledge representation.

Aim of this paper is to show that partially specified collections can be conveniently exploited also in a more conventional setting, namely an imperative O-O programming language. In particular, we consider a *library-based approach* where the new data abstractions are defined by using the language data abstraction mechanisms, without requiring any extension of the host language.

We prove our claim by presenting a number of simple examples written in Java using JSetL [19], a general-purpose library that supports partially specified lists and sets, within the O-O language Java. JSetL integrates the notions of logical (or constrained) variable, (set) unification and constraints that are typical of constraint logic programming languages into the Java language. In particular, it allows logical variables to occur in list and set data structures and provides operations to deal with such objects.

The presented examples aim at showing, in particular, how partially specified lists and sets can be exploited for:

- supporting a *declarative programming style*
- providing data modelling tools (in particular sets and set unification) which prove to be particularly suitable for problems involving non-determinism
- supporting constraint programming.

It is important to note that our main concern here is on providing modelling and programming support. Conversely we are only relatively little interested in the efficiency issue (e.g., efficiency of the constraint solver). Furthermore, our focus is on *programming* rather than on constraint solving or satisfiability checking of formulas involving partially specified collections.

Though our discussion is limited to lists and sets, similar considerations could apply to other kinds of collections as well. Moreover, though we are focusing on Java, the same considerations could be easily exported to other O-O languages (such as C++) where these data abstractions could be supplied by libraries, while they can serve as useful guidelines to devise extensions to programming languages where these new data abstractions could be provided as primitive ones.

The paper is organized as follows. Section 2 introduces the notion of partially specified collection, showing how logical variables can be used within collections to represent unknown elements or part of the collection. In Section 3 we start considering operations on partially specified collections, specifically (set) unification. In Section 4 we show how simple inequality, membership and integer comparison constraints over elements of partially specified collections can be used to describe and compute solutions for a number of common programming problems. Section 5 takes into account also set variables, whose domains are collections of sets, and it briefly shows how set variables and set constraints can be advantageously exploited in conjunction with partially specified sets. Sections 6 and 7 are devoted to further remarks and to draw some conclusion.[1]

2 Partially Specified Collections

Declarative programming languages (e.g., functional and logic programming languages) are based on a notion of variable, often called *logical* (or *mathematical variables*, *constrained*) *variable*, that differs from that of imperative programming languages. Logical variables represent unknowns, not memory cells. As such they have no modifiable value stored in them, as ordinary programming language variables have. Conversely, one can associate values to logical variables through relations (or *constraints*) involving logical variables and values from some specific domains.

When the domain of a (logical) variable is restricted to a single value then the variable is said to be *bound* (or *instantiated*) to this value. Otherwise, the variable is *unbound*. An equality relation, in particular, may restrict the domain of a logical variable to a single value. For example, if x is a logical variable ranging over the domain of integers, the equality $x = 3$ forces x to be bound to the value 3. However, the same result could be obtained through other relations, e.g., $x < 4 \wedge x > 2$. The value of a logical variable is immutable. That is it can not be changed, e.g. by an assignment statement like in imperative languages.

Logical variables are found also in conventional programming languages that support *constraint programming* (e.g., Alma-0 [1]), as well as in libraries for constraint programming, such as ILOG [13] and Choco [3] (see also the recent Java Specification Request for a standard Java Constraint Programming API [14]).

In this paper we will refer to what is provided by JSetL [19], a Java library that supports declarative (constraint) programming in an O-O framework. In JSetL a (generic) logical variable is an instance of the class `LVar`. Basically, `LVar` objects can be manipulated through constraints, namely equality (`eq`), inequality (`neq`), membership (`in`) and not membership (`nin`) constraints. Moreover the library provides methods to test whether a variable is bound or not, to get the value of a bound variable (but not to modify it), to get its external name, and so on. Values associated with generic logical variables can be of any type. For

[1] An extended version of this paper appeared in Computer Languages, Systems & Structures, Elsevier, vol. 37/4, 178-192, 2011

some specific domains, however, JSetL offers specializations of the `LVar` data type, which provide further specific constraints. In particular, for the domain of integers, JSetL offers the class `IntLVar`, which extends `LVar` with a number of new methods and constraints specific for integers. In particular, `IntLVar` provides integer comparison constraints such as $<$, $\leq$, etc.

Other important specializations of logical variables are the class `LCollection` and its derived subclasses, `LSet` (for Logical Sets) and `LList` (for Logical Lists). Values associated with `LSet` (`LList`) are objects of the `java.util` class `Set` (`List`). A number of constraints are provided to work with `LSet` (`LList`), which extend those provided by `LVar`. In particular, `LSet` provides equality and inequality constraints that account for the semantic properties of sets (namely, irrelevance of order and duplication of elements); moreover it provides constraints for many of the standard set-theoretical operations, such as union, intersection, set difference, and so on.

Example 1. Logical lists/sets in JSetL.

```
LVar x = new LVar();          // an unbound logical variable
LVar y = new LVar("y",1);     // a bound logical variable
                              // with external name "y" and value 1
LList ll = new LList();       // an unbound logical list
LSet ls_1 = new LSet();       // an unbound logical set
```

A collection where either some elements, or part of the collection itself, are unbound logical variables represents a *partially specified collection*. For example (using a Prolog-like notion), the list `[1,X,Y]`, where `X` and `Y` are unbound logical variables, denotes a partially specified list containing one known element `1` and two unknown elements, denoted `X` and `Y`. As another example, the list `[1,2|Z]`, where `Z` is an unbound logical list (i.e., a logical variable ranging over the domain of lists), designates an *open list* containing two known elements 1 and 2 and an unknown remainder part denoted by `Z`.

A partially specified set (list) can be created in JSetL by adding possibly unknown elements, i.e., unbound `LVar` objects, to either a known or an unknown set (list) using the element insertion operator `ins`.

Example 2. The two partially specified lists `[1,X,Y]` and `[1,2|Z]` can be defined in JSetL as follows:

```
LVar X = new LVar();
LVar Y = new LVar();
LList cl =                                          // the closed p.s. list
    LList.empty().ins(Y).ins(X).ins(1);             // [1,X,Y]
```

and

```
LList Z = new LList();                              // the open p.s. list
LList ol = Z.ins(2).ins(1);                         // [1,2 |Z]
```

JSetL provides a number of facilities, mainly in the form of constraints, to access and manipulate partially specified lists/sets (e.g., unification). Conversely,

most of the non-CLP languages/libraries mentioned above allow (possibly unbound) logical variables to occur in conventional data structures, such as vectors, but they do not provide any specific operation to deal with them as partially specified collections. Even languages/libraries that provide set constraints [11], such as ILOG-Solver [13], Choco [3], and Mozart-Oz [15], do consider only completely specified sets.

3 (Set) Unification

One of the basic operation to access and manipulate partially specified collections in logic programming languages is unification. JSetL provides both standard unification, over `LList` objects, and set unification, over `LSet` objects. Both forms of unification are implemented by the equality method `eq`. The meaning of `o1.eq(o2)` is the unification between the objects `o1` and `o2`. `o1` is either a simple logical variable (i.e., an instance of `LVar`) or a possibly partially specified data collection (i.e., an instance of `LList` or `LSet`). `o2` is either an object of the same type as `o1` or an admissible value for it (i.e., a `Set` object, an `Integer` object, and so on). The following is an example of unification over lists in JSetL.

Example 3. Check whether list `l3` is the concatenation of lists `l1` and `l2`. The problem can be modelled as a (list) unification problem. If the first list `l1` is the empty list, the other two lists must be unifiable (and vice versa). Otherwise, the given lists must satisfy the constraint (using a Prolog-like abstract notation) $\texttt{l1} = [\texttt{x} \mid \texttt{l1new}] \ \wedge \ \texttt{l3} = [\texttt{x} \mid \texttt{l3new}]$, where `l3new` is the concatenation of the "shorter" list `l1new` and the given list `l2`, and equality = represents unification.

```
public static boolean concat(LList l1, LList l2, LList l3) {
    if ( solver.check(l1.eq(Lst.empty()).and(l2.eq(l3))) )
        return true;
    else {
        LVar x = new LVar();
        LList l1new = new LList();
        LList l3new = new LList();
        return
            solver.check(l1.eq(l1new.ins(x)).and(l3.eq(l3new.ins(x))))
            && concat(l1new,l2,l3new);
    }
}
```

The first call to the method `check` tests satisfiability of the constraint $\texttt{l1} = [] \ \wedge \ \texttt{l2} = \texttt{l3}$. `solver` is an object of the class `SolverClass` which is assumed to be created outside the method `concat`. Solving an equality calls into play unification: if the two lists are unifiable, then `check` returns `true`, otherwise it returns `false`. If the constraint $\texttt{l1} = [] \ \wedge \ \texttt{l2} = \texttt{l3}$ is `false`, then the constraint $\texttt{l1} = [\texttt{x} \mid \texttt{l1new}] \ \wedge \ \texttt{l3} = [\texttt{x} \mid \texttt{l3new}]$ is checked and the result is combined with the result of the recursive call to `concat`.

This example shows how the use of a partially specified collection and unification allows us to devise a truly declarative solution to the given problem. Another advantage of using unification—in place of assignment and standard programming variables—is that the same methods can be used both to assign values to variables and to test known values. In particular, unification over partially specified lists can be used both to access single elements of a list and to construct the list itself. For example, the method `concat` can be used either to test if a list `l3` is the concatenation of two given lists `l1` and `l2` or to get one list given the other two. No assumption is made about which are input and which are output parameters. As an example, if `l3` is the JSetL `LList` for `['d','a','t','a','.','t','x','t']`, `l2` is the JSetL `LList` for `['.','t','x','t']`, and `l1` is an unbound `LList`, then the invocation `concat(l1,l2,l3)` binds `l1` to the LList `['d','a','t','a']`.

Set unification differs from standard unification in that the former must account for the properties of sets, namely that order and repetition in a set do not matter. Thus, for example, the two set unification problems, $\{a\} = \{a, a\}$ and $\{a, b\} = \{b, a\}$ have a solution, whereas they would have no solution using standard unification. A general survey of the problem of unification in presence of sets, across different set representations and different admissible classes of set terms, can be found in [8]. The following is a simple example using partially specified sets and set unification.

Example 4. (Coloring of a map) Given a map of n regions and a set of m colors find an assignment of colors to regions such that neighboring regions have different colors. To solve this problem we assume to represent the map as a set whose elements are themselves sets containing two neighboring regions and to represent each region as a distinct unbound logical variable. Hence, the map is represented by a partially specified set. For example,

$$\{\{r_1, r_2\}, \{r_2, r_3\}, \{r_3, r_4\}, \{r_4, r_1\}\}$$

where r_1, r_2, r_3, r_4 are unbound logical variables, is a map of four regions. With these assumptions, the coloring problem can be modelled as the problem of unifying the set representing the map with the set of all admissible unordered pairs of colors that can be constructed from the given m colors. For example, given the map considered above and the set of colors $\{1, 2, 3\}$, the problem to be solved turns out to be (using the usual Prolog-like abstract notation):

$$\{\{r_1, r_2\}, \{r_2, r_3\}, \{r_3, r_4\}, \{r_4, r_1\} \mid R\} = \{\{1, 2\}, \{2, 3\}, \{1, 3\}\}$$

where R represents the set of viable color pairs that possibly have not been used in the computed solution. One of the many possible solutions to this problem is:

$$r_1 = 1, r_2 = 2, r_3 = 1, r_4 = 3, R = \{\{2, 3\}\}.$$

Using JSetL, the proposed modelling of the coloring problem can be easily implemented in Java by the following method:

```
public static void coloring(LSet map, LSet colorPairs) {
  LSet R = new LSet();
  solver.check(colorPairs.eq(R.insAll(map)));
}
```

where `R.insAll(map))` is the set containing all elements of the set `map` and an unknown part `R`, and `colorPairs` is the set of all admissible unordered pairs of colors. This set can be constructed "by hand" or it can be computed starting from the set of colors by exploiting set unification in a way similar to the one shown in Example 5.

The coloring example shows that the use of partially specified sets and set unification allows us to devise very concise and declarative—though possibly inefficient—solutions.[2]

A key feature of unification over partially specified sets is the *non-determinism* embedded in the set unification algorithm. As a matter of fact, a set unification problem may have more than one (independent) solution (non-uniqueness of mgu's). For example, $\{X, Y\} = \{a, b\}$, where X and Y are unbound variables, has *two* solutions, namely (i) $X = a, Y = b$ and (ii) $X = b, Y = a$. The set unification algorithm can compute all these solutions, one after the other, through backtracking. The following is an example that shows how to exploit this feature within a Java program.

Example 5. (Permutations) Print all permutations of the integer numbers from 1 to n ($n \geq 0$). The problem can be modelled as the problem of unifying a (partially specified) set of n logical variables $\{X_1, \ldots, X_n\}$ with the set of the integer numbers from 1 to n, i.e., $\{X_1, \ldots, X_n\} = \{1, \ldots, n\}$. Each solution to this problem yields an assignment of (distinct) values to variables $X_1, \ldots, X_n$ that represents a possible permutation of the integers between 1 and n. The following method `allPermutations` implements this solution in Java using JSetL.

```
public static void allPermutations(int n) {
    IntLSet I = new IntLSet(1,n);      // I = {1,2,...,n}
    LSet S = LSet.mkLSet(n);           // S = {X1,X2,...,Xn}
    solver.add(S.eq(I));               // {X1,X2,...,Xn} = {1,2,...,n}
    solver.check();
    do {
        S.printElems(' ');
        System.out.println();
    } while (solver.nextSolution());
}
```

The invocation `LSet.mkLSet(n)` creates a set composed of `n` unbound logical variables. This set is unified, through the constraint `eq`, with the set of `n` integers `I`. The method `add` allows a constraint to be added to the *constraint store* of the specified solver. Constraints stored in the constraint store can be checked for satisfiability by calling the constraint solving procedure of the solver, e.g. by invoking the method `check()`. Calling the method `nextSolution()` allows to check whether the constraint in the constraint store of the current solver admits further solutions and possibly to compute the next one. Finally, the invocation

[2] An alternative, more efficient, solution of the coloring problem using constraints will be hinted at in the next section.

`s.printElems(' ')` prints all elements of the set `s` on the standard output separated by the specified character (blank in this case).

This example also illustrates the interplay between Java statements and the non-determinism mechanism embedded in the JSetL constraint solving procedure. In particular, using the method `nextSolution()` within an iterative statement such as the `do_while` construct provides a simple way to compute all possible solutions for a given problem. As a matter of fact, `nextSolution()` exploits the backtracking mechanism embedded in the constraint solver: calling `nextSolution()` forces the computation to go back until the nearest open choice point. In this example the choice points are created during execution of the set unification algorithm called into play by solving the equality constraint between the two sets `I` and `S`.

Remark 1. The set unification problem has been proved to be a *NP-complete problem*. This may lead, in general, to an exponential growth in the complexity of the satisfaction procedure used in JSetL. However, the full power of set unification is not always required. As an example, in the problem of Example 5 we can observe that one of the two sets is completely known (the set `I`), while the other (the set `S`) is a closed set of exactly n unbound variables. Thus we can replace the equality constraint $\mathtt{S} = \mathtt{I}$ that implements general set unification with the conjunction of constraints $\mathtt{1} \in \mathtt{S} \wedge \mathtt{2} \in \mathtt{S} \wedge \ldots \wedge n \in \mathtt{S}$.[3] The new version of the method `allPermutations` using n membership constraints turns out to be executed much more efficiently than the previous one.

4 Constrained Partially Specified Collections

Logical variables occurring in partially specified collections can be suitably constrained. This allows us, in general, to characterize more precisely the collection of data structures denoted by a partially specified collection. For example, the partially specified list $[X, Y]$, where X and Y are unbound logical variables, with the additional constraint $X > Y$, can be used to designate any list of two elements with the first element greater than the second. The following is a simple example showing how inequality and membership constraints can be advantageously exploited in conjunction with partially specified lists.

Example 6. (All pairs) Compute the set of all pairs `[x,y]` such that both `x` and `y` belong to a given set `s` and $\mathtt{x} \neq \mathtt{y}$.

```
public static Set<?> allPairs(Set<?> s) {
    LVar x = new LVar();
    LVar y = new LVar();
    LList pair = new LList.empty().ins(y).ins(x));  // a pair [x,y]
    solver.add(x.in(s).and(y.in(s)).and(x.neq(y)));
    return solver.setOf(pair);
}
```

[3] Note that `S` has fixed cardinality, which prevents it from containing other values other than $1..n$.

The first four statements serve to specify the form of the pairs we are looking for: partially specified lists of the form `[x,y]`, with $\texttt{x} \in \texttt{s} \wedge \texttt{y} \in \texttt{s} \wedge \texttt{x} \neq \texttt{y}$. The last statement calls the method `setOf` of the current solver which collects in a Java set all the pairs satisfying this specification and returns it as its result.

This example shows an interesting interaction between partially specified collections (namely, JSetL logical lists), containing logical variables, and completely specified collections (namely, sets and lists of `java.util`). The latter are used to contain known values, while the former are used to describe the general form of values to be computed.

The method `setOf` implements a (limited) form of *intensionally defined sets*. A set s is defined as the set of all possible values of a variable x for which a condition φ, involving x, turns out to be satisfied, i.e. $s = \{x : \varphi\}$. If φ is represented as a JSetL constraint, the set collection can be easily performed by using `setOf()` over this constraint. As a matter of fact, the set s computed in the above example is the set $\{[\texttt{x},\texttt{y}] : \texttt{x} \in \texttt{s} \wedge \texttt{y} \in \texttt{s} \wedge \texttt{x} \neq \texttt{y}\}$.

The same effect can be obtained more explicitly by combining iteration and `nextSolution()`, similarly to Example 5. Specifically, the call to the method `setOf` in Example 6 can be replaced by the following code fragment:

```
Set pairSet = new HashSet();
solver.check();
do {
    pairSet.add(pair.getValue());
} while (solver.nextSolution());
```

In this solution each computed pair is explicitly added to the set of all solutions s by using the (destructive) `add` method of Java sets. Collecting all solutions in a data structure such as s—instead of simply printing them as in Example 5—allows the program to subsequently work on them using operations provided by that data structure.

In the next example we make use of a list of constrained integer logical variables to represent the solution we are looking for. Assigning possible values to these variables will allow us to obtain all the admissible solutions for the given problem (actually, only one solution in the considered example).

Example 7. (Sort) Sort a collection of n distinct integer numbers in ascending order. The problem can be modelled as a finite domain constraint satisfaction problem in the following way. Let s be the `Set` object representing the collection to be ordered, and $lOrd$ be a list of n integer logical variables X_i, $i = 1 \ldots n$. Variables in $lOrd$ are constrained so that each X_i has domain s and the following conjunction holds: $\bigwedge_{i=1}^{n-1} X_i < X_{i+1}$. Then a solution to this problem is computed by finding the instance of $lOrd$ (i.e., an assignment of values to all its variables) that satisfies the given constraint. This modelling of the sorting problem is easily implemented in Java using JSetL by the following method:

```
public static List sortList(Set s) {
    int n = s.size();
```

```
    IntLList lOrd = IntLList.mkLList(n);      // lOrd = [X1,X2,...,Xn]
    Iterator it = lOrd.iterator();            // it = Iterator over LList
    while(it.hasNext())                       // ∀i ∈ 1..n, Xi.in(s)
        solver.add((((IntLvar)it.next()).in(s)));
    for(int i=0; i<n-1; i++)                  // ∀i ∈ 1..n, Xi.lt(Xi+1)
        solver.add(((IntLvar)lOrd.get(i)).lt((IntLvar)lOrd.get(i+1)));
    solver.check();
    return lOrd.getValue();
}
```

The invocation `IntLList.mkLList(n)` creates a partially specified list composed of `n` integer logical variables (`IntLList` is a subclass of `LList` where list elements are restricted to be integers—either constants or logical variables of type `IntLVar`). The `while` statement allows membership constraints to be added to the *constraint store* of the constraint solver `solver` for each variable of the list `ordList`. The next `for` statement adds the `lt` constraints that force values for the variables in `ordList` to be assigned respecting the desired ordering relation (namely, $<$). These constraints, along with the membership constraints posted before, are checked for satisfiability through the method `check()`. Finally, the Java list representing the value of the logical list `ordList` is obtained by invoking the method `getValue` and returned as the final result of the method `sortList`. As an example, if `s` is the set $\{5, 2, 4\}$, the constraint to be solved is

$$X_1 \in \{5,2,4\} \wedge X_2 \in \{5,2,4\} \wedge X_3 \in \{5,2,4\} \wedge X_1 < X_2 \wedge X_2 < X_3.$$

Efficiency of the proposed solutions strongly depends in general on how efficiently the constraint solver can handle the involved constraints. For instance, if the constraints generated in Example 7 are solved by using a simple generate & test approach, computational complexity of the proposed solution is in $\mathcal{O}(n!)$ and it is clearly unacceptable even for extremely small n. Conversely, if the solver can exploit the efficient techniques used to solve finite domain (FD) constraints (see, e.g., [10] for an overview), the computational behavior of the proposed solution is in $\mathcal{O}(2^n)$ (the assignments of values to variables that the solver has to test constitute paths of a *binomial tree*), and this is more acceptable, at least for small values of n. This behavior is obtained in JSetL by simply replacing the membership constraint `in` by a constraint `dom` in the first `while` statement[4], and adding the posting of the `label` constraint over the variables in `lOrd`, i.e., `solver.add(lOrd.label())`, before calling the `check` method. With the new solution, values will be assigned one by one to the unbound variables by the `label` constraint and propagation will shrink the domains of the involved variables consequently, until all the domains get reduced to singletons.

Sometimes it may be convenient to reformulate the problem solution in order to exploit the more efficient constraint solving techniques made available by the constraint solver. For example, the simple but very inefficient solution

[4] The semantics of `x.dom(s)` is the same as that of the membership constraint `x.in(s)`, but solving the former simply updates the domain of `x` to `s`, whereas the latter (non-deterministically) assigns all values in `s` to `x`.

to the coloring problem of Example 4 can be replaced by a slightly more complex but much more efficient solution that exploits inequality constraints and FD constraint solving. The only differences w.r.t. the previous version are that colors are represented by integers and the set of all regions (actually, a partially specified set containing one logical variable for each region in the map) is passed explicitly to the method `coloring`. The idea is to state that the domain of each variable representing a region is the set of colors, and then to post the constraints that specify that the two regions in each pair of the map must be distinct.

However, it is important to note that, generally speaking, execution efficiency is not a primary requirement when using the kind of partially specified collections we are considering in this paper. Actually, easiness of problem modelling, as well as easiness of program development and understanding, are definitively more important features in this context.

5 Set Constraints

In Section 4 we have considered constraints over the *elements* of a (partially specified) collection. However, logical variables can range over the domain of collections as well (e.g., sets), and constraints can apply to *whole collections*. Various proposals exist in the literature that take into account this kind of constraints (see [12] for a survey). In particular, *set constraints* [11] are based on set variables whose domain is a finite collection of sets of integers, usually specified as a *set interval* $[l, u]$ representing the lattice of sets induced by the subset partial ordering relation $\subseteq$ having sets l and u as the greatest lower bound and the least upper bound, respectively.

Implementations of set constraints solvers are provided, e.g., by ECLiPSe [4], B-Prolog [20], Mozart-Oz [15], and Choco [3]. All these proposals consider only completely specified sets, whose elements are usually restricted to integer values.

However, as shown in previous sections, it may be useful in many occasions to allow also partially specified collections to be taken into account. As a matter of fact, this could significatively enhance the problem modelling capabilities. For example, if we want to say that s is a set containing at least the element 1 but not containing 0 we could simply state this as

$S = \{1 \mid R\} \wedge 0 \notin S$

that is, by using an open partially specified set and a non-membership (set) constraint over it.

Though partially specified, it may nevertheless be convenient to perform various operations on the (whole) collections. The case of equality over partially specified lists and sets has already been discussed in Section 3, where it is reconduct to list/set unification. Other operations, such as membership, inclusion, union, intersection, cardinality, and so on, could be conveniently applied to partially specified collections as well. For example, in the above example, we could want to say also that the intersection between the set S and another set T must be empty, i.e., $S \cap T = \emptyset$.

The ability to process such operations on partially specified collections require to deal with them as constraints and to have a constraint solver that is able to properly account for possibly unbound variable occurring within the collections. Such a solver could be able, in particular, to detect unsatisfiability of the given constraint even if the involved collections are only partially known, possibly leading in this way to significatively cut the search space. With reference to the above example, if we further require that the set T contains the set $\{1\}$, that is:

$S = \{1 \mid R\} \wedge 0 \notin S \wedge S \cap T = \emptyset \wedge T \subseteq \{1\}$

the solver could be able to immediately detect that the constraint is unsatisfiable.

A complete solver for these kind of constraints, specifically constraints over partially specified sets, is provided by {log} [6, 7, 5]. A Java implementation of this solver is also made available in the context of the Java language by the JSetL library. Obviously solving this kind of constraints turns out to be in general much more inefficient than solving set constraints over only completely specified sets. Moreover, as it happens with set unification, the solution of many of these constraints may involve non-determinism.

The following are a few examples of the resolution of simple constraints over either open or closed partially specified sets in JSetL.

Example 8. Partially specified set constraints.

```
LVar x = new LVar();
IntLVar n = new IntLVar();
LSet r = new LSet();
LSet s1 = LSet.empty().ins(1);                  // s1 = {1}
LSet s2 = LSet.empty().ins(2).ins(x);           // s2 = {x,2}
LSet s3 = r.ins(1).ins(x);                      // s3 = {x,1 | r}
```

- Solving the constraint

  ```
  solver.solve(r.inters(s1,s2));                // r = s1 ∩ s2
  ```

  we get the two distinct answers: $x \neq 1 \wedge r = \{\}$ and $x = 1 \wedge r = \{1\}$.
- Solving the constraint

  ```
  solver.solve(s2.size(n));                     // n = |s2|
  ```

  we get the two distinct answers: $x \neq 2 \wedge n = 2$ and $x = 2 \wedge n = 1$.
- Solving the constraint

  ```
  solver.solve(s3.ncontains(0));                // 0 ∉ s3
  ```

  we get: $x \neq 0 \wedge 0 \notin r$; if we would consider in addition the constraint $1 \notin s3$ then we would get an immediate failure.

The ability to deal with collections whose elements can be of any type (not only integers) allows one to account also for collections containing other collections as their elements, i.e., *nested collections*. Moreover, the inner collections can be partially specified collections or even unbound logical lists/sets. For example, the object `ss`, which is created by the following two Java statements using JSetL:

```
LSet r = new LSet();
```

```
LSet ss = new LSet().ins(r);
```
represents any set containing at least another (nested) set `r`. As a more concrete example, if we want to model a system composed of at least k pieces, each of which has a name $\mathtt{n}_i$, a quantity $\mathtt{q}_i$ and a unit cost $\mathtt{c}_i$, and if we assume that the name and cost of each piece are known, whereas its quantity is unknown, we can represent such a system by the following partially specified set of lists (actually triples):

$\{[\texttt{"}\mathtt{n}_1\texttt{"},\mathtt{c}_1,\mathtt{q}_1],\ldots,[\texttt{"}\mathtt{n}_k\texttt{"},\mathtt{c}_k,\mathtt{q}_k] \mid \mathtt{r}\}$

where $\mathtt{q}_1$, ..., $\mathtt{q}_k$ are unbound integer logical variables and `r` is an unbound logical set variable. Then we can post constraints either on the integer logical variables $\mathtt{q}_i$ (e.g., $\mathtt{q}_i$`.in(1,10)`), or on the set variable `r` or on the set representing the whole system (e.g., `r.contains(["o",c,q])`, `c`, `q` unbound logical variables).

6 Discussion and related work

Logic variables are a key feature of languages/libraries to support *constraint programming*. However, in the context of conventional programming languages, logical variables are rarely exploited to support further possibilities, such as:

- in conjunction with unification, to allow the same code to be used for different purposes, e.g., to test a given condition or to compute those values for which the condition holds (*procedure invertibility*);
- to allow *partially specified collections* to be represented and handled, possibly by using suitable constraints.

As concerns procedure invertibility, one exception is the language Alma-0 [1] which explicitly considers it by providing a generalized test expression (which becomes an assignment in the case one side is un unbound variable and the other side is an expression with a known value) and suitable parameter passing mechanisms.

As concerns partially specified collections, languages/libraries that support constraint programming in the context of conventional programming usually allow logic variables to occur in compound data structures. For instance, Alma-0 allows the user to define arrays of (possibly unbound) logical variables, and it allows the programmer to deal with them by a few global constraints such as the `ALL_DIFFERENT` constraint. However, no one of these proposals does really consider these structures as partially specified data and provide operations to deal with them accordingly.

Of course, partially specified collections can be defined in the formal languages provided by generic proof assistants such as Coq [2] and Isabelle [16] and manipulated as such by using their primitive facilities. However, these tools are hardly exploitable from within conventional O-O languages such as Java, in the manner we have shown in this paper.

As concerns efficiency, we have no precise empirical assessment with respect to Prolog systems where unification and backtracking are primitive mechanisms

(actually, as already mentioned, performance is not our main concern). However, unification, constraint solving and (embedded) backtracking can, at least in principle, be implemented within the library as efficiently as they are implemented in Prolog systems. With this respect, there is no price to pay for integrating concepts from logical programming into Java. Conversely, user-defined non-deterministic procedures, that can be defined in JSetL by exploiting the user-defined constraint facility [18], are likely to be much more inefficient in JSetL than in Prolog.

7 Conclusions

In this paper we have presented the potential offered by the presence of partially specified collections, in particular, lists and sets, in a conventional O-O programming language. Usefulness of partially specified collections in declarative programming languages is widely accepted. We have shown that introducing this kind of data abstractions in a more conventional programming setting is feasible and offers several potential advantages. To support our claim we have presented a number of simple examples written in Java using JSetL, a general-purpose library that provides general forms of data collections.

Although we have considered only lists and sets, there are a number of other kinds of data collections, akin to lists and sets, which are often provided by programming languages and libraries and which could be allowed to be partially specified. The solutions and techniques described in this paper could be adapted to these other data collections as well. In particular, a formal characterization of (possibly partially specified) multisets, i.e., unordered collections where element repetitions are allowed, and the definition of suitable solving procedures for equality and inequality constraints over them can be found in [9].

A weak point of our library-based approach is that the notation, necessary to respect the syntax of Java, is much heavier than what one would like to see. A preprocessor is under development at present to remedy, at least partially, this problem. No real extension to the Java syntax is required but a smart use of comments. Using the preprocessor, it will be possible for instance to create a set by using a sort of set literal, such as `LSet s1 = new LSet(/*{1,2,3}*/)`.

As a future work, we plan to compare the proposed approach with *lazy data structures* as those provided, e.g., by the Scala language [17], that seem to have interesting similarities with our partially specified structures. However, a strict requirement of our work is to use a library-based approach. This approach has the undeniable advantage of being easier to develop and more likely to be accepted by the (rather conservative) programmers. On the other hand, integration between the considered data abstractions and the rest of the language is weaker in our proposal than in an extended language, such as Alma-0 or Scala, where the desired abstractions are added as first-class citizen of the language.

References

1. Apt, K.R., Schaerf, A. Programming in Alma-0, or Imperative and Declarative Programming Reconciled. In D. M. Gabbay and M. de Rijke, Eds., *Frontiers of Combining Systems 2*, 1–16, Research Studies Press Ltd (2000)
2. Bertot, Y., Castéran, P. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*. Springer, 469 pages (2004)
3. Choco. `http://www.emn.fr/z-info/choco-solver/`.
4. ECLiPSe *The ECLiPSe Constraint Logic Programming System*. London. `http://eclipseclp.org/`.
5. Dal Palù, A., Dovier, A., Pontelli, E., and Rossi, G. Integrating Finite Domain Constraints and CLP with Sets. In Miller, D., Ed., *Fifth ACM-SIGPLAN Int'l Conf. on Principles and Practice of Declarative Programming*, ACM Press, 219–229 (2003)
6. Dovier, A., Omodeo, E. G., Pontelli, E., and Rossi, G. {log}: A Language for Programming in Logic with Finite Sets. *Journal of Logic Programming*, 28(1):1–44 (1996)
7. Dovier, A., Piazza, C., Pontelli, E., and Rossi, G. Sets and Constraint Logic Programming. *ACM Transactions on Programming Languages and Systems*, 22(5):861–931 (2000)
8. Dovier, A., Pontelli, E., and Rossi, G. Set unification. *Theory and Practice of Logic Programming*, 6:645–701 (2006)
9. Dovier, A., Piazza, C., and Rossi G. A uniform approach to constraint-solving for lists, multisets, compact lists, and sets. *ACM Transactions on Computational Logic*, 9(3):1–30 (2008)
10. Henz, M., Müller, T. An Overview of Finite Domain Constraint Programming *Proceedings of the Fifth Conference of the Association of Asia-Pacific Operational Research Societies*, Singapore (2000)
11. Gervet, C. Interval Propagation to Reason about Sets: Definition and Implementation of a Practical Language. *Constraints*, 1(3):191–244, (1997)
12. Gervet, C. Constraints over Structured Domains. In Rossi, F., van Beek, P., and Walsh, T., Eds., *Handbook of Constraint Programming*, Elsevier (2006)
13. ILOG *ILOG Solver 6.0 Reference Manual* (2003)
14. JSR-331, Java Constraint Programming API (Early Draft). Java Community Process. `http://www.jcp.org` (2010)
15. The Mozart Consortium. The Mozart programming system. `http://www.mozart-oz.org` (2006)
16. Nipkow, T., Paulson, L., and Wenzel, M. (2002) *Isabelle/HOL. A Proof Assistant for Higher-Order Logic.* LNCS 2283, Springer Verlag, 218 pages.
17. Odersky, M., Spoon, L., and Venners, B. *Programming in Scala: a comprehensive step-by-step guide.* Artima Press (2008)
18. Rossi, G., Bergenti, F. Nondeterministic Programming in Java with JSetL. In Cantone, D., Nicolosi Asmundo, M., eds, CILC 2013: Italian Conference on Computational Logic, CEUR Workshop Proceedings, Vol. 1068, 211-226 (2013)
19. Rossi, G., Panegai, E., and Poleo, E. JSetL: A Java Library for Supporting Declarative Programming in Java. *Software-Practice & Experience*, 37:115-149 (2007)
20. Zhou, N-F. B-Prolog `http://www.probp.com/`.