Verifying B Proof Rules using Deep Embedding and Automated Theorem Proving

Mélanie Jacquel[1], Karim Berkani[1], David Delahaye[2], and Catherine Dubois[3]

[1] Siemens SAS I MO, Châtillon, France,
Melanie.Jacquel@siemens.com
Karim.Berkani@siemens.com
[2] CEDRIC/CNAM, Paris, France,
David.Delahaye@cnam.fr
[3] CEDRIC/ENSIIE, Évry, France,
dubois@ensiie.fr

Abstract. We propose a formal and mechanized framework which consists in verifying proof rules of the B method, which cannot be automatically proved by the elementary prover of Atelier B and using an external automated theorem prover called Zenon. This framework contains in particular a set of tools, named BCARe and developed by Siemens SAS I MO, which relies on a deep embedding of the B theory within the logic of the Coq proof assistant and allows us to automatically generate the required properties to be checked for a given proof rule. Currently, this tool chain is able to automatically verify a part of the derived rules of the B-Book, as well as some added rules coming from Atelier B and the rule database maintained by Siemens SAS I MO.

Keywords: B Method, Proof Rules, Verification, Deep Embedding, Automated Theorem Proving, Coq, Zenon.

1 Introduction

The B method [1], or B for short, allows engineers to develop software with high guarantees of confidence; more precisely it allows them to build correct by design software. B is a formal method based on theorem proving and emphasizing a refinement-based development process. A typical scenario consists in first writing high-level formal specifications as abstract machines, and then refining them step by step into low-level sequential pseudo-code that can be automatically translated into C or Ada programs. Proof is required to verify the correctness of abstract machines (mainly ensuring the preservation of user-written invariant properties) and the correctness of the refinement steps (roughly speaking, the behavior is preserved by the refinement steps that introduce algorithmic decisions or data representation choices). In practice, it means that the user must discharge proof obligations. The Atelier B environment [8] is a platform that supports B and offers, among other tools, both automated and interactive provers.

A famous and significant use of B and Atelier B has concerned the control system of the driverless Meteor line 14 metro in Paris (opened 13 years ago).

Since Meteor, Siemens SAS I MO has generalized its use of B for building other critical systems, e.g. the communication-based train control systems of the New York City Canarsie line. On both cases, a huge number of proof obligations (27,800 obligations for Meteor) had to be handled manually (using the interactive prover). In fact, most of them were proved by adding new proof rules (1,400 rules for Meteor) that the automated provers can exploit. Even today, many projects developed at Siemens SAS I MO still require to add new proof rules.

These new proof rules must of course be proved correct, otherwise the proof process is invalid. The proof of these proof rules is done by the elementary prover provided by Atelier B, which does not use any of them. Some of the added rules (900 rules in the Meteor case) can be proved by the Atelier B elementary prover, some of them cannot and are then proved manually by experts.

The main point in this approach is to prove the proof rules, basic or added ones. Currently, the proof rule database of Atelier B used at Siemens SAS I MO contains about 5,300 proof rules, 2,900 of which can be proved automatically by the elementary prover. The problem raised here is not to question the confidence into the Atelier B elementary prover, but to prove the proof rules that are not automatically proved and that must be proved manually. The latter process is tedious, long, and error-prone. We propose to replace it by a mechanized verification with the help of a more powerful Automated Theorem Prover (ATP), a first order one, e.g. Zenon [5]. In order to increase confidence in this external verification, it is important to be able to check the proofs done automatically, and furthermore to be able to connect them with the inference rules of the underlying B logic [1]. Thus, our approach is not only to use an external prover, but also to rely on a proof assistant which checks the generated proofs, i.e. Coq [15].

However, as Coq is not fully automated and may require human interaction, we propose to use Coq only to describe the B underlying logic and to serve as a proof verifier for the proofs delegated to the ATP. The expected results for a B proof rule will be, in case of success, a proof in the B logic (or, more technically, a Coq proof term encoding it). Some years ago, a first experiment using Coq has been conducted at Siemens SAS I MO to verify the Atelier B proof rules (see [3]), but it required human interaction and all the proofs done in Coq were done manually. However, this first manual attempt allowed us to handle 274 proof rules proved manually by experts and considered by the authors as representative ones. The methodology consisted in playing the manual proofs within Coq: 7 valid proof rules had incorrect proofs but the proofs could be given properly within Coq, and 13 rules were not valid because of lack of hypotheses about variable non-freeness. This discovery were important for the design of the verification platform presented in this paper, namely BCARe.

BCARe is a set of tools developed by Siemens SAS I MO to verify added proof rules. It contains tools to check if a proof rule is correctly protected by non-freeness assumptions, to typecheck a rule, and to prove a rule (by using Coq and Zenon). One of the main objectives of BCARe is to assist the experts to find proofs of proof rules, but the development standards used at Siemens SAS I MO expect these experts to give their final assessment. BCARe contains a deep embedding

of the B logic within Coq, that is an encoding of the B formulas and inference rules into the Coq logic, namely the calculus of inductive constructions. Other embeddings [4, 6, 7, 12] of B have been implemented with different purposes in related work. For example, BiCoq [12] is a deep embedding of B in Coq. Like BCARe, BiCoq follows scrupulously the B-Book [1]; however, the former uses names whereas the latter uses De Bruijn indexes.

Some experiments like [9, 14, 10] concern automated verification of B proof obligations with ATPs or SMT solvers. We are interested in proof rules and not in proof obligations. Furthermore, as we do want the best degree of confidence in our mechanical proofs, it is essential to rely on an ATP able to provide proof traces checkable by a proof checker, e.g. Coq. Zenon is one of the ATPs able to provide several output proof formats, one of which is a Coq script that will be adapted to give us a proof using the B logic.

The paper is organized as follows: in Section 2, we first present the several steps required to verify B proof rules; we then introduce, in Section 3, the BCARe environment, which is a mechanized support for the verification of proof rules; finally, in Section 4, we describe our experiments for automating the verification proofs and provide some benchmarks concerning derived rules and added rules coming from Atelier B and the rule database maintained by Siemens SAS I MO.

2 Rule Verification in Atelier B

In this section, we present the notion of proof rules of Atelier B, together with the several steps required to ensure their verification, i.e. the steps which guarantee that the application of such rules does not introduce inconsistencies.

2.1 The B Set Theory

The B method [1] aims to assist experts to develop certified software. The initial step is defined with abstract properties of a model. Several steps of property refinement are then applied until the release of the complete software. A refinement step is characterized by adding details on the software behavior under construction. For each step, generated proof obligations must be demonstrated.

The B method is based on a typed set theory. There are two rule systems: one for demonstrating that a sentence is well-typed, and one for demonstrating that a sentence is a logical consequence of a set of axioms. The main aim of the type system is to avoid inconsistent sentences, such as Russell's paradox for example. The B proof system is based on a sequent calculus with equality. Six axiom schemes define the basic operators and the extensionality which, in turn, defines the equality of two sets. In addition, the other operators ($\cup$, $\cap$, etc) are defined using the previous basic ones.

2.2 The Atelier B Proof Assistant

Proofs Atelier B [8] is a tool, developed by ClearSy, that implements the B method. Once a model is specified, its correctness is ensured by several mecha-

nisms. The first one is typechecking, which is fully automated. If no error occurs during typechecking, then the proof obligations can be generated. They represent the properties that must be proved to verify the mathematical correctness of the model compared with its properties. A proof system helps the developer make the corresponding demonstrations.

These proof obligations can be demonstrated automatically with a tactic of the Atelier B proof assistant. When this tactic fails, an interactive proof mode can be used. In this proof mode, the user can apply tactics on the goal and/or on the hypotheses to complete his/her proof. A tactic is an ordered list of theories (a theory is basically a container of rules), that determines the traversal of a rule base to determine if one or several rules can be applied.

Rules We distinguish two kinds of rules in Atelier B: deduction and rewrite rules. The former is of the form $A_1 \wedge \ldots \wedge A_n \Rightarrow B$, where the A_i are the antecedents (guards or predicates) and B the consequent (predicate). Guards are used to add some conditions to the use of the rule. A deduction rule can be used in both backward and forward ways. A rewrite rule is of the form $A_1 \wedge \ldots \wedge A_n \Rightarrow B == C$, where the A_i are the antecedents (guards or predicates) and where B and C are either expressions or predicates. The binary symbol "==" is the syntactic replacement: if B matches a subterm of the goal, then it is replaced by the corresponding instance of C. More precisely, the syntax of rules is defined as follows:

$$
\begin{array}{l}
V := I \mid V \mapsto V \\
E := V \mid [V := E]E \mid E \mapsto E \mid \mathit{choice}(S) \mid S \\
S := S \times S \mid \mathbb{P}(S) \mid \{V|P\} \mid \mathit{BIG} \mid I \\
P := P \wedge P \mid P \Rightarrow P \mid \neg P \mid \forall V.P \mid [V := E]P \mid E = E \mid E \in S \mid I \\
A := P \mid I \setminus P \mid I \setminus E \mid \mathsf{binhyp}(P) \mid \mathsf{blvar}(I) \mid A \wedge A \\
C := P \mid E == E \mid P == P \\
R := C \mid A \Rightarrow C
\end{array}
$$

where V represents the variables (in which I denotes the identifiers), E the expressions, S the sets, P the predicates, A the antecedents, C the consequents, and R the rules. Regarding guards, we only consider the non-freeness predicates ($I \setminus P$ and $I \setminus E$), as well as the binhyp and blvar guards, where $\mathsf{binhyp}(P)$ checks the presence of P in the proof context and $\mathsf{blvar}(I)$ instantiates I with the bound variables at the rewrite point. We are also able to deal with more complicated guards that we will not present in this paper. Furthermore, free variables may occur in E, S, and P; these variables are considered as metavariables (for pattern-matching). As for the other logical connectives (such as $\Leftrightarrow$, $\vee$, and $\exists$) and set operators (e.g. $\cup$, $\cap$, etc), they are defined from those given above.

Let us illustrate the two previous kinds of rules with some examples of added rules coming from Atelier B.

Example 1 (Deduction Rule). ForAllX.3: $(a \setminus A) \wedge A = \emptyset \Rightarrow \forall a.a \notin A$
If used in a backward way, this rule can be applied on a goal if a is non-free in A and if the current goal matches the consequent $\forall a.a \notin A$. It therefore generates the goal $A = \emptyset$ to be proved.

Example 2 (Rewrite Rule). SimplifyRelDorXY.2:
$\mathsf{binhyp}(f \in u \mathrel{+\!\!\!\rightarrow} v) \wedge \mathsf{binhyp}(a \in dom(f)) \wedge \mathsf{blvar}(Q) \wedge (Q \setminus (f \in u \mathrel{+\!\!\!\rightarrow} v)) \wedge$
$(Q \setminus (a \in dom(f))) \Rightarrow \{a\} \lhd f == \{(a \mapsto f(a))\}$
This rule can be applied on a goal if there are some hypotheses of the context matching $f \in u \mathrel{+\!\!\!\rightarrow} v$ and $a \in dom(f)$, a term of the goal matching $\{a\} \lhd f$, and if the quantified variables at the rewrite point do not appear in u, v, a, and f.

Rule Verification The verification of a rule is carried out in four steps:

1. The first step only deals with rewrite rules and consists in verifying that rewrite rules are correctly protected against variable capture. This is due to the fact that Atelier B does not verify the context of application when applying a rewrite rule and applies it in a purely syntactical way. As a consequence, when a rewrite rule is applied under binders and involves bound variables, variable capture may occur and lead to inconsistencies.
2. The second step aims to verify that the rule is well-formed, which amounts to typechecking the rule according to the B typing rules (see [1]). However, a rule may contain metavariables whose type may be left implicit. Therefore, a preliminary step is required to first infer the types of all metavariables such that the rule enriched with these type constraints can be typechecked.
3. The third step consists in verifying that the rule is well-defined. In [2], it is pointed out that conditional definitions may lead to some ill-defined expressions, such as division by zero or the application of a function to an argument lying outside its domain. A syntactical filter to be applied to the rule is proposed and contains all the well-definedness proof obligations.
4. The last step must verify that the rule can be derived using the B proof rules (see [1]). It is possible to do so over a rule, after applying another syntactical filter, defined in [2] in particular, in order to remove the proof obligations related to well-definedness.

3 The BCARe Environment

In this section, we present the BCARe environment, which is developed by Siemens SAS I MO, and which proposes a formal and mechanized framework for verifying B proof rules.

3.1 Rationale for Designing BCARe

Currently, an automated tool is used at Siemens SAS I MO for verifying the rules developed with Atelier B. However, when a proof fails, the rule is verified manually without the help of any proof assistant. The first aim of the BCARe environment, developed by Siemens SAS I MO, is to overcome this problem. For example, in the rule ForAllX.3, $a \setminus A$ must be verified before the application of the rule. It is possible to check that the previous condition is necessary with BCARe, while it is impossible to do so with the other available tools. Thus, the

BCARe environment has been essentially developed to deal with the rules whose correctness cannot be automatically established. The scope of this environment is currently a subset of the B set theory (propositional and first order logics, basic set theory operators, functions, generalized and quantified intersections and unions). Some other features, such as induction, are being integrated.

The different steps of a rule verification with BCARe follows the several steps defined in Section 2. If the rule is a rewrite rule then a tool checks that its guards correctly protect the free variables (see Subsection 3.3). Another tool then infers types for the rule using a type inference algorithm which has been defined regarding the B typing rules (see [1]), after which the typing lemma can be generated (see Subsection 3.4). Finally, this tool also generates the well-definedness lemma, as well as the lemma corresponding to the rule itself (See Subsections 3.5 and 3.6). Once these three lemmas are generated, their proofs must be completed. The generation of these lemmas and the corresponding proofs are realized using the Coq proof assistant [15]. In particular, this relies on an environment, called BCoq, which is an embedding of the B set theory in Coq (see Subsection 3.2). The proofs of these lemmas can be partially automated, and Section 4 describes our approaches regarding the automation of these proofs.

3.2 The BCoq Embedding

The generation of the previous lemmas is realized within the BCoq environment, which is a deep embedding of the B set theory in Coq, and where the B operators, as well as the deduction systems for types and proofs, are specified inductively (see [3]). Compared to a shallow embedding, the advantage of such an approach is that the correctness of a type or proof derivation is provided by construction.

The BCoq syntax is defined in Coq as follows (we do not provide the Coq concrete syntax, but only an abstraction of this syntax written with "$\dot{\cdot}$"):

$$\dot{V} := \dot{I} \mid \dot{V} \mathrel{\dot{\mapsto}} \dot{V}$$
$$\dot{E} := \dot{V} \mid [\dot{V} :\dot{=} \dot{E}]\dot{E} \mid \dot{E} \mathrel{\dot{\mapsto}} \dot{E} \mid \dot{choice}\ (\dot{S}) \mid \dot{S}$$
$$\dot{S} := \dot{S} \times \dot{S} \mid \dot{\mathbb{P}}\ (\dot{S}) \mid \{\dot{V} \mid \dot{P}\} \mid \dot{BIG} \mid \dot{I}$$
$$\dot{P} := \dot{P} \mathrel{\dot{\wedge}} \dot{P} \mid \dot{P} \mathrel{\dot{\Rightarrow}} \dot{P} \mid \dot{\neg}\ \dot{P} \mid \dot{\forall}\ \dot{V}.\dot{P} \mid [\dot{V} :\dot{=} \dot{E}]\dot{P} \mid \dot{E} \mathrel{\dot{=}} \dot{E} \mid \dot{E} \mathrel{\dot{\in}} \dot{S} \mid \dot{I}$$

where $\dot{I}$, $\dot{V}$, $\dot{E}$, $\dot{S}$, and $\dot{P}$ respectively represent the reified versions of the several sets of terms I, V, E, S, and P, defined in Section 2.

In this grammar, there is no syntax for rules, as they are intended to be reified into predicates. However, there are still metavariables (free variables occurring in $\dot{E}$, $\dot{S}$, and $\dot{P}$). The reification process is performed by using the set of functions $[\![\cdot]\!]_X$, where $X \in \{V, E, S, P, A, C, R\}$, and which are defined in Figure 1. In this process, the sets of metavariables are computed, and these metavariables are then bound by means of shallow binders. The names of binders are managed using shallow binders as well, in order to deal with α-conversion and skolemization in particular (see Subsection 3.4). In the same way, the non-freeness guards are reified and kept in the term using shallow non-dependent products.

$[\![I_1]\!]_V = (I_2, \{(I_1, I_2)\})$, where $I_2 \notin \mathcal{V} \cup \mathcal{B}$, and $\mathcal{B} \leftarrow \mathcal{B} \cup \{I_2\}$

$[\![V_1 \mapsto V_2]\!]_V = (V_1' \dot{\mapsto} V_2', \{B_1 \cup B_2\})$, where $(V_1', B_1) = [\![V_1]\!]_V$ and $(V_2', B_2) = [\![V_2]\!]_V$

$$[\![I_1]\!]_V^b = \begin{cases} I_2, \text{ if } (I_1, I_2) \in b \\ I_1, \text{ otherwise and } \mathcal{M}_E \leftarrow \mathcal{M}_E \cup \{I_1\} \end{cases} \qquad [\![V_1 \mapsto V_2]\!]_V^b = [\![V_1]\!]_V^b \dot{\mapsto} [\![V_2]\!]_V^b$$

$[\![V_1]\!]_E^b = [\![V_1]\!]_V^b \qquad [\![[V_1 := E_1]E_2]\!]_E^b = [V_2 :\dot{=} [\![E_1]\!]_E^b][\![E_2]\!]_E^{b \cup B}$, where $(V_2, B) = [\![V_1]\!]_V$

$[\![E_1 \mapsto E_2]\!]_E^b = [\![E_1]\!]_E^b \dot{\mapsto} [\![E_2]\!]_E^b \qquad [\![S_1]\!]_E^b = [\![S_1]\!]_S^b$

$[\![S_1 \times S_2]\!]_S^b = [\![S_1]\!]_S^b \dot{\times} [\![S_2]\!]_S^b \qquad [\![\mathbb{P}(S_1)]\!]_S^b = \dot{\mathbb{P}}\,([\![S_1]\!]_S^b)$

$[\![\{V_1 | P_1\}]\!]_S^b = \{V_2 \mid [\![P_1]\!]_P^{b \cup B}\}$, where $(V_2, B) = [\![V_1]\!]_V$

$[\![I_1]\!]_S^b = I_1$, and $\mathcal{M}_S \leftarrow \mathcal{M}_S \cup \{I_1\}$ if $I_1 \notin b$

$[\![P_1 \wedge P_2]\!]_P^b = [\![P_1]\!]_P^b \dot{\wedge} [\![P_2]\!]_P^b \qquad [\![P_1 \Rightarrow P_2]\!]_P^b = [\![P_1]\!]_P^b \dot{\Rightarrow} [\![P_2]\!]_P^b$

$[\![\neg P_1]\!]_P^b = \dot{\neg}\,[\![P_1]\!]_P^b \qquad [\![\forall V_1.P_1]\!]_P^b = \dot{\forall}\, V_2.[\![P_1]\!]_P^{b \cup B}$, where $(V_2, B) = [\![V_1]\!]_V$

$[\![[V_1 := E_1]P_1]\!]_P^b = [V_2 :\dot{=} [\![E_1]\!]_E^b][\![P_1]\!]_P^{b \cup B}$, where $(V_2, B) = [\![V_1]\!]_V$

$[\![E_1 = E_2]\!]_P^b = [\![E_1]\!]_P^b \doteq [\![E_2]\!]_P^b$

$[\![E_1 \in S_1]\!]_P^b = [\![E_1]\!]_E^b \dot{\in} [\![S_1]\!]_P^b \qquad [\![I_1]\!]_P^b = I_1$, and $\mathcal{M}_P \leftarrow \mathcal{M}_P \cup \{I_1\}$ if $I_1 \notin b$

$[\![P_1]\!]_A = [\![P_1]\!]_P^\emptyset \qquad [\![I_1 \setminus P_1]\!]_A = \top$, and $\mathcal{N} \leftarrow \mathcal{N} \cup [\![I_1]\!]_V^\emptyset \mathbin{\dot{\setminus}} [\![P_1]\!]_P^\emptyset$ if $I_1 \notin \mathcal{R}$

$[\![I_1 \setminus E_1]\!]_A = \top$, and $\mathcal{N} \leftarrow \mathcal{N} \cup [\![I_1]\!]_V^\emptyset \mathbin{\dot{\setminus}} [\![E_1]\!]_E^\emptyset$ if $I_1 \notin \mathcal{R}$

$[\![\mathsf{binhyp}(P_1)]\!]_A = [\![P_1]\!]_P^\emptyset \qquad [\![\mathsf{blvar}(I_1)]\!]_A = \top$ with $\mathcal{R} \leftarrow \mathcal{R} \cup \{I_1\}$

$$[\![A_1 \wedge A_2]\!]_A = \begin{cases} [\![A_i]\!]_A, \text{ if } [\![A_j]\!]_A = \top \text{ with } (i,j) = (1,2) \text{ or } (2,1) \\ \top, \text{ if } [\![A_i]\!]_A = \top \text{ with } i = 1,2 \\ [\![A_1]\!]_A \dot{\wedge} [\![A_2]\!]_A, \text{ otherwise} \end{cases}$$

$[\![P_1]\!]_C = [\![P_1]\!]_P^\emptyset \qquad [\![E_1 == E_2]\!]_C = [\![E_1 = E_2]\!]_P^\emptyset \qquad [\![P_1 == P_2]\!]_C = [\![P_1]\!]_P^\emptyset \dot{\Leftrightarrow} [\![P_2]\!]_P^\emptyset$

$$[\![A_1 \Rightarrow C_1]\!]_R = \begin{cases} \forall x \in \mathcal{M}, \mathcal{B}, \mathcal{N}.[\![C_1]\!]_C^\emptyset, \text{ if } [\![A_1]\!]_A = \top \\ \forall x \in \mathcal{M}, \mathcal{B}, \mathcal{N}.[\![A_1]\!]_A \dot{\Rightarrow} [\![C_1]\!]_C^\emptyset, \text{ otherwise} \end{cases}$$

$[\![C_1]\!]_R = \forall x \in \mathcal{M}, \mathcal{B}, \mathcal{N}.[\![C_1]\!]_C^\emptyset$

$\mathcal{V}$	is the set of variables of the initial rule.
$\mathcal{M}_{E/S/P}$	is the set of metavariables of expressions, sets, and predicates.
$\mathcal{M}$	is the set of metavariables $\mathcal{M}_E \cup \mathcal{M}_S \cup \mathcal{M}_P$.
$\mathcal{B}$	is the set of bound variables.
$\mathcal{N}$	is the set of non-freeness hypotheses of the initial rule.
$\mathcal{R}$	is the set of variables bounded by the guard blvar.

$\forall x \in \mathcal{M}, \mathcal{B}, \mathcal{N}.T \equiv$
$\forall_{x \in \mathcal{M}_E} x \in \dot{E}.\forall_{x \in \mathcal{M}_S} x \in \dot{S}.\forall_{x \in \mathcal{M}_P} x \in \dot{P}.\forall_{x \in \mathcal{B}} x \in \dot{I}.N_1 \rightarrow \ldots \rightarrow N_n \rightarrow T$,
where $\mathcal{N} = \{N_1, \ldots, N_n\}$ and T is a reified term.

Fig. 1. Reification of the Atelier B Rules

The BCoq environment also provides the reified relations "$\dot{\vdash}$" and "$\dot{\vdash}_\tau$", respectively for proof and typing judgments (see [1]).

3.3 Rewrite Rule Verification

As said in Section 2, Atelier B does not verify the context of application when applying a rewrite rule and applies it in a purely syntactical way. Therefore, when a rewrite rule is applied under binders and involves bound variables, variable capture may occur and lead to inconsistencies. For instance, let us consider the rewrite rule $\mathsf{binhyp}(x = a) \Rightarrow x == a$ and the goal $n = 0 \vdash \forall n.n \in \mathbb{N} \Rightarrow n = 0$. This goal is trivially false, but the rewrite rule can be applied, which leads to the goals $n = 0 \vdash \forall n.n \in \mathbb{N} \Rightarrow 0 = 0$ and $n = 0 \vdash n = 0$. These two goals can be completed and an inconsistency is then introduced (due to the capture of variable n in hypothesis by the binder of the conclusion).

To avoid variable capture, a first solution is to prevent us from performing rewriting under binders when bound variables are involved. Considering a rewrite rule of the form $G \wedge A \Rightarrow E == F$, where G is the conjunction of the guards, A the conjunction of the antecedents (other than guards), and E and F two expressions, this corresponds to the following criterion:

$$\mathsf{blvar}(Q) \wedge Q \setminus (E = F) \tag{1}$$

Using this criterion, the previous rewrite rule $\mathsf{binhyp}(x = a) \Rightarrow x == a$ is then rejected as the variables x and a are not protected. To be correct, this rule must be of the form $\mathsf{binhyp}(x = a) \wedge \mathsf{blvar}(Q) \wedge Q \setminus x = a \Rightarrow x == a$.

However, this criterion is a little too restrictive as it prevents us from defining some useful rewrite rules. For instance, the rewrite rule $s \cap t == t \cap s$ is rejected by this criterion whereas this rule cannot generate variable capture.

To accept this kind of rewrite rules, a second solution consists in allowing rewriting to be performed under binders only if the bound variables involved do not occur in the antecedents. More precisely, this criterion is defined as follows:

$$\mathsf{blvar}(Q) \wedge Q \setminus (G \wedge A) \tag{2}$$

With this criterion, the corrected rule and the other rule are both accepted. However, criteria (1) and (2) are complementary, and we use both of them as it allows us to accept more rules. Both criteria have been formally verified in [13].

3.4 Rule Typechecking

As seen in Section 2, it is required to verify that a rule is well-formed, which amounts to typechecking the rule according to the B typing rules (see [1]). However, a rule may contain metavariables whose type may be left implicit. For example, in the rule $a \cup b = b \cup a$, the types of a and b are unknown. The B type system does not allow us to infer types for metavariables occurring in rules. It only allows us to check that a predicate is well-typed when all the types are explicit. Therefore, we have to first infer a type for all metavariables.

To do so, we define a type inference system, which is described in Figure 2. The different rules correspond to the core language described in Subsection 3.2 and deal with reified rules. There are also some dedicated rules for other logical connectives and set operators, but they are not presented in this paper due to space restrictions. This type inference system is aimed to find types for variables of expressions (bound or not) and metavariables of sets, but not for metavariables of predicates which cannot be typechecked using the B typing rules. This is possible if metavariables of sets are not distinguished from variables of expressions, and in the following, a variable will denote either a variable of expression (bound or not), or a metavariable of set.

First, the algorithm assigns a unique type variable to each variable and all these variables with their types are gathered in a typing context Γ. The type inference tree is then built according to the rules of Figure 2. Some constraints may appear during this step due to some non-linearity constraints (for instance, see the rule "$\doteq$"). Once the tree is closed, the algorithm tries to solve the constraints. If it succeeds, the types of variables of Γ are updated with their instantiations.

Before generating the corresponding typing lemma, it is necessary to generate new hypotheses of non-freeness without which this lemma cannot be proved in general, as skolemization cannot be performed when eliminating binding terms. This is realized by means of the following operator:

$$\mathcal{S}_{U,V}(T) = N_1 \rightarrow \ \ldots \ \rightarrow N_n \rightarrow T$$

where T is a reified term, and for all $u \in U$, for all $v \in V$ such that $u \neq v$ and $u \setminus v \notin \mathcal{N}$, then there exists $i \in 1 \ldots n$ such that $N_i = u \setminus v$.

Finally, for a reified rule of the form $\forall x.\mathcal{M}, \mathcal{B}.\mathcal{N} \Rightarrow P$ and from the resulting typing context Γ, the typing lemma can be generated as follows:

$$\forall x.x \in \mathcal{M}, \mathcal{B}, \mathcal{N}.\mathcal{S}_{\mathcal{B},\mathcal{B}\cup\mathcal{M}_\Gamma}(G, H \vdash_\tau \dot{check}\ (P))$$

where: $\mathcal{M} \leftarrow \mathcal{M'}_E \cup \mathcal{M'}_S \cup \mathcal{M}_P$ with $\mathcal{M'}_E \leftarrow \mathcal{M}_E \cup \mathcal{M}_S$ and $\mathcal{M'}_S \leftarrow \mathcal{M}_\Gamma$, where $\mathcal{M}_\Gamma$ is the set of type variables of Γ; G is the reification of the types of Γ such that for all $(v,t) \in \Gamma$ and $v \notin \mathcal{B}$, $\dot{given}\ (t) \in G$ (in which $\dot{given}\ (t)$ means that t is the super-set of itself); H is the reification of the typing context Γ such that for all $(v,t) \in \Gamma$ and $v \notin \mathcal{B}$, $[\![v \in t]\!]_P \in H$.

3.5 Well-Definedness Verification

As said in Section 2, it is pointed out in [2] that conditional definitions may lead to some ill-defined expressions, such as the application of a function to an argument lying outside its domain. Thus, a syntactical filter to be applied to the rule to prove is proposed in [2], and contains all the proof obligations related to well-definedness. The filter is called $\mathcal{L}$ and is defined as a function over reified rules. The computation rules of this function are split into two sets of rules: decomposition and atomic rules. The decomposition rules are the following:

Rules for V

$$\frac{}{x : s, \Gamma \vdash x : s}\ var \qquad \frac{\Gamma \vdash x : s \quad \Gamma \vdash y : t}{\Gamma \vdash x \mapsto y : s \dot{\times} t}\ \dot{\mapsto}_V$$

Rules for E

$$\frac{\Gamma \vdash x : t \quad \Gamma \vdash E : t}{\Gamma \vdash x :\doteq E : S_\tau}\ \doteq \qquad \frac{\Gamma \vdash x :\doteq E : S_\tau \quad \Gamma \vdash F : t}{\Gamma \vdash [x :\doteq E]F : t}\ subst_E$$

$$\frac{\Gamma \vdash x : s \quad \Gamma \vdash y : t}{\Gamma \vdash x \mapsto\ y : s \dot{\times} t}\ \dot{\mapsto}_E \qquad \frac{\Gamma \vdash s : \dot{\mathbb{P}}\,(t)}{\Gamma \vdash\ \dot{choice}\,(s) : t}\ \dot{choice}$$

where S_τ is the type of substitutions.

Rules for S

$$\frac{\Gamma \vdash S : \dot{\mathbb{P}}\,(s) \quad \Gamma \vdash T : \dot{\mathbb{P}}\,(t)}{\Gamma \vdash S \dot{\times} T : \dot{\mathbb{P}}\,(s \dot{\times} t)}\ \dot{\times} \qquad \frac{\Gamma \vdash E : \dot{\mathbb{P}}\,(s)}{\Gamma \vdash \dot{\mathbb{P}}\,(E) : \dot{\mathbb{P}}\,(\dot{\mathbb{P}}\,(s))}\ \dot{\mathbb{P}}$$

$$\frac{x : s, \Gamma \vdash P : P_\tau}{\Gamma \vdash \{x \mid P\} : \dot{\mathbb{P}}\,(s)}\ \dot{\{|\}} \qquad \frac{}{\Gamma \vdash B\dot{I}G : \dot{\mathbb{P}}\,(B\dot{I}G)}\ B\dot{I}G$$

Rules for P

$$\frac{\Gamma \vdash P : P_\tau \quad \Gamma \vdash Q : P_\tau}{\Gamma \vdash P \dot{\wedge} Q : P_\tau}\ \dot{\wedge} \qquad \frac{\Gamma \vdash P : P_\tau \quad \Gamma \vdash Q : P_\tau}{\Gamma \vdash P \dot{\Rightarrow} Q : P_\tau}\ \dot{\Rightarrow}$$

$$\frac{\Gamma \vdash P : P_\tau}{\Gamma \vdash\ \dot{\neg}\, P : P_\tau}\ \dot{\neg} \qquad \frac{\Gamma \vdash x : t \quad \Gamma \vdash P : P_\tau}{\Gamma \vdash\ \dot{\forall}\, x.P : P_\tau}\ \dot{\forall}$$

$$\frac{\Gamma \vdash x :\doteq E : S_\tau \quad \Gamma \vdash P : P_\tau}{\Gamma \vdash [x :\doteq E]P : P_\tau}\ subst_P \qquad \frac{\Gamma \vdash E : t \quad \Gamma \vdash F : t}{\Gamma \vdash E \doteq F : P_\tau}\ \doteq$$

$$\frac{\Gamma \vdash E : t \quad \Gamma \vdash S : \dot{\mathbb{P}}\,(t)}{\Gamma \vdash E \,\dot{\in}\, S : P_\tau}\ \dot{\in}$$

where P_τ is the type of predicates.

Fig. 2. Type Inference Rules

$$\mathcal{L}(\forall x \in \mathcal{M}, \mathcal{B}, \mathcal{N}.P) = \forall x \in \mathcal{M}, \mathcal{B}, \mathcal{N}.\mathcal{L}(P)$$
$$\mathcal{L}(P \mathbin{\dot\wedge} Q) = \mathcal{L}(P) \mathbin{\dot\wedge} (P \mathbin{\dot\Rightarrow} \mathcal{L}(Q)) \qquad \mathcal{L}(P \mathbin{\dot\Rightarrow} Q) = \mathcal{L}(P) \mathbin{\dot\wedge} (P \mathbin{\dot\Rightarrow} \mathcal{L}(Q))$$
$$\mathcal{L}(\dot\neg\, P) = \mathcal{L}(P) \qquad \mathcal{L}(\dot\forall x.P) = \dot\forall x.\mathcal{L}(P)$$

The atomic rules essentially aim to deal with applications of functions and handle atomic predicates, i.e. every predicate other than those considered above. The atomic rules are defined as follows:

$$\mathcal{L}(A) = \dot{true}$$
$$\mathcal{L}(A_{f(E)}) = \dot\exists(s \mapsto t).(f \mathbin{\dot\in} s \mathbin{\dot\to} t \mathbin{\dot\wedge} E \mathbin{\dot\in} \dot{dom}(f)) \mathbin{\dot\wedge} \dot\forall y.(y \doteq f(E) \mathbin{\dot\Rightarrow} \mathcal{L}(A_y))$$
where $y \notin \mathcal{M} \cup \mathcal{B}$, and $\mathcal{B} \leftarrow \mathcal{B} \cup \{y\}$
$$\mathcal{L}(A_{\{x \mid x \mathbin{\dot\in} S \mathbin{\dot\wedge} P\}}) = \dot\forall x.(\mathcal{L}(x \mathbin{\dot\in} S) \mathbin{\dot\wedge} (x \mathbin{\dot\in} S \mathbin{\dot\Rightarrow} \mathcal{L}(P))) \mathbin{\dot\wedge} \dot\forall y.(y \doteq \{x \mid x \mathbin{\dot\in} S \mathbin{\dot\wedge} P\} \mathbin{\dot\Rightarrow} \mathcal{L}(A_y))$$
where $x, y \notin \mathcal{M} \cup \mathcal{B}$, and $\mathcal{B} \leftarrow \mathcal{B} \cup \{x, y\}$

in which an atomic predicate may be of the following form:

1. A, where $f(E), \{x \mid x \mathbin{\dot\in} S \mathbin{\dot\wedge} P\} \notin A$;
2. $A_{f(E)}$, where $f(E) \in A_{f(E)}$, but $g(F), \{x \mid x \mathbin{\dot\in} S \mathbin{\dot\wedge} P\} \notin f, E$;
3. $A_{\{x \mid x \mathbin{\dot\in} S \mathbin{\dot\wedge} P\}}$, where $\{x \mid x \mathbin{\dot\in} S \mathbin{\dot\wedge} P\} \in A_{\{x \mid x \mathbin{\dot\in} S \mathbin{\dot\wedge} P\}}$.

The binding rules, i.e. the rules for atomic predicates of the form (3) must be applied first, before the rule for atomic predicates of the form (2), in order to avoid to eliminate applications of functions under binders.

Compared to [2], our approach relaxes the restrictions over the super-set S in the atomic predicate for comprehension sets, which implies to add the recursive call $\mathcal{L}(x \mathbin{\dot\in} S)$ in the corresponding rule. In addition, we are also able to deal with substitutions (for expressions and predicates), as well as lambda-expressions (we do not provide the corresponding rules here in order to simplify our presentation).

Once this filter has been applied to a reified rule of the form $\forall x \in \mathcal{M}, \mathcal{B}, \mathcal{N}.P$, the well-definedness lemma to be proved is generated as follows:

$$\forall x \in \mathcal{M}, \mathcal{B}, \mathcal{N}.\mathcal{S}_{\mathcal{B}, \mathcal{B} \cup \mathcal{M}_\Gamma}(H \mathbin{\dot\vdash} \mathcal{L}(P))$$

where $\mathcal{M}_S \leftarrow \mathcal{M}_S \cup \mathcal{M}_\Gamma$.

3.6 Rule Verification

Once the proof obligations related to well-definedness have been extracted from the rule by means of a first syntactical filter, it is possible to apply another filter to the rule, which eliminates the conditional definitions unconditionally and produces an equivalent rule simpler to prove. This new filter, which is introduced in [2], is called $\mathcal{E}$, and is defined as a function over reified rules, which unconditionally eliminates all the applications of functions. Considering the $\mathcal{L}$ filter

seen previously, it can be shown that $\mathcal{L}(P) \Rightarrow (P \Leftrightarrow \mathcal{E}(P))$. In the same way as for the $\mathcal{L}$ filter, the computation rules of $\mathcal{E}$ are split into two sets of rules: decomposition and atomic rules. The decomposition rules are the following:

$$\mathcal{E}(\forall x \in \mathcal{M}, \mathcal{B}, \mathcal{N}.P) = \forall x \in \mathcal{M}, \mathcal{B}, \mathcal{N}.\mathcal{E}(P) \qquad \mathcal{E}(P \mathbin{\dot\wedge} Q) = \mathcal{E}(P) \mathbin{\dot\wedge} \mathcal{E}(Q)$$
$$\mathcal{E}(P \mathbin{\dot\Rightarrow} Q) = \mathcal{E}(P) \mathbin{\dot\Rightarrow} \mathcal{E}(Q) \qquad \mathcal{E}(\dot\neg P) = \dot\neg\, \mathcal{E}(P) \qquad \mathcal{E}(\dot\forall x.P) = \dot\forall x.\mathcal{E}(P)$$

The atomic rules are defined as follows:

$$\mathcal{E}(A) = A$$
$$\mathcal{E}(A_{f(E)}) = \dot\forall y.((E, y) \mathbin{\dot\in} f \mathbin{\dot\Rightarrow} \mathcal{E}(A_y)) \text{ where } y \notin \mathcal{M} \cup \mathcal{B}, \text{ and } \mathcal{B} \leftarrow \mathcal{B} \cup \{y\}$$
$$\mathcal{E}(A_{\{x \mid x \mathbin{\dot\in} S \mathbin{\dot\wedge} P\}}) = \dot\forall y.(y = \{x \mid x \mathbin{\dot\in} S \mathbin{\dot\wedge} \mathcal{E}(P)\} \mathbin{\dot\Rightarrow} \mathcal{E}(A_y))$$
where $y \notin \mathcal{M} \cup \mathcal{B}$, and $\mathcal{B} \leftarrow \mathcal{B} \cup \{y\}$

Once this filter has been applied to a reified rule of the form $\forall x \in \mathcal{M}, \mathcal{B}, \mathcal{N}.P$, the rule lemma to be proved is generated in the following way:

$$\forall x.\mathcal{M}, \mathcal{B}, \mathcal{N}.\mathcal{S}_{\mathcal{B}, \mathcal{B} \cup \mathcal{M}_\Gamma}(H \mathbin{\dot\vdash} \mathcal{E}(P))$$

where $\mathcal{M}_S \leftarrow \mathcal{M}_S \cup \mathcal{M}_\Gamma$ with $\mathcal{M}_\Gamma$ the set of the type variables of the typing context Γ, and where H is the reification of Γ.

3.7 Examples

In the following, we describe two examples of rule verification using BCARe.

Example 3 (Verification of ForAllX.3). This rule is a deduction rule, and there is no need to verify that there is no variable capture. As there is no application function, only the typing and rule lemmas are generated as follows:

Lemma $type\_ForAllX3 : \mathbf{forall}\ t : \dot S,\ \mathbf{forall}\ A\ a : \dot V,$
$a \mathbin{\dot\setminus} (A, t) \to given(t),\ A \mathbin{\dot\subseteq} t \mathbin{\dot\vdash}_\tau check(A \doteq \dot\emptyset \mathbin{\dot\Rightarrow} \dot\forall a.a \mathbin{\dot\notin} A).$

Lemma $rule\_ForAllX3 : \mathbf{forall}\ A\ t : \dot S,\ \mathbf{forall}\ a : \dot V,$
$a \mathbin{\dot\setminus} (A, t) \to A \mathbin{\dot\subseteq} t \mathbin{\dot\vdash} (A \doteq \dot\emptyset \mathbin{\dot\Rightarrow} \dot\forall a.a \mathbin{\dot\notin} A).$

Example 4 (Verification of SimplifyRelDorXY.2). This rule is a rewrite rule, and we must verify that the guards correctly protect the free variables. As the elements of Q do not belong to $\{f, u, v, a\}$, the criterion (2) is verified. The three lemmas are then generated in the following way:

Lemma $type\_SimplifyRelDorXY\_2 :$
$\mathbf{forall}\ t_1\ t_2 : \dot S,\ \mathbf{forall}\ a\ f\ u\ v : \dot V,$
$given(t_1),\ given(t_2),\ u \mathbin{\dot\in} \dot{\mathbb{P}}(t_2),\ v \mathbin{\dot\in} \dot{\mathbb{P}}(t_1),\ f \mathbin{\dot\in} \dot{\mathbb{P}}(t_2 \mathbin{\dot\times} t_1),\ a \mathbin{\dot\in} t_2 \mathbin{\dot\vdash}_\tau$
$check(f \mathbin{\dot\in} u \mathbin{\dot\nrightarrow} v \mathbin{\dot\wedge} a \mathbin{\dot\in} dom(f) \mathbin{\dot\Rightarrow} \{a\} \mathbin{\dot\lhd} f \doteq \{a \mapsto f(a)\}).$

Lemma $wdef\_SimplifyRelDorXY\_2$:
forall $f\ t_1\ t_2\ u\ v : \dot{S}$, **forall** $a : \dot{E}$, **forall** $s\ t : \dot{V}$,
$s \setminus (t, f, u, v, a, t_1, t_2) \to t \setminus (s, f, u, v, a, t_1, t_2) \to$
$u \in \dot{\mathbb{P}}(t_2),\ v \in \dot{\mathbb{P}}(t_1),\ f \in \dot{\mathbb{P}}(t_2 \times t_1),\ a \in t_2 \vdash$
$f \in u \pfun v \wedge a \in dom(f) \Rightarrow \exists(s \mapsto t).(f \in s \pfun t \wedge a \in dom(f))$.

Lemma $rule\_SimplifyRelDorXY\_2$:
forall $f\ t_1\ t_2\ u\ v : \dot{S}$, **forall** $a : \dot{E}$, **forall** $y : \dot{V}$,
$y \setminus (f, u, v, t_1, t_2, a) \to u \in \dot{\mathbb{P}}(t_2),\ v \in \dot{\mathbb{P}}(t_1),\ f \in \dot{\mathbb{P}}(t_2 \times t_1),\ a \in t_2 \vdash$
$f \in u \pfun v \wedge a \in dom(f) \Rightarrow \forall y.((a, y) \in f \Rightarrow \{a\} \lhd f = \{a \mapsto y\})$.

4 Automated Verification of Proof Rules

In this section, we discuss some solutions that we have provided to automate the verification of proof rules in the framework of the BCARe environment. In particular, the several solutions aim to automatically prove the different lemmas generated in Coq from proof rules by BCARe. To do so, we have developed a set of tactics using the $\mathcal{L}_{\text{tac}}$ tactic language of Coq [11]. In the specific case of the rule lemma, we have also considered an alternative approach based on an external ATP called Zenon [5]. Both approaches are able to deal with rules involving all the set operators defined before the functional abstraction (introduction of anonymous functions) in the B-Book [1].

4.1 Verification using $\mathcal{L}_{\text{tac}}$

To deal with the different lemmas generated in Coq from proof rules by the BCARe environment (see Section 3), a set of tactics has been developed using the $\mathcal{L}_{\text{tac}}$ tactic language of Coq [11]. Regarding the proof of the typing lemma, we have designed a correct and complete tactic (the B typechecking is decidable), which essentially performs pattern-matching over the goal in order to select the appropriate B typing rules. As for the proof of well-definedness lemma, we have written another tactic, which is able to manage only specific cases. This tactic mostly looks for instantiations which allow us to complete the proof using a direct propositional combination of the current hypotheses. Finally, the proof of the rule lemma is handled by means of a tactic which relies on a naive and incomplete heuristic, even though it succeeds in proving about 200 derived rules of the B-Book. This heuristic mainly consists in only considering Skolem symbols when instantiating (no unification is performed), while right contraction is never used. In addition, this tactic has also some efficiency issues in some cases that can be observed in Subsection 4.3. To palliate these several drawbacks, we have developed an alternative approach based on the use of an external ATP, able to provide a more powerful and efficient proof search procedure, and able to be easily interfaced with Coq (i.e. producing proof traces that can be exploited).

4.2 Verification using Zenon

As an alternative approach to $\mathcal{L}_{\text{tac}}$ tactics, we have developed an interface with the Zenon ATP [5], in order to prove the rule lemmas in particular. One of the main difficulties when using an external ATP is to bring together the B set theory and the ATP logic. As seen previously, the B set theory [1] is actually based on a simplification of classical set theory. As for Zenon, it relies on the classical first order logic with equality (using the tableau method as proof search), and does not deal explicitly with the set theory. The idea consists in normalizing the formula to be proved (unfolding definitions), in order to obtain a first order logic formula containing only the "$\dot{\in}$" (reified) set operator. This formula is then syntactically interpreted within the ATP logic, in which the "$\dot{\in}$" operator is considered as a regular uninterpreted predicate symbol.

Another difficulty is to ensure the correctness of the external deduction. We have adopted a skeptical approach by building B proofs from the proofs produced by the ATP. In this way, it is possible to check the validity of these proofs that have been automatically found. However, this requires the ATP proof traces to be comprehensible enough so as to allow us to reconstruct proofs. This is the case of Zenon, which produces several proof traces at different levels. In particular, it produces Coq proofs, which can be used to build proofs within the BCoq embedding of BCARe (see Section 3). To do so, the Coq proofs generated by Zenon have to be (re)reified. This translation is syntactical and relies on an embedding of each tactic occurring in the Coq proofs produced by Zenon.

4.3 Benchmarks

In the following, we present the results of our implementation using Zenon on several examples of proof rules. This implementation actually consists of a Coq tactic written in OCaml. We also compare these results to those obtained using the corresponding $\mathcal{L}_{\text{tac}}$ tactic. To realize these benchmarks, we have tested both tactics on derived rules of the B-Book, as well as on several added rules coming from Atelier B and the database maintained by Siemens SAS I MO.

Regarding derived rules, we have considered about 200 rules and the results of the tests (run on an Intel Pentium D 3.40GHz/4GB computer) are summarized in the graph of Figure 3, where a point represents the test of a proof rule and where the x-axis and y-axis respectively correspond to the $\mathcal{L}_{\text{tac}}$ and Zenon proof times (expressed in seconds). In this graph, we only consider derived rules for which the proof times for $\mathcal{L}_{\text{tac}}$ and Zenon are less than 30s (this corresponds to about 66% of the tested rules). We can see that Zenon is faster than the $\mathcal{L}_{\text{tac}}$ tactic for the most part of the tested rules (for 71% of these rules, more precisely). Furthermore, over the 200 rules for which Zenon succeeds in finding a proof, 15 rules cannot be proved using the $\mathcal{L}_{\text{tac}}$ tactic. For the sake of scalability, we have also tested our tactics on added rules coming from Atelier B and the database maintained by Siemens SAS I MO. We have selected 1279 rules (over a total of 5039 rules) within the scope of both tactics. For these rules, Zenon can prove 813 rules (64%), whereas the $\mathcal{L}_{\text{tac}}$ tactic manages to prove 498 rules (39%).

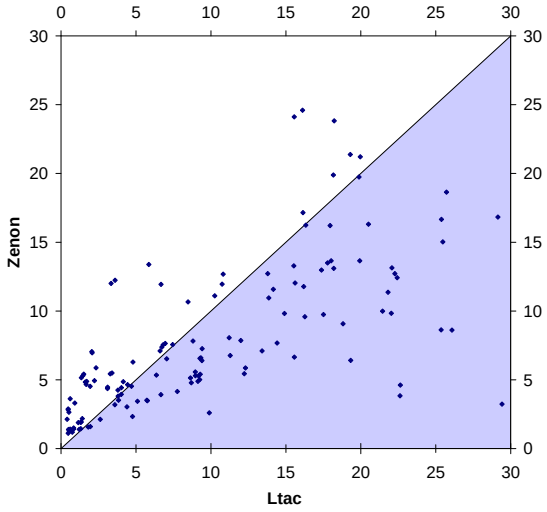

Fig. 3. Proof Times of Rule Lemmas using Zenon and $\mathcal{L}_{\text{tac}}$

In addition, the larger the $\mathcal{L}_{\text{tac}}$ proof time is, the larger is the number of rules for which Zenon is faster than $\mathcal{L}_{\text{tac}}$. These several experimental results tend to show that the use of Zenon is an approach which is not only more satisfactory than that of $\mathcal{L}_{\text{tac}}$, but also very promising in terms of scalability.

5 Conclusion

We have proposed a formal and mechanized framework which allows us to verify proof rules of the B method, and which is able to use an external automated theorem prover called Zenon. This framework relies on the BCARe set of tools, developed by Siemens SAS I MO, which provides a deep embedding of the B theory within the logic of the Coq proof assistant and allows us to automatically generate the required properties to be checked for a given proof rule. Currently, this tool chain is able to automatically verify about 200 derived rules of the B-Book, as well as 800 added rules coming from Atelier B and the rule database maintained by Siemens SAS I MO.

As future work, we first aim to completely verify the derived rules of the B-Book. The BCARe environment is already able to deal with all these derived rules, but the automated verification part (using Zenon) has to be adapted. In particular, this part has to be extended to manage proofs of properties involving applications of functions, substitutions, arithmetics, induction, and sequences. It seems clear that all the proofs will not be able to be automated, and our goal consists in automating at least a large part of them and characterizing the lack of automation for the other proofs. To palliate this potential lack of automation,

we could consider alternative ATPs (other than Zenon) or SMT solvers, which might be more appropriate for some specific properties. In this case, we should develop a verification platform able to use several provers and solvers. Once these derived rules have been verified, we plan to deal with the rest of the added rules of Atelier B (about 1,400 rules), and thereafter the rest of those of the database developed by Siemens SAS I MO (about 3,100 rules). If the latter focuses on the development of applications, the former consists in certifying Atelier B as a tool used in a safety-critical and high-integrity chain of production.

References

1. J.-R. Abrial. *The B-Book, Assigning Programs to Meanings*. Cambridge University Press, Cambridge (UK), 1996. ISBN 0521496195.
2. J.-R. Abrial and L. Mussat. On Using Conditional Definitions in Formal Theories. In *Formal Specification and Development in Z and B (ZB)*, volume 2272 of *LNCS*, pages 317–322, Grenoble (France), Jan. 2002. Springer.
3. K. Berkani, C. Dubois, A. Faivre, and J. Falampin. Validation des règles de base de l'Atelier B. *Technique et Science Informatiques (TSI)*, 23(7):855–878, 2004.
4. J.-P. Bodeveix, M. Filali, and C. Muñoz. A Formalization of the B-Method in Coq and PVS. In *B Users Group Meeting*, Toulouse (France), Sept. 1999.
5. 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.
6. P. Chartier. Formalisation of B in Isabelle/HOL. In *B Conference*, volume 1393 of *LNCS*, pages 66–82, Montpellier (France), Apr. 1998. Springer.
7. H. Cirstea and C. Kirchner. Using Rewriting and Strategies for Describing the B Predicate Prover. In *Strategies in Automated Deduction*, pages 25–36, Lindau (Germany), July 1998.
8. ClearSy. *Atelier B 4.0*, Feb. 2009. http://www.atelierb.eu/.
9. J.-F. Couchot, F. Dadeau, D. Déharbe, A. Giorgetti, and S. Ranise. Proving and Debugging Set-Based Specifications. In *Workshop on Formal Methods*, volume 95 of *ENTCS*, pages 189–208, Campina Grande (Brazil), Oct. 2003. Elsevier.
10. D. Déharbe. Automatic Verification for a Class of Proof Obligations with SMT-Solvers. In *Abstract State Machines, Alloy, B and Z (ABZ)*, volume 5977 of *LNCS*, pages 217–230, Orford (Canada, QC), Feb. 2010. Springer.
11. D. Delahaye. A Tactic Language for the System Coq. In *Logic for Programming and Automated Reasoning (LPAR)*, volume 1955 of *LNCS/LNAI*, pages 85–95, Reunion Island (France), Nov. 2000. Springer.
12. É. Jaeger and C. Dubois. Why Would You Trust B? In *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 4790 of *LNCS/LNAI*, pages 288–302, Yerevan (Armenia), Oct. 2007. Springer.
13. É. Le Lay. Automatiser la validation des règles. Master's thesis, INSA (Rennes), Siemens SAS I MO, Sept. 2008.
14. L. Mikhailov and M. Butler. An Approach to Combining B and Alloy. In *Formal Specification and Development in Z and B (ZB)*, volume 2272 of *LNCS*, pages 140–161, Grenoble (France), Jan. 2002. Springer.
15. The Coq Development Team. *Coq, version 8.3*. INRIA, Oct. 2010. http://coq.inria.fr/.