Discharging Proof Obligations from Atelier B Using Multiple Automated Provers*

David Mentré[1], Claude Marché[2,3],
Jean-Christophe Filliâtre[3,2], and Masashi Asuka[4]

[1] Mitsubishi Electric R&D Centre Europe, Rennes, F-35708
[2] INRIA Saclay – Île-de-France, Orsay, F-91893
[3] Lab. de Recherche en Informatique, Univ Paris-Sud, CNRS, Orsay, F-91405
[4] Advanced Technology R&D Center, Mitsubishi Electric Corp., Amagasaki, Japan

Abstract. We present a method to discharge proof obligations from Atelier B using multiple SMT solvers. It is based on a faithful modeling of B's set theory into polymorphic first-order logic. We report on two case studies demonstrating a significant improvement in the ratio of obligations that are automatically discharged.

1 Introduction

The B Method [1] is a formal approach to develop safety critical embedded systems. It is mainly used in the European railway industry [2,5]. This method allows the design of correct-by-construction programs, thanks to refinement techniques. The soundness of refinement steps is expressed by logic formulas, called *proof obligations* (PO for short), that must be proved valid. The system *Atelier B* implements the B Method and provides a dedicated theorem prover. It is mostly an automated prover for B's set theory. To discharge POs that are not proved automatically, a user interface allows interactive proof steps.

In recent years, there has been tremendous progress in the domain of *Satisfiability Modulo Theories* (SMT for short). Some SMT solvers have proved powerful in the context of extended static checking, *e.g.* Simplify for ESC/Java, Z3 for Boogie, Spec#, and VCC. A natural question is whether we would gain automation by using SMT solvers on POs generated by Atelier B. This is the question we address in this paper. We propose a technique to translate B POs into the input language of Why3 [6], an environment providing a common front-end to various external provers. Why3 implements a polymorphic first-order logic, in which we axiomatize B's set theory. A main difficulty is to make sure that this axiomatization is in a suitable form for the SMT provers to solve the generated goals.

This paper is organized as follows. Sect 2 presents the necessary background regarding B and Why3. Sect 3 exposes our technique to perform the translation from B to Why3. Sect 4 reports on experiments made with our implementation. We compare with related work in Sect 5.

* This work is partly funded by the U3CAT project (ANR-08-SEGI-021, `http://frama-c.com/u3cat/`) of the French national research organization (ANR).

```
MACHINE Timer(initial_timer_value_ms)

SEES Configuration

CONSTRAINTS initial_timer_value_ms ∈ NAT1

VARIABLES  active, remaining_time

INVARIANT active ∈ 𝔹 ∧ remaining_time ∈ NAT ∧
          (active = FALSE ⇒ remaining_time = 0) ∧
          (active = TRUE ⇒ remaining_time ≤ initial_timer_value_ms)

INITIALISATION  active := FALSE || remaining_time := 0

OPERATIONS
   start_timer = PRE active = FALSE THEN
      active := TRUE || remaining_time := initial_timer_value_ms
   END;

   decrement_timer = PRE active = TRUE THEN
      remaining_time : ( remaining_time ∈ NAT ∧
         (remaining_time$0 ≥ cycle_duration
            ⇒ remaining_time = remaining_time$0 − cycle_duration) ∧
         (remaining_time$0 < cycle_duration ⇒ remaining_time = 0) )
   END;
END
```

Fig. 1. Abstract State Machine defining a timer using B Method

2 Background

2.1 The B Environment

The B Method is organized around Abstract State Machines. Each Abstract State Machine contains a state defined through variables as well as operations allowing to modify this state. One can use Booleans, integers, and set theory to express the state of an abstract machine. For example, in Fig. 1 showing a timer defined using B Method, the state is defined through Boolean variable active and natural integer variable remaining_time. The two operations start_timer and decrement_timer allow the use of this timer by updating those variables.

Correctness properties that should be fulfilled by a machine are defined in an invariant of each machine as well as in the definition of each operation. One can use first-order logic to express those properties. In Fig. 1, the **INVARIANT** clause states that if the timer is not active, the remaining_time should be zero otherwise the remaining time should be less or equal the initial timer value. In a similar way, the specification of the decrement_timer operation states that this operation recomputes the remaining_time variable. If the value of the variable at

$$
\begin{aligned}
&\mathsf{cycle\_duration} = 100 \land \\
&(\mathsf{active} = \mathbf{TRUE} \Rightarrow \mathsf{remaining\_time} \leq \mathsf{initial\_timer\_value\_ms}) \land \\
&\mathsf{active} = \mathbf{TRUE} \land\ 1 \leq \mathsf{initial\_timer\_value\_ms} \land \\
&\mathsf{remaining\_time\$1} \in \mathbb{Z} \land 0 \leq \mathsf{remaining\_time\$1} \land \mathsf{remaining\_time\$1} \leq 2147483647 \land \\
&(\mathsf{cycle\_duration} \leq \mathsf{remaining\_time} \\
&\quad \Rightarrow \mathsf{remaining\_time\$1} = \mathsf{remaining\_time} - \mathsf{cycle\_duration}) \land \\
&(\mathsf{remaining\_time} + 1 \leq \mathsf{cycle\_duration} \Rightarrow \mathsf{remaining\_time\$1} = 0) \\
\Rightarrow\ &\mathsf{remaining\_time\$1} \leq \mathsf{initial\_timer\_value\_ms}
\end{aligned}
$$

Fig. 2. Example of Proof Obligation

operation entry ($0 notation) is bigger than the cycle duration, then it should be decreased by the amount of cycle duration, otherwise it should be zero. Moreover, those operations are constrained by a precondition that ensures the start_timer operation is only used when the timer is inactive while the decrement_timer operation is only used when the timer is active.

Abstract State Machines are similar to formal specifications. They are transformed into an actual implementation through the use of manual refinements that lead in one or more steps to an implementation. An implementation might import one or more other machines in order to use their operations.

The B Method ensures that correctness properties defined in the invariant or the operations are kept through the refinements and up to the final implementation. This is done through the generation of POs, following patterns defined in the B-Book [1], that must be proved valid. For example, the PO shown in Fig. 2 checks that the invariant $\mathsf{active} = \mathbf{TRUE} \Rightarrow \mathsf{remaining\_time} \leq \mathsf{initial\_timer\_value\_ms}$ is preserved by the decrement_timer operation. The upper part of this PO describes the effect of the operation specification (here used as an assumption for this PO), while the lower part being the property to prove under $\mathsf{active} = \mathbf{TRUE}$ assumption. The $1 notation denotes the state of the variable after execution of the operation.

Tools are available to use the B Method in an industrial context, like Atelier B made by ClearSy company. This tool contains an editor as well as automatic and interactive provers. When developing software using the B Method, the code corresponding to specifications, refinements and implementations is entered into Atelier B. Then proof obligations are automatically generated and in a second step are proved, either automatically or under user's guidance. The amount of interactive proofs is a direct cost for a project and usually corresponds to 5% to 40% of the total amount of proof obligations for industrial projects. The PO shown in Fig. 2 is not proved by the automatic prover of Atelier B.

2.2 The Why3 System

Why3 [6] is a set of tools for program verification. Basically, it is composed of two parts, which are depicted in Fig. 3: a logical language called Why with an infrastructure to translate it to existing theorem provers; and a programming

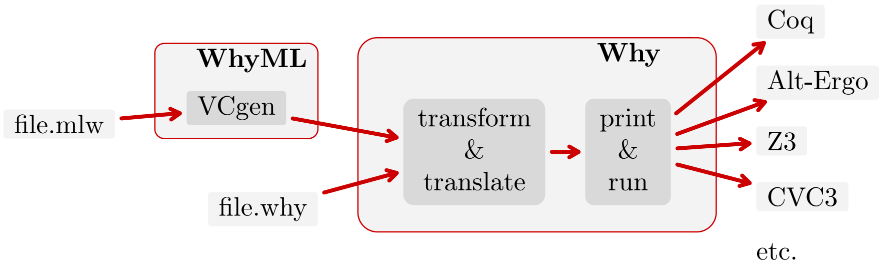

Fig. 3. Overview of Why3

```
goal g: forall active:bool,
   remaining_time remaining_time_1 initial_timer_value_ms cycle_duration : int.
   cycle_duration = 100 ∧ active = True ∧
   (active = True → remaining_time ≤ initial_timer_value_ms) ∧
   1 ≤ initial_timer_value_ms ∧
   0 ≤ remaining_time_1 ∧ remaining_time_1 ≤ 2147483647 ∧
   (cycle_duration ≤ remaining_time
       → remaining_time_1 = remaining_time − cycle_duration) ∧
   (remaining_time + 1 ≤ cycle_duration → remaining_time_1 = 0)
 → remaining_time_1 ≤ initial_timer_value_ms
```

Fig. 4. The same Proof Obligation as Fig. 2, in Why3

language called WhyML with a verification condition generator. In this paper, we are not using the programming facilities of Why3; we are only concerned with its logic, that is the right part of Fig. 3.

The logic of Why3 is a polymorphic first-order logic with recursive definitions, algebraic data types, and inductive predicates [7]. Logical declarations are organized in small units called *theories*. The purpose of Why3 is, among other things, to extract goals from theories and to translate them to the native language of external theorem provers. Such provers range from interactive proof assistants, such as Coq, to general-purpose automated theorem provers, such as Alt-Ergo, Z3, or CVC3, and even to dedicated theorem provers, such as Gappa.

Fig. 4 shows a Why3 file containing one goal, equivalent to the PO of Fig 2. Using Why3, this goal is proved valid with any of Alt-Ergo [12], Z3 [13], or CVC3 [4].

3 A Translator from B to Why3

This section details the core of our contribution: a method to translate B proof obligations into the Why3 form, so as to call the various provers available as Why3 back-end. The method is based on two components: first a modeling in Why3 of the set theory used in B (Sect. 3.1 below), second a standalone tool

```
theory Set
  type set α                                          (* abstract type for polymorphic sets *)

  predicate mem α (set α)                                                    (* membership *)

  predicate (==) (s1 s2: set α) = forall x : α. mem x s1 ↔ mem x s2          (* equality *)
  axiom extensionality: forall s1 s2: set α. s1 == s2 → s1 = s2

  predicate subset (s1 s2: set α) = forall x : α. mem x s1 → mem x s2       (* inclusion *)

  function empty : set α                                                      (* empty set *)
  axiom empty_def: forall x: α. ¬ (mem x empty)

  function union (set α) (set α) : set α                                          (* union *)
  axiom union_def: forall s1 s2: set α, x: α.
    mem x (union s1 s2) ↔ mem x s1  ∨  mem x s2
[...]
end
```

Fig. 5. Why3 theory of sets (excerpt)

that reads a B file containing proof obligations and translates it into a set of equivalent Why3 goals (Sect. 3.2). Then in Sect. 3.3 we discuss the soundness of this method.

3.1 Modeling B Set Operators as Why3 Theories

The first theory we pose is a theory of sets. An excerpt of it is shown on Fig. 5. To model the different possible types of elements, we make use of the type polymorphism of Why3, and thus declare a polymorphic type set α where the type parameter α denotes the type of elements. The type set is not defined in Why3 but only *axiomatized*. The first and essential ingredient of this axiomatization is the predicate mem which is intended to denote membership of an element in a set. Indeed, most of the other operators that we introduce afterwards are axiomatized with respect to mem, as exemplified in Fig. 5 for the (polymorphic) empty set, the union operator and the predicate subset.

In the POs generated by B, it is very common to test equality of two sets. In Why3, the built-in symbol = denotes a polymorphic equality, which is assumed to be a congruence relation on any type it is used on. However, for sets, the intended equality is not arbitrary: we want to model the fact that two sets are equal if and only if they contain the same elements (Axiom SET 4 of the B-Book [1, p. 61]). This is done by defining the predicate == of Fig. 5 just as said above, and posing an axiom of *extensionality* which states that sets that are equivalent for == are equal.

Both to exemplify our model of sets, and to define commonly used sets of integers in B, let's show how we model intervals of integers. This is done in

```
theory Interval
  use export int.Int
  use export Set

  function mk int int : set int
  axiom mem_interval: forall x a b : int. mem x (mk a b) ↔ a ≤ x ≤ b

  function integer : set int
  axiom mem_integer: forall x:int.mem x integer

  function natural : set int
  axiom mem_natural: forall x:int. mem x natural ↔ x ≥ 0
[...]
end
```

Fig. 6. Why3 theory of intervals

a new Why3 theory, importing those of sets, as shown in Fig. 6. We declare a logic function mk such that mk a b denotes the interval $[a, b]$. We also pose definitions of the B built-in sets $\mathbb{Z}$ and $\mathbb{N}$ as two constants integer and natural with appropriate axioms. We reuse Why3 computer arithmetic operators which are the same as B-Book's ones. Other set constructs are axiomatized in a similar way: relations, power sets, sequences, finite sets, etc.

We detail our model of B *relations*, as shown in Fig. 7. A relation between two sets S and T is just a set of pairs of elements of $S \times T$. Domain and range of such a relation are axiomatized with natural axioms. Partial functions in B are just particular cases of relations. The set of partial functions on some sets s and t is axiomatized in the Function theory of Fig. 7. Our axioms are designed as transcriptions of those of the B-Book [1, p. 86], independently of the case studies. We also provide a few lemmas about functions. These were added while working on the case studies. They provide a form of hint to the SMT solvers. Unlike axioms, these are logical consequences of the axiomatization. They are proved, using Why3, either automatically with SMT solvers or interactively with Coq.

The set of total functions is defined similarly. A non-trivial construct of B is function application f(x). In B, this construct is subject to the condition x $\in$ dom(f) [1, p. 89]. We model this construct in Why3 using an explicit operator apply. It is axiomatized for total functions only (see the last two axioms in Fig. 7) and unspecified otherwise.

3.2 The Translation Process

Addition of the Why3 proof tool chain inside Atelier B is made after generation of proof obligations. For each B machine (specification, refinement, or implementation), Atelier B generates an internal PO file (with suffix .po). We read and translate this PO file into Why3.

```
theory Relation "Relations between two sets"
  use export Set

  type rel α β = set (α,β)

  function dom (rel α β) : set α
  axiom dom_def: forall r : rel α β, x : α. mem x (dom r) ↔ exists y : β. mem (x,y) r

  function ran (rel α β) : set β
  axiom ran_def: forall r : rel α β, y : β. mem y (ran r) ↔ exists x : α. mem (x,y) r
[...]
end

theory Function "Partial functions as relations"
  use export Relation

  function (+->) (s:set α) (t:set β) : set (rel α β)
  axiom mem_function: forall f:rel α β, s:set α, t:set β.
    mem f (s +-> t) ↔
       (forall x:α, y:β. mem (x,y) f → mem x s ∧ mem y t) ∧
       (forall x:α, y1 y2:β. mem (x,y1) f ∧ mem (x,y2) f → y1=y2)

  lemma range_function: forall f:rel α β, s:set α, t:set β, x:α, y:β.
    mem f (s +-> t) → mem (x,y) f → mem y t

  lemma function_extend_range: forall f:rel α β, s:set α, t u:set β.
    subset t u → mem f (s +-> t) → mem f (s +-> u)

  function (-->) (s:set α) (t:set β) : set (rel α β)
  axiom mem_total_functions: forall f:rel α β, s:set α, t:set β.
    mem f (s --> t) ↔ mem f (s +-> t) ∧ dom f == s

  lemma total_function_is_function: forall f:rel α β, s:set α, t:set β.
    mem f (s --> t) → mem f (s +-> t)

  function apply (rel α β) α : β
  axiom apply_def1: forall f:rel α β, s:set α, t:set β, a:α.
    mem a s ∧ mem f (s --> t) → mem (a, apply f a) f
  axiom apply_def2: forall f:rel α β, s:set α, t:set β, a:α, b:β.
    mem f (s --> t) ∧ mem (a,b) f → b = apply f a
[...]
end
```

Fig. 7. Why3 theory of relations and functions (excerpt)

Our bpo2why translator is made of three steps: the parsing of Atelier B's PO file into an abstract syntax tree, the application of a type inference algorithm on the read tree, and finally the translation of the typed tree into Why3.

```
THEORY ProofList IS
  _f(1) ∧ _f(2) ∧ _f(6) ∧ decrement_timer.2,(_f(10) ⇒ _f(11));
[...]
END
∧
THEORY Formulas IS
1  ("‘Component constraints’" ∧ initial_timer_value_ms ∈ ℤ ∧
   0 ≤ initial_timer_value_ms ∧ initial_timer_value_ms ≤ 2147483647 ∧
   ¬(initial_timer_value_ms = 0) ∧ cycle_duration = 100;
2  ("‘Component invariant’" ∧ active ∈ 𝔹 ∧ remaining_time ∈ ℤ ∧
   0 ≤ remaining_time ∧ remaining_time ≤ 2147483647 ∧
   (active = FALSE ⇒ remaining_time = 0) ∧
   (active = TRUE ⇒ remaining_time ≤ initial_timer_value_ms));
[...]
6  ("‘decrement_timer preconditions in this component’" ∧ active = TRUE);
[...]
10  ("‘Local hypotheses’" ∧ remaining_time$1 ∈ ℤ ∧ 0 ≤ remaining_time$1 ∧
    remaining_time$1 ≤ 2147483647 ∧
    (cycle_duration ≤ remaining_time ⇒
          remaining_time$1 = remaining_time − cycle_duration) ∧
    (remaining_time + 1 ≤ cycle_duration ⇒ remaining_time$1 = 0));
11  (remaining_time$1 ≤ initial_timer_value_ms)
END
∧
THEORY EnumerateX IS
  t_BOOM_MOVEMENT_ORDER = {go_up, go_down}
END
```

Fig. 8. Part of proof obligation file generated for Timer machine

The parsing step is quite usual. The format of the PO file is not publicly documented but it is generated as a text file and we have reverse-engineered it. Fig. 8 shows part of the generated PO file for the Timer machine of Fig. 1. This file contains three parts: a set of logic expressions to prove (ProofList part), a set of formulas identified by their sequence number (Formulas part) and referred as _f(n) in previous logic expressions, and a set of enumerated sets (EnumerateX part). We build an abstract syntax tree from the content of this file, using the same priority and associativity as B's operators [9]. As the B syntax is quite big (about 200 keywords and operators), we currently do not parse all of it but a significant subset[1] needed for our tests.

The type inference step decorates the abstract syntax tree with the B type of all operators and identifiers. It is necessary for a precise translation in the

[1] This subset includes $\exists$ and $\forall$ quantifiers, Boolean expressions (with $\Rightarrow$, $\Leftrightarrow$, $\wedge$, $\vee$, $\neg$ connectors and bool operator), usual integer arithmetic expressions ($+$, $-$, $*$, $/$ and *mod* operators, $<$, $\leq$, $\geq$, $>$ comparison operators, 32 bits constants), set expressions (with $\mathbb{P}$, a..b, $\in$, $*$, $\cap$, $\cup$ and $-$ set operators), $\mathbb{Z}$, $\mathbb{N}$ and $\emptyset$ sets, operators on functions and relations (including seq, f^{-1}, $\leftrightarrow$, $\twoheadleftarrow$, $\rightarrow$, f[s], f(x), dom, ran, size).

```
theory B_translation
  use import bool.Bool
  use import int.Int
  use import bpo2why_prelude.Interval
[...]
  type enum_t_BOOM_MOVEMENT_ORDER = E_go_up | E_go_down
[...]
  predicate f1 (v_remaining_time_1: int) (v_remaining_time: int)
             (v_initial_timer_value_ms: int) (v_cycle_duration: int) (v_active: bool) =
    ((((mem v_initial_timer_value_ms integer)) ∧ (0 ≤ v_initial_timer_value_ms))
    ∧ (v_initial_timer_value_ms ≤ 2147483647))
    ∧ ¬(v_initial_timer_value_ms = 0)) ∧ (v_cycle_duration = 100))

  predicate f2 [...] = ((((((mem v_remaining_time integer)) ∧ (0 ≤ v_remaining_time))
    ∧ (v_remaining_time ≤ 2147483647))
    ∧ ((v_active = False) → (v_remaining_time = 0)))
    ∧ ((v_active = True) → (v_remaining_time ≤ v_initial_timer_value_ms)))
[...]
  predicate f6 [...] = (v_active = True)
[...]
  predicate f10 [...] = ((((mem v_remaining_time_1 integer))
    ∧ (0 ≤ v_remaining_time_1)) ∧ (v_remaining_time_1 ≤ 2147483647))
    ∧ ((v_cycle_duration ≤ v_remaining_time)
        → (v_remaining_time_1 = (v_remaining_time − v_cycle_duration))))
    ∧ (((v_remaining_time + 1) ≤ v_cycle_duration) → (v_remaining_time_1 = 0)))

  predicate f11 [...] = (v_remaining_time_1 ≤ v_initial_timer_value_ms)

  goal decrement_timer_2 :
    forall v_remaining_time_1: int, v_remaining_time: int,
      v_initial_timer_value_ms: int, v_cycle_duration: int, v_active: bool.
    ((f1 v_remaining_time_1 v_remaining_time v_initial_timer_value_ms v_cycle_duration v_active)
     ∧ (f2 [...]) ∧ (f6 [...]) ∧ (f10 [...]))
    →
    (f11 [...])
[...]
end
```

Fig. 9. Why3 translation of Timer proof obligation

next step. We use a classical Hindley-Milner type inference algorithm [16]. An additional issue is to support operator overloading, *e.g.* "$*$" which is both the arithmetic multiplication and the Cartesian product of two sets.

In a third step, we translate the typed abstract syntax tree into a Why3 file. This is done through a top-down traversal of the tree, translating each node into Why3 syntax and then recursively translating sub-trees of this node. This translation step uses the Why3 theories of B operators defined in Section 3.1. In case operators would have several possible translations, we use the inferred type in previous step to determine the kind of Why3 operator to use. For example, the "=" B's operator is translated into Why3's "=" if it is an integer equality or into Why3's "==" operator if it is a set equality. Enumerated sets are translated into Why3's sum types. All B's expressions in a PO file are translated, except two kinds related to enumerated sets (an enumerated set is not empty and an enumerated set is finite) as those assumptions are implicitly guaranteed by Why3's sum types. Fig. 9 shows the PO file of Fig.8 translated into Why3. We have kept the same structure as the input file, with the definition of "fn" predicates and their use in a Why3's "**goal**". All predicates are quantified over all variables used in the PO file. The t_BOOM_MOVEMENT_ORDER enumerated set is translated into a sum type. We have used an explicit parenthesizing of expressions to avoid any priority issue. We keep the PO comments produced by Atelierb B as Why3 labels. Thanks to our modeling of B operators, we are able to translate set related expressions. For example in predicate f2 of Fig. 9, we translate the PO expression "remaining_time $\in \mathbb{Z}$" into "mem v_remaining_time integer", using the mem set operator defined in Sect. 3.1. In the same way, the symbol "integer" is the one of Fig. 6.

By default, we generate a Why3 file for each original PO file. However, when a PO file contains more than 200 proof obligations, we split the generated Why3 file into several files, each one containing at most 200 goals. We also include in those files only the "fn" predicates needed by goals of a given Why3 file. This approach reduces the processing time and proof context of Why3 under acceptable limits, as well as the time needed to call provers. Otherwise a single Why3 file with 1,600 goals and 1,400 predicates would take several minutes to simply load the file.

3.3 Soundness of the Translation

We claim that our translation process is sound in the sense that if the translation of a B proof obligation is a valid formula then the original one is also valid. That soundness property relies upon two things: first the modeling of B operators as presented in Sect. 3.1 must be faithful to the B-Book, second the translation mechanism given in Sect. 3.2 must be sound. Both of these ingredients are small and natural, so we are confident on their soundness. The modeling contains 3 type declarations, 35 function symbols, 5 predicate symbols, 25 axioms, and 21 lemmas[2]. The bpo2why translator is made of 2,057 lines of OCaml: 701 lines for parsing, 957 lines for type inference, and 399 lines for the translation.

However, in such a process it is easy to make a mistake when writing down axioms, which could result in an inconsistent theory in which we could prove anything. To prevent from such an inconsistency, we designed Coq *realizations* of the Why3 theories in use. Realizing theories in Coq is a feature provided by Why3. It automatically translates a given Why3 theory into a Coq module, where each abstract definition or axiom is respectively written as a concrete definition or a lemma. The latter must then be filled in by the user.

The first step is to provide a Coq definition of the type of polymorphic sets. We use the higher-order features of Coq, and define set α as a function $\alpha \rightarrow$ bool, that is a set S of elements of type α is identified with its characteristic function. The membership function is thus defined trivially as (mem x s) := (s x). From such a definition, it is straightforward to define the basic set operators empty set, union, etc. and prove that the axioms we pose are valid. However, realizing our set equality and our extensionality axiom is not an easy task. It is indeed not provable in Coq that s1==s2 implies s1=s2: pointwise equality of functions

[2] We modeled only the B constructs needed for our case studies.

does not imply equality of these functions, it is the so-called extensionality of function equality.

Thus, we pose functional extensionality as an axiom in Coq. Actually functional extensionality is not the only axiom we need. We also admit the excluded middle, because we need to have decidability of membership in a set, and finally we admit the axiom of choice to be able to realize the apply operator, which allows to construct a function from a relation. It is commonly admitted that adding these general-purpose axioms in the Coq calculus of inductive constructions is consistent, indeed by interpretation into a standard set-theoretic boolean model [3].

4 Experiments

We applied our technique on a proprietary use case called RCS3. This is a B project modeling the software controlling a railway level crossing system. This project has been entirely proved inside Atelier B, so all proof obligations are valid. While being a small project (about 3,000 lines of generated C code), it is representative of a B development with sets, sets of sets, relations, sequences, and linear integer arithmetic. The project is made of 31 machines (specification, refinement, or implementation), generating 2,247 proof obligations. Atelier B 4.0.2 automatic prover in F1 force proves 94% of them using a 10 seconds time limit, leaving 129 unproved proof obligations.

Our bpo2why translator can be applied on all generated PO files. We can then launch the Why3 tool chain on them using Alt-Ergo, CVC3, and Z3 provers. We use the following strategy to run the provers: the three provers are launched in parallel on all proof obligations, four at a time, with a 2 seconds time limit. For remaining unproved goals, we run once again the three provers with a 60 seconds time limit.

The comparison of the two proof chains is given in Fig. 10 (only machines generating proof obligations are shown). Overall, the Why3 proof tool chain proves more proof obligations than Atelier B's automatic prover (including the Timer machine previously presented). Only 19 proof obligations are not proved, corresponding to a 85% improvement. In only one machine, Automaton_context_i, the Why3 tool chain proves less proof obligations than Atelier B. This machine contains set expressions between enumerated sets. We do not know yet why such expressions are difficult for our tool chain. The 10 proof obligations in Warning_section_i machine are considered "difficult" ones. They need an elaborated mathematical proof with exhibition of witnesses (for existential quantifiers) based on properties of a bijection.

An interesting by-product of this experiment is that none of Alt-Ergo, CVC3, and Z3 automatic provers proves all proof obligations, even with a 60s time limit. For the three provers, there is at least one proof obligation which is proved by this prover and by none of the others. This result confirms the usefulness of the Why3 tool chain that targets several provers and thus allows to use them in a complementary way.

Machine	# of PO	Unproved by Atelier B	Unproved by Why3
Automaton	4	0	0
Automaton_context_i	10	**8**	9
Automaton_i	229	71	**0**
Automaton_transitions	189	7	**0**
Automaton_transitions_i	1678	25	**0**
Boom_detectors_i	16	0	0
Configuration_i	7	4	**0**
Indicators_i	12	0	0
Lamps_bells_i	4	0	0
Timer	3	1	**0**
Timer_i	10	0	0
Track_circuit	2	0	0
Track_circuit_i	1	0	0
Train_detector_i	4	0	0
Warning_section	2	0	0
Warning_section_i	59	11	**10**
Warning_section_r	17	2	**0**
Total	2247	129	**19**

Fig. 10. Comparison of Why3 tool chain with Atelier B on RCS3 use case (smaller is better)

Regarding proving time, the Why3 tool chain takes 35 min 34 s to prove all goals with the three provers using 4 cores, roughly 12 min per prover. Using F1 proving force, automatic prover of Atelier B 4.0 on one core[3] proves its proof obligations in 1 min 2 s. Using F3 force, we do not get any answer from Atelier B in 30 minutes. Discarding machine Automaton_context_i, it completes in 7 min 5 s. There is net gain of 2 proof obligations in machine Warning_section_r. Overall, Atelier B is much faster to prove the proof obligations, but Why3 produces a better result in an acceptable time. As the time needed by the user to look at unproved proof obligation is very costly, we think that any gain in automatic proofs is an effective development time gain.

Digital Watch Example. We have also applied our tool on a second example, the model of a digital watch. This model is less complex. It generates 777 proof obligations, of which 11 are not proved by Atelier B in F1 force. Using our translator and then the Why3 tool chain with Alt-Ergo, CVC3 and Z3, we can automatically prove all but one proof obligation, the remaining one being not provable (a bug in the original model). This result confirms that the Why3 tool chain improves the efficiency of proofs by exploiting the capabilities of modern SMT provers (this model contains a lot of integer arithmetic expressions).

[3] Latest Atelier B 4.0.2 is able to use all cores of a multi-core machine but we could not use this version for our tests.

5 Comparison with Related Work

Bodeveix, Filali, and Muñoz [8] formalized the semantics of B in both Coq and PVS. They define a (mostly) shallow embedding of the B notions of generalized substitutions and machines. B's set theory is not formalized at all; the native logics of Coq and PVS are used instead.

The BRILLANT [11] toolset made by Colin et al. generates B's proof obligation that can be incorporated inside the Coq proof assistant thanks to the BiCoax [10] libraries. (The BiCoax work is itself an extension of the B/PhoX [17] work based on PhoX proof assistant.) The proof obligations can then be proved manually or by Coq automatic tactics. Jacquel et al. [15] propose another deep embedding of B's set theory in Coq, whose purpose is to check using Coq that the rewrite rules used in the B prover are valid. Our Coq realization is similar to both Coq formalizations above. However our Coq model is only built for the purpose of showing the consistency, not for the purpose of performing proofs interactively with Coq.

Déharbe made a work [14] very similar to ours. Namely, Déharbe interfaces SMT solvers having an SMT-LIB interface with the Rodin development tool for Event-B. The proof obligations generated by Rodin are transformed into Boolean formulas, sets being transformed into their characteristic predicate. Déharbe's approach is limited to basic sets (*i.e.* no set of sets) while ours is able to transform all set-related expressions of the B Method. Moreover, Why3 is able to interface itself to more automatic provers, not limited by the SMT-LIB interface. For his tests, Déharbe used only one SMT solver, veriT. But even using one solver, he obtained a significant improvement in proofs, as we did.

6 Conclusion and Perspectives

In this paper, we have presented an approach and a tool to transform Atelier B's proof obligations into the Why3 proof tool chain in order to prove them using several automated provers. While being a shallow embedding of B logic into Why3 logic, we have arguments to believe that this translation is sound: mainly the translation is short and we can check axioms' correctness through Coq realization. We have applied this approach on a small but reasonably complex use case and we found a significant improvement in the number of proof obligations that are automatically proved.

This work could be improved in several ways. First of all, we could support more B operators in order to handle more complex and industrial models. The current subset of operators is the one needed to handle our use cases. Adding one B operator amounts to incrementally complete the Why3 theories, complete its Coq realization, and add a translation rule in the translator. Secondly we could try to increase the number of automatically proved proof obligations by analyzing in detail why some of them are not proved. This may amount to provide more lemmas as hints, or annotate them with *triggers*. Thirdly, we could increase our confidence level in the embedding of B into Why3 by proving B-Book's lemmas into our Why3 framework. Fourthly we could better integrate our tool chain into Atelier B tool, for example by applying it after Atelier B automatic prover and then merging our results into Atelier B GUI. Last but not least, we could try to improve the automated provers themselves in order to better handle proof obligations generated by the B Method. E.g. an interesting theoretical question is whether the rewriting techniques used by the B prover could be combined with the satisfiability modulo theory approach.

References

1. Abrial, J.-R.: The B-Book: Assigning programs to meanings. Cambridge University Press (1996)
2. Badeau, F., Amelot, A.: Using B as a High Level Programming Language in an Industrial Project: Roissy VAL. In: Treharne, H., King, S., C. Henson, M., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 334–354. Springer, Heidelberg (2005)
3. Barras, B.: Sets in Coq, Coq in sets. Journal of Formalized Reasoning (2010)
4. Barrett, C.W., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
5. Behm, P., Benoit, P., Faivre, A., Meynadier, J.-M.: Météor: A Successful Application of B in a Large Project. In: Wing, J.M., Woodcock, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 369–387. Springer, Heidelberg (1999)
6. Bobot, F., Filliâtre, J.-C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Workshop on Intermediate Verification Languages (2011)
7. Bobot, F., Paskevich, A.: Expressing Polymorphic Types in a Many-Sorted Language. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCos 2011. LNCS, vol. 6989, pp. 87–102. Springer, Heidelberg (2011)
8. Bodeveix, J.-P., Filali, M., Muñoz, C.: A formalization of the B-method in Coq and PVS. In: B-User Group Meeting, Formal Methods, pp. 33–49 (1999)
9. ClearSy. Language Keywords and Operators, 1.8.5 edn., `http://www.tools.clearsy.com/images/3/33/Symboles_en.pdf`
10. Colin, S., Mariano, G.: Coq, l'alpha et l'omega de la preuve pour B ? (February 2009), `http://hal.archives-ouvertes.fr/hal-00361302/PDF/bicoax.pdf`
11. Colin, S., Petit, D., Mariano, G., Poirriez, V.: BRILLANT: an open source platform for B. In: Workshop on Tool Building in Formal Methods (February 2010)
12. Conchon, S., Contejean, E., Kanig, J., Lescuyer, S.: CC(X): Semantic combination of congruence closure with solvable theories. ENTCS 198(2), 51–69 (2008)
13. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
14. Déharbe, D.: Integration of SMT-solvers in B and Event-B development environments. Science of Computer Programming (2011)
15. Jacquel, M., Berkani, K., Delahaye, D., Dubois, C.: Verifying B Proof Rules Using Deep Embedding and Automated Theorem Proving. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 253–268. Springer, Heidelberg (2011)
16. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences 17 (1978)
17. Rocheteau, J., Colin, S., Mariano, G., Poirriez, V.: Évaluation de l'extensibilité de PhoX: B/PhoX un assistant de preuves pour B. In: JFLA, pp. 139–153 (2004)