BARReL: A modern backend for Atelier B in Lean

Ghilain Bergeron
Université de Lorraine, CNRS, Inria, LORIA

Vincent Trélat
Université de Lorraine, CNRS, INRIA, Nancy, France

Abstract

BARReL is a Lean 4 library bridging Atelier B, an industrial tool for the B method, and the Lean proof assistant by enabling users to conduct their formal B developments—up to machine refinement and implementation—interactively inside Lean, while retaining standard B syntax. B partial operators are carefully encoded by generating explicit well-definedness conditions, leveraging Lean's dependent types to enforce a well-definedness discipline by construction. That is, proof obligations and proof steps cannot silently rely on ill-typed or ill-defined instantiations. BARReL also features basic automation to try to discharge such well-definedness conditions automatically. The implementation is written entirely using Lean meta-programming and is designed to be modular: extending the supported B fragment typically requires only adding new syntax and encoding clauses. We illustrate the approach on a small but representative case study, and argue that BARReL can act as a stepping stone towards a strongly reliable Atelier B toolchain grounded in the Lean proof assistant.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Parsing; Theory of computation → Assertions; Theory of computation → Invariants

Keywords and phrases Lean 4, Atelier B, Meta-programming, Interactive Theorem Proving

Digital Object Identifier 10.4230/LIPIcs...

Funding *Vincent Trélat*: ANR project BLaSST (ANR-21-CE25-0010).

1 Introduction

The B method [1] and its associated toolsets, notably Atelier B [8], have been used for decades in industrial developments where strong assurance arguments are required. The B method is a formal development methodology based on stepwise refinement of abstract high-level specifications into executable code, supported by a rich mathematical language rooted in set theory and first-order logic. B specifications are written as *abstract machines* with sets, constants, variables, properties, invariants and operations; Atelier B generates *proof obligations* (POs) that guarantee, for example, invariant preservation and refinement correctness, and ships with automated and interactive provers tailored to this logic. In many certified developments, these tools are trusted as part of the verification chain.

In parallel, interactive theorem provers have seen rapid adoption, driven by their expressive type theories, powerful automation, and growing libraries of reusable mathematics and verification components. Lean 4 [11] is based on the calculus of inductive constructions with quotients, and features a powerful dependently typed system. In particular, it offers a small trusted kernel, a rich standard library, and a meta-programming framework that makes it attractive both as a programming language and a proof assistant. However, despite B's long-standing industrial role, support for using modern interactive theorem provers as backends for Atelier B remains very limited: some previous work targets Isabelle [12] but is either no longer maintained [13] or provides specific support [5] for Event-B [2] in Rodin [3]. There were also developments in Rocq [14] targeting PVS [7], focusing primarily on formalizing the semantics of B rather than providing an integrated backend.

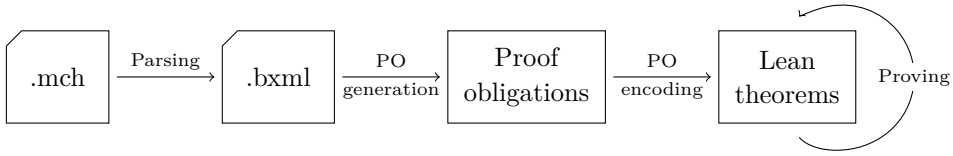

Figure 1 High-level picture of BARReL.

This paper presents *BARReL* (B Automated tRanslation for Reasoning in Lean), a Lean 4 library that bridges Atelier B and Lean by turning B proof obligations into Lean theorem statements that must be later proved by the user. Given a B machine (or an existing POG file), BARReL invokes Atelier B to generate POs when needed and translates them into Lean theorems expressed over Mathlib [4]'s set-theoretic primitives, preserving the original structures and notations as closely as possible. BARReL also features its own generation of *well-definedness* (WD) proof obligations, which capture the definedness side-conditions of B partial operators (e.g. minimum, function application, cardinality). These are treated as standard Lean statements, so that definedness constraints can be proved once and then reused in subsequent interactive proofs. The library provides commands allowing a user to turn B developments into a sequence of Lean theorems corresponding to the POs of the developments; users then discharge these goals interactively with standard tactic scripts, and the resulting theorems are added to the environment under names derived from the original POG tags. The translation pipeline is organized into clearly separated components: a lightweight embedding of B syntax, a POG file reader, an encoding layer mapping B constructs to Lean expressions, and a discharger that generates theorem declarations and connects them to user proofs. This high-level architecture is illustrated in Figure 1.

Our contributions are therefore threefold:

- BARReL, an open-source lightweight and modular backend for B developments written in Lean 4;
- a translation pipeline preserving B notations and proof obligations, properly handling partial operators via explicit well-definedness conditions;
- an evaluation on a small refinement chain demonstrating an Atelier B-style refinement workflow inside Lean, with most well-definedness side-conditions discharged automatically.

Finally, BARReL opens the door to using Lean as an independent backend for the B Method and suggests interesting extensions, such as an integrated proof obligation generator—which could even be verified—and even an embedded DSL for writing B developments directly in Lean.

The rest of this paper is organised as follows. We first recall the necessary background on the B method, proof obligations, and Lean. We then present the design and implementation of BARReL, detailing the encoding of B syntax, the interface with Atelier B's proof obligation generator, and our treatment of partial operators and their well-definedness obligations. We also discuss basic automation features to reduce user effort in discharging common proof obligations. This is followed by a case study illustrating the end-to-end workflow of using BARReL on a representative B development. We then compare BARReL's workflow against Atelier B's built-in interactive prover, highlighting notable differences in proof styles. We conclude with a discussion of future extensions, and position our work with respect to existing approaches to connecting B with modern interactive theorem provers.

2 Related Work

Several approaches have explored using interactive theorem provers to increase confidence in developments written in the B family of methods by moving proof obligations into a prover with a small, auditable kernel. An early line of work translates (fragments of) B into PVS, leveraging type synthesis to recover the typing information needed by the target logic and then relying on PVS for interactive discharge of the resulting obligations [7]. In the Isabelle/HOL ecosystem, mechanizations have focused primarily on Event-B. The Isabelle plugin for Rodin [13] provides a formalization of the Event-B logic, including treatment of partial functions via definitional extensions and supporting automation. A more user-facing shallow embedding was proposed recently, viewing Event-B as a DSL hosted in Isabelle/HOL [5]. Beyond these, the Rocq proof assistant has also been used in complementary work [7] that emphasizes mechanized semantics and meta-theory, rather than providing a drop-in backend integrated with an industrial proof-obligation generator.

BARReL is complementary in scope and emphasis. Rather than formalizing the logic, it targets the existing Atelier B workflow. By consuming POG files or invoking the generator, BARReL turns each obligation into a Lean theorem over Mathlib's set-theoretic primitives [4], while preserving B notations and focusing on usability for B developers. BARReL also aims at minimizing the learning effort required to get started and thus provides basic automation to reduce user effort, emphasizing on explicit handling of well-definedness conditions.

3 Overview of BARReL

This section presents BARReL and its workflow as depicted by Figure 1. We first recall the B method, then explain how BARReL handles well-definedness conditions and finally detail its inner workings and automation features.

3.1 Background

The B method and proof obligations

The B method is a state-based formal development approach centered on *abstract machines* and stepwise *refinement*. A machine declares abstract sets and constants, a state described by variables constrained by an inductive invariant, an initialization, and operations. A development proceeds by refining an abstract machine into more concrete refinements down to an implementation, introducing representation data together with a *gluing invariant* relating concrete and abstract states. From each machine and refinement step, Atelier B generates *proof obligations* (POs): logical sequents whose discharge guarantees that the model is consistent and that refinement is correct. Concretely, POs cover invariant preservation by the initialization and each operation under its precondition, and simulation-style obligations ensuring that concrete operations preserve the abstract behavior through the gluing invariant. In addition, the PO generator emits *well-definedness* (WD) obligations for the many partial constructs of the language.

POs can be discharged automatically using Atelier B's internal provers and by delegating to external backends. In particular, several approaches translate B obligations to SMT solvers such as Z3 [9] or cvc5 [6] via dedicated encodings, including the *ppTransSMT* [10] and the more recent *BEer* [15] encodings. Obligations that remain—often those requiring non-trivial quantifier instantiation, refinement reasoning, or tedious WD bookkeeping—are typically

discharged interactively using Atelier B's GUI-oriented prover, where the user guides rule applications and instantiations.

The B language in a nutshell

B's mathematical language is based on classical set theory and first-order logic with equality. Specifications are written using the usual set-theoretic constructors such as comprehension, powerset, cartesian product, and an algebra of relations. Ordered pairs are first-class values (written as *maplets* $x \mapsto y$), and relations are simply sets of such pairs, i.e. $R \subseteq A \times B$. Functions are not primitive: a partial function f from A to B (written $f \in A \nrightarrow B$) is a relation $f \subseteq A \times B$ whose domain is contained in A and which is *functional*: for all $x \in A$ and $y, z \in B$, if $(x \mapsto y) \in f$ and $(x \mapsto z) \in f$ then $y = z$. Total ($A \longrightarrow B$), injective (partial $A \rightarrowtail B$ and total $A \rightarrowtail B$), and surjective (partial $A \twoheadrightarrow B$ and total $A \twoheadrightarrow B$) variants are defined similarly as usual.

Although the underlying logic is set-theoretic and thus untyped, Atelier B enforces a static well-formedness discipline often described as "typing": every identifier and expression must be assigned a "parent set" that must be deducible for every constant, variable, and expression in a specification. This guarantees in particular that constructed sets are homogeneous and rules out ill-formed terms. These deduced parent sets provide exactly the information needed to interpret B expressions in Lean's typed logic in our translation.

▶ **Example 3.1.** The expression $1 + \top$ is well-formed in ZFC set theory—and definitionally equal to the set $\mathbb{B}$ of Booleans, or the natural number 2—but it is not a valid expression in B. Indeed, 1 has parent set $\mathbb{Z}$ and $\top$ has parent set $\mathbb{B}$, and $+$ expects both its arguments to have parent set $\mathbb{Z}$.

Foundations of B in Lean

We leverage these typing annotations and ground our encoding in typed higher order logic (rather than ZFC set theory) by using Mathlib's `Set` and `SetRel` types[1] to represent sets and relations: each inferred B carrier is mapped to a corresponding Lean type—`INTEGER` to `Int`, cartesian products to Lean product types, and powersets to `Set` over the element type. This is a convenient representation since Mathlib comes with many definitions and theorems related to `Set` for us to use when proving the translated obligations in Lean. This representation also allows us to express every kind of set used in B specifications since all sets in B are required to be *homogeneous*, i.e. restricted to a uniform shape of elements. For instance, the set $\{0, "hi"\}$ must be considered ill-formed by implementations of the B method.

We start by defining standard B operators directly in Lean—reusing their notations from B, in order for the POs to be easily understandable by B users—and typing them appropriately in Lean to keep their use as generic as possible. Some examples (the set of relations, the set of partial functions, the set of total injective functions and domain subtraction) are given in Listing 1. We then provide proofs, using Mathlib's infrastructure and theorems, of various laws and theorems of B: for any total function $f \in A \longrightarrow B$ its domain $\mathrm{dom}(f)$ equals A; assuming that $a \leq b$, the minimum of $a..b$ is a; etc. Both the definitions and theorems are readily available to end users when importing BARReL in a

[1] Which are defined respectively as `Set` $\alpha \triangleq \alpha \to$ `Prop` and `SetRel` $\alpha\ \beta \triangleq$ `Set`$(\alpha \times \beta)$.

Listing 1 Encoding of some B operators in Lean.

```
abbrev rels {α β} (A : Set α) (B : Set β) : Set (SetRel α β) :=
  𝒫 (A ×ˢ B)
scoped infixl:125 " ⟷ " => rels

abbrev pfun {α β} (A : Set α) (B : Set β) : Set (SetRel α β) :=
  { f ∈ A ⟷ B | ∀ {⦃x y z⦄}, (x, y) ∈ f → (x, z) ∈ f → y = z }
scoped infixl:125 " ⇸ " => pfun

abbrev injTFun {α β} (A : Set α) (B : Set β) : Set (SetRel α β) :=
  A ⤖ B ∩ A ⟶ B
scoped infixl:125 " ↣ " => injTFun

abbrev domSubtr {α β} (E : Set α) (R : SetRel α β) : SetRel α β :=
  { z ∈ R | z.1 ∉ E }
scoped infixl:160 " ⩤ " => domSubtr
```

Lean project, and serve as basic building blocks for proofs of B obligations translated by our tool.

3.2 Encoding partial B operators

A central challenge in this setting is the use of *partial* B operators, meaning that they are only defined when some *well-definedness* (WD) conditions are satisfied. A typical example of such an operator is the minimum operator $\min(S)$, which when given a set S (of integers or reals) returns its smallest element (w.r.t. the ordering $\leq$ on integers or reals), if it exists, and is undefined otherwise. These partial operators are typically encoded using Hilbert's bounded choice operator $\epsilon\ x \in S.\ P$, which is left unspecified if no element of S satisfies P. The *WD condition* of the corresponding operator is precisely that P is satisfied by (at least) an element of S.

Atelier B handles this partiality by generating the WD conditions of each partial operator used in a PO as side goals and assumes in the main goal that the partial operators are well-defined. However, one must trust that these WD conditions actually correspond to the WD conditions that are expected from the context of those partial operators, and there is no clear dependency—even after inspecting the generated POG file—between the main goal and the WD side goals. Furthermore, these WD conditions are only generated for the goal itself but not during proof steps. This can lead to inconsistencies if the user is not careful enough for instance when instantiating quantified variables in the interactive prover[2].

Our approach in BARReL differs from Atelier B in that the dependency between the WD goals and the main goals is clear and explicit. First, we do not rely on the WD conditions that are generated by Atelier B and instead generate our own to match the exact context in which they are inserted. This is not necessarily true of the WD conditions generated by Atelier B since they undergo some simplification steps or do not match exactly our encoding of some predicates and WD conditions.

▶ Remark 3.2. Atelier B's WD conditions for the min and max operators are slightly different from ours. For instance, it treats integers and reals differently: a set S of integers

[2] This is concretely illustrated below in Section 5.

Listing 2 Encoding of the $\min(S)$ and $F(x)$ partial operators.

```
structure min.WD {α} [LinearOrder α] (S : Set α) : Prop where
  isBoundedBelow : ∃ x ∈ S, ∀ y ∈ S, x ≤ y

noncomputable
abbrev min {α} [LinearOrder α] (S : Set α) (wd : min.WD S) : α :=
  Classical.choose wd.isBoundedBelow

structure app.WD {α β} (f : SetRel α β) (x : α) : Prop where
  isPartialFunction : f ∈ dom f ⟶ ran f
  isInDomain : x ∈ dom f -- ≡ ∃ y, (x, y) ∈ f

noncomputable
abbrev app {α β} (f : SetRel α β) (x : α) (wd : app.WD f x) : β :=
  Classical.choose wd.isInDomain
```

admits a minimum if it is non-empty when intersected with $\mathbb{N}$, and a maximum if it is non-empty when intersected with $-\mathbb{N}$; whereas for reals, S must be non-empty and bounded below (resp. above) to admit a minimum (resp. maximum). BARReL makes use of the same WD conditions for both integers, reals, and actually any type implementing a *partial order*, requiring non-emptiness and boundedness below (resp. above) for minimum (resp. maximum)—which is logically equivalent in the case of integers, yet more generic:

```
theorem min.WD_iff_AtelierB_WD {S : Set ℤ} :
  min.WD S ↔ S ≠ ∅ ∧ S ∩ (INTEGER \ NATURAL) ∈ FIN INTEGER
theorem max.WD_iff_AtelierB_WD {S : Set ℤ} :
  max.WD S ↔ S ≠ ∅ ∧ S ∩ NATURAL ∈ FIN INTEGER
```

In practice, our tool generates WD side goals that are equivalent to the WD side goals generated by Atelier B. The only difference is that Atelier B may remove unnecessary hypotheses and perform basic simplification on the WD goals at generation time, which may alter the PO, whereas our WD conditions are generated from the exact Lean context. Then, we take advantage of the dependent types of Lean to encode partial operators using `Classical.choose` (Hilbert's ϵ operator, in Lean, whose type is $\{\alpha : \texttt{Sort}\ u\} \to \{p : \alpha \to \texttt{Prop}\} \to (\exists x,\ p\ x) \to \alpha$) which explicitly depends on a proof that the predicate is satisfied by at least one element.[3] This means that one cannot construct a term depending on a partial operator without providing a proof that it is well-defined. The statement of this proof, which is the WD condition of the operator, is then generated as a side theorem (or a subgoal) which must be discharged for the main theorem to be fully proven, i.e. not depend on the `sorryAx` axiom—the axiom that is introduced by admitted theorems. To illustrate this use of `Classical.choose`, the encoding (and WD conditions) of $\min(S)$ and function application $F(x)$ are given in Listing 2.

[3] Lean has no built-in notion of partiality and must therefore exhibit WD conditions as explicit proof parameters of the relevant total functions.

3.3 Inner Workings of BARReL and Automation

User-level workflow

BARReL is organized into four distinct passes. First, the machine/system/refinement/implementation is parsed into the BXML format, and proof obligations are generated from this format using Atelier B's internal tools `bxml` and `pog`.

Although Atelier B's `pog` can generate WD conditions for the given machine, we choose to rely on our own generation instead, since the WD conditions that are generated by `pog` may not exactly match the context in which they will appear in the Lean terms. Then, the resulting POs (in XML format) are parsed by BARReL into an Abstract Syntax Tree (AST) and normalized. Normalization ensures that B predicates like $P \wedge Q \Rightarrow R$ are treated the same as the logically equivalent predicate $P \Rightarrow Q \Rightarrow R$. Although it is unnecessary, we perform normalization of B terms solely for the purpose of generating Lean goals that overall require fewer conjunction destructions. The normalized B formulas are then transcribed into Lean terms using Lean's facilities for meta-programming and elaboration. However, a lot of bookkeeping has to happen in order to encode partial B operators, since they rely on WD conditions that must be inserted on the fly. Instead of handling the bookkeeping ourselves, we let Lean handle it internally by generating meta-variables (containing their local contexts) in place of the WD conditions. We later assign the generated meta-variables with fresh theorem names whose propositions are extracted from the context of the meta-variable.

▶ **Example 3.3.** Let us illustrate how WD conditions are handled by BARReL on the following (already normalized) B predicate, stating that every non-empty finite set of integers—i.e. belonging to $\mathrm{FIN}_1(\mathbb{Z})$—whose elements are non-positive has a minimal element that is non-positive:

$$\forall S \cdot (S \in \mathrm{FIN}_1(\mathbb{Z}) \Rightarrow (\forall x \cdot x \in S \Rightarrow x \leq 0) \Rightarrow \min(S) \leq 0)$$

The first step of the pipeline is generating a Lean term while inserting meta-variables in place of WD conditions (where h_1, h_2 and h_3 are fresh identifiers):

$$P := \forall\,(S : \texttt{Set}\ \mathbb{Z})\,(h_1 : S \in \texttt{B.Builtins.FIN}_1\ \mathbb{Z})(h_2 : \forall\,(x : \mathbb{Z})\,(h_3 : x \in S),\ x \leq 0),$$
$$\texttt{B.Builtins.min}\ S\ (?m_1\ S\ h_1\ h_2) \leq 0$$

The new meta-variable $?m_1$ is generated internally by BARReL, and contains $S : \texttt{Set}\ \mathbb{Z}$, $h_1 : S \in \texttt{B.Builtins.FIN}_1\ \mathbb{Z}$ and $h_2 : \forall\,(x : \mathbb{Z})\ (h_3 : x \in S),\ x \leq 0$ in its local context. Although its type is $\texttt{B.Builtins.min.WD}\ S$, its local context is implicitly universally quantified, hence the full application $?m_1\ S\ h_1\ h_2$ in the goal. A fresh theorem statement min_wd_1 is then generated—and remains to be proved—from the type of $?m_1$ universally quantified by its local context:

$$\texttt{theorem}\ min\_wd_1 :$$
$$\forall\,(S : \texttt{Set}\ \mathbb{Z})\,(h_1 : S \in \texttt{B.Builtins.FIN}_1\ \mathbb{Z})(h_2 : \forall\,(x : \mathbb{Z})\,(h_3 : x \in S),\ x \leq 0),$$
$$\texttt{B.Builtins.min.WD}\ S$$

$?m_1$ is finally assigned min_wd_1, and substituted in P to obtain the final goal to be proved by the user:

$$\vdash \forall\,(S : \texttt{Set}\ \mathbb{Z})\,(h_1 : S \in \texttt{B.Builtins.FIN}_1\ \mathbb{Z})(h_2 : \forall\,(x : \mathbb{Z})\,(h_3 : x \in S),\ x \leq 0),$$
$$\texttt{B.Builtins.min}\ S\ (min\_wd_1\ S\ h_1\ h_2) \leq 0$$

Listing 3 Some WD lemmas about extrema in our B library.

```
@[wd_min]
theorem min.WD_singleton {α : Type _} [PartialOrder α] {a : α} :
  min.WD {a} := ⋯
@[wd_min]
theorem min.WD_of_finite {α : Type _} [LinearOrder α] {S A : Set α}
  (h : S ∈ FIN₁ A) : min.WD S := ⋯
@[wd_min]
theorem min.WD_interval {lo hi : ℤ} (h : lo ≤ hi) :
  min.WD (lo..hi) := ⋯
```

All these previous steps are performed by the `import <type> <name> from <folder>` command of our tool. `<type>` may be one of `machine`, `system`, `refinement`, `implementation` or `pog` depending on whether the file to be imported is a machine, system, refinement, implementation or straight up a POG file, and `<name>` is the file name without the extension. In the case of POG files, the first step converting into the BXML format is skipped. Finally, the command `prove_obligations_of <name>` gives back all the generated WD conditions and propositions to the user for them to prove. Each WD condition and proposition is then inserted into the global context as a theorem, which the user can re-use instead of duplicating proofs or check for no use of the `sorryAx` axiom—so that every obligation is fully proven.

Proof automation

Before presenting goals to the user, BARReL runs a lightweight, predictable automation pass whose primary purpose is to discharge routine side conditions induced by our encoding of B partial operators. Concretely, the tactic layer repeatedly applies potentially relevant lemmas—backtracking as needed—that relate WD subgoals and available hypotheses. These helper lemmas provide definitional equations and rewriting/simplification facts for the B constructs supported by BARReL about sets, relations, functions, arithmetic and finiteness idioms, so that many generated WD goals reduce to elementary facts about membership, domain/range inclusion, finiteness, and order bounds and can be closed automatically. We leverage Mathlib's features to tag helper lemmas with relevant categories—`wd_min`, `wd_max`, `wd_app`, `wd_card`—and tap into its rich library of set-theoretic results to keep the tactic small and focused. To those tags also correspond the names of the tactics that are run automatically by BARReL's automation layer, so that users may also invoke them manually when needed or extend them with their own lemmas.

Beyond the core encoding of B primitives, BARReL comes with a curated—still small, but already well-furnished—library of theorems about built-in operators. Its purpose is twofold:

1. provide reusable WD facts that discharge the side goals generated for partial operators;
2. offer canonical rewrite lemmas that turn B-shaped expressions into "Mathlib-friendly" goals.

For instance, in the arithmetic fragment used throughout our running examples, the library includes dedicated WD lemmas for the minimum operation, as shown in Listing 3, each of them being tagged with the relevant theorem category.

The library also provides basic membership results about finite sets and intervals; and in the same spirit, "computational" equalities are exposed as rewrite rules, again all tagged

Listing 4 Some membership lemmas about finite sets and intervals in our B library.

```
theorem interval.FIN₁_mem {lo hi : ℤ} (h : lo ≤ hi) :
  lo .. hi ∈ FIN₁ INTEGER := ···
@[simp]
theorem min.of_singleton {α : Type _} [PartialOrder α] {a : α} :
  min {a} (min.WD_singleton) = a := ···
@[simp]
theorem interval.min_eq {lo hi : Int} (h : lo ≤ hi) :
  min (lo .. hi) (min.WD_interval h) = lo := ···
```

appropriately to be picked up by relevant automation, as shown in Listing 4—with proofs replaced by ··· for brevity.

The remaining obligations—typically invariant preservation and refinement simulation goals—are left as interactive proof tasks, keeping the automation transparent and predictable while still removing most of the WD bookkeeping arising from B developments.

Trust boundary and guarantees

BARReL should be understood as a *proof-producing backend* for the obligations it emits: once a proof script succeeds, the corresponding theorem is checked by Lean's kernel. Concretely, the remaining trusted assumptions are:

- Lean's kernel, and its standard meta-theory,
- the correctness of the external PO generation step performed by Atelier B (`bxml/pog`) at BARReL's import phase,
- the faithfulness of BARReL's set-theoretic embedding of B constructs and proof obligations to the intended B semantics.

What BARReL *does* guarantee by construction is that partial B operators cannot be used in generated Lean terms without supplying explicit WD evidence, and that any successfully discharged goal yields a kernel-checked Lean theorem—i.e. a Lean (proof-)term having the right type—under that embedding. A fully independent chain would additionally require a verified PO generator, which we leave as future work.

4 Case study: a refinement chain for minimum search

This section evaluates BARReL on a small example with a specification and two levels of refinement, fully carried out with BARReL in Lean and compared to Atelier B's own interactive prover. The development computes the minimum of a non-empty finite set of integers and is progressively refined into an index-based traversal of a table, therefore demonstrating BARReL's support for Atelier B's refinement pipeline[4] while correctly generating and exposing WD conditions needed for partial B operators at each level.

The development consists of: a specification `MinSearch.mch`, a first refinement pass `MinSearch_r1.ref` and a second refinement pass `MinSearch_r2.ref`, together with a Lean script `MinSearch.lean` containing the proofs of all generated POs. This chain exercises both data refinement and partial operators (`min`, `card`, and application), making it a compact stress-test for BARReL's WD generation and automation.

[4] Down to implementation, but this is not shown in this development.

Listing 5 Specification `MinSearch.mch`.

```
MACHINE MinSearch
CONSTANTS
  S
PROPERTIES
  S ∈ FIN1(INTEGER)
VARIABLES done, m
INVARIANT
  done ∈ FIN1(S) ∧
  m ∈ S          ∧
  m = min(done)
INITIALISATION
  ANY x WHERE x ∈ S
  THEN
    done := {x} || m := x
  END
OPERATIONS
  mi ⟵ search_min =
    PRE done = S THEN
      mi := m
    END;
  step =
    IF done ≠ S THEN
      ANY add WHERE
        add ∈ FIN1(S - done)
      THEN
        done := done ∪ add ||
        m := min(done ∪ add)
      END
    END
END
```

Listing 6 Lean goal corresponding to the first conjunct of the preservation of the invariant preservation for operation `step` in `MinSearch.mch`.

```
S done add : Set ℤ
m x : ℤ
S_fin      : S ∈ FIN₁ INTEGER
done_fin   : done ∈ FIN₁ S
m_mem_S    : m ∈ S
m_is_min   : m = B.Builtins.min done ⋯
done_ne_S  : done ≠ S
add_fin    : add ∈ FIN₁ (S \ done)
⊢ done ∪ add ∈ FIN₁ S
```

4.1 Abstract specification

A B *MACHINE* declares *CONSTANTS* (fixed parameters) constrained by a *PROPERTIES* clause, state *VARIABLES* constrained by an *INVARIANT*, and an *INITIALISATION* that must establish the invariant from an initial state. The initial machine `MinSearch` shown in Listing 5 fixes a non-empty constant subset `S ∈ FIN1(INTEGER)`, where `FIN1(INTEGER)` denotes the set of *non-empty finite* subsets of integers. It maintains a non-empty set `done` of explored elements of `S` together with a candidate `m` satisfying `m = min(done)`. The operation `step` non-deterministically adds a non-empty finite subset of unexplored elements and recomputes the minimum; `search_min` returns the candidate once all elements have been processed.

Operations may carry a precondition (*PRE*): the construct *ANY* `x` *WHERE* `P` *THEN* `S` *END* non-deterministically chooses an arbitrary value satisfying `P`. The notation `mi ⟵ search_min` indicates that `mi` is an output parameter, and `||` denotes parallel assignment.

This level already illustrates BARReL's treatment of partial operators: since `min(done)` and `min(done ∪ add)` are only defined when the corresponding sets have a least element, BARReL generates 4 associated WD side obligations—either as theorems or subgoals depending on the context—alongside the usual 4 proof obligations, and discharges them automatically using its built-in automation. The remaining 2 subgoals correspond to standard invariant preservation and operation correctness obligations. We illustrate this with Listing 6 showing

Listing 7 Refinement `MinSearch_r1.ref` (excerpt): refined `step`.

```
step =
  IF done ≠ S THEN
    ANY x WHERE x ∈ S - done THEN
      done := done ∪ {x} || IF x < m THEN m := x END
    END
  END
```

Listing 8 Refinement `MinSearch_r2.ref`.

```
REFINEMENT MinSearch_r2
REFINES MinSearch_r1
CONSTANTS n, tab
PROPERTIES
  S ∈ FIN1(INTEGER) ∧
  tab ∈ 1..n ⟼ S   ∧
  n = card(S)       ∧
  ran(tab) = S
VARIABLES i, mc
INVARIANT
  i ∈ 1..n          ∧
  m = mc            ∧
  done = tab[1..i]  ∧
  mc ∈ tab[1..i]    ∧
  ∀jj.(jj ∈ 1..i ⇒ mc ≤ tab(jj))
INITIALISATION
  i := 1 || mc := tab(1)
OPERATIONS
  mi <-- search_min =
    PRE i = n THEN
      mi := mc
    END;
  step =
    IF i < n THEN
      i := i + 1;
      IF tab(i) < mc THEN
        mc := tab(i)
      END
    END
END
```

the Lean goal generated by BARReL for the first conjunct of the invariant preservation for operation `step`.

4.2 First refinement: reducing non-determinism

The first refinement `MinSearch_r1` tightens `step` to non-deterministically add a *single* fresh element and update the minimum incrementally, as shown in Listing 7. Although the concrete update avoids recomputing `min`, refinement of the operation and invariant preservation still relate `m` to min(`done`), so WD facts about `min` remain part of the proof landscape. 52 WD conditions are generated at this level, which is significantly more than what Atelier B produces. Although this is entirely expected by the current design of BARReL as exposed earlier in Section 3.3, optimizations to lower this number—via subsumption or caching—are discussed in Section 6.2. All WD conditions are again automatically discharged by BARReL, while the remaining 9 POs correspond to standard refinement obligations.

4.3 Second refinement: scanning an explicit table

The second refinement `MinSearch_r2` shown in Listing 8 introduces an explicit enumeration and refines set exploration into a traversal using an index `i` and a current minimum candidate `mc`. A gluing invariant links the abstract set `done` with `tab[1..i]` (denoting the image of the interval `1..i` under `tab`) and identifies `m` with `mc`, and states that `mc` is a lower bound for all scanned entries.

This level is particularly relevant for BARReL because it combines refinement obligations

with multiple occurrences of different partial constructs like cardinality `card` and function application `tab(i)`. Correctly discharging the resulting POs requires BARReL to generate and instantiate the corresponding WD conditions for every occurrence of such operators, which makes the number of WD subgoals grow significantly: 100 WD conditions are generated—and are all automatically discharged—alongside the 21 POs corresponding to standard refinement obligations.

4.4 Discharging refinement POs inside Lean

Listing 9 Driving the refinement chain with BARReL.

```
import machine     MinSearch    from "specs/case_study"
import refinement  MinSearch_r1 from "specs/case_study"
import refinement  MinSearch_r2 from "specs/case_study"

prove_obligations_of MinSearch
next ...   -- 4 remaining goals

prove_obligations_of MinSearch_r1
next ...   -- 9 remaining goals

prove_obligations_of MinSearch_r2
next ...   -- 21 remaining goals
```

Working with BARReL involves standard Lean workflow: B files are imported and POs are generated as Lean theorems to be proved, as shown in Listing 9.

Table 1 Proof-obligation statistics for the `MinSearch` refinement chain. "#PO" excludes WD conditions, while "#WD" counts them separately. "Auto" counts obligations discharged without user interaction (Atelier B: Force 1; Lean: BARReL's default automation and `grind`). Because Atelier B and BARReL place WD obligations at different granularity (machine-level sharing vs. per-occurrence reification), "#WD" counts are not expected to match one-to-one.

	#PO (Atelier B)		#PO (BARReL)		#WD (Atelier B)		#WD (BARReL)	
File	Total	Auto	Total	Auto	Total	Auto	Total	Auto
`MinSearch`	4	2	4	2	4	4	4	4
`MinSearch_r1`	19	15	19	15	2	2	42	42
`MinSearch_r2`	21	9	21	0	9	9	100	100

In total for this case study, BARReL produces 190 goals across the three levels, 146 of which are WD conditions that are all automatically discharged using BARReL's underlying automation. The remaining 44 goals correspond to the expected proof obligations and are discharged with straightforward Lean proofs, except for 15 goals that are also discharged automatically using BARReL's automation. Table 1 summarizes the size and proof obligation statistics of each artifact. As expected, the number increases significantly with refinement, especially when moving to an explicit representation (`tab`) that introduces many occurrences of partial operators (notably application). BARReL's automation should be able to discharge most of the resulting WD conditions automatically—in this specific case, all of them.

5 Comparison with Atelier B's interactive prover

Atelier B already provides effective automation and an interactive mode for discharging proof obligations. In particular, it is tightly integrated with the PO generator and can solve many goals using its built-in provers. In practice, however, interactive proofs in Atelier B are primarily carried out through a dedicated, GUI-oriented workflow, where proof progress often depends on manual rule application and instantiation steps, especially for recurrent WD obligations. Moreover, the interactive prover documentation explicitly warns that some instantiation commands, e.g. `se` (*suggest for exist*) for existentially quantified variables, do not enforce typing and well-definedness of user-provided terms, potentially leading to inconsistent proofs, as illustrated by the following example.

▶ **Example 5.1.** Consider this very simple B machine[5], containing only one, contradictory assertion:

```
1 MACHINE DerivFalse
2 ASSERTIONS
3   ∃ x. (x ∈ INTEGER ∧ x ∉ INTEGER)
4 END
```

Yet, Atelier B's interactive prover allows discharging this assertion by executing the following commands in sequence in the interactive proof editor:

- `se(min(∅))`, instantiating the existential variable `x` with `min(∅)`, which is ill-defined since the empty set has no minimum: the prover does not check well-definedness here however;
- `mp` (*mini-prover*) invoking internal automatic provers to close the goal.

Once falsehood is derived, any proof obligation can be discharged trivially by invoking command `ah` (*add hypothesis*) with this contradiction as a hypothesis. This example is reproducible with all Community Edition releases of Atelier B, up to the current latest 24.04.2.

This pitfall concerns the interactive layer (unchecked instantiation and rule application), not the PO generator itself. Nonetheless, this reflects that the interactive mode is not proof-producing and therefore relies on user discipline for sound instantiations. By contrast, BARReL lets users discharge the same obligations as ordinary Lean goals. In particular, Lean tactics such as `exists` can only accept type-correct instantiations, hence ill-typed terms are rejected by construction; and under BARReL's encoding, partial B operators require explicit well-definedness proofs, so ill-defined instantiations—like `min(∅)`—cannot even be formed without providing the corresponding evidence.

Beyond these soundness guarantees, Lean provides a significantly richer proof-engineering ecosystem. Once B obligations are expressed in Lean, users can leverage Mathlib's libraries and automation to simplify routine reasoning steps using normalization and algebraic tactics such as `ac_rfl`, `ring` and related arithmetic tactics when needed. This becomes particularly valuable when developments go beyond the integer/set-theoretic core of B, for instance by introducing real-valued models.

6 Discussion and Future Work

We identify three main directions for future work:

[5] This example is available as "`DerivFalse.lean`" in the artifacts.

Table 2 Current coverage of BARReL. Bold operators are partial and generate WD side goals.

Category	Constructs
Logical operators	conjunction, disjunction, negation, implication, equivalence, bounded universal and existential quantification, equality
Set theory	basic sets (`NATURAL`, `NATURAL1`, `NAT`, `NAT1`, `BOOL`, `REAL`, etc.), singleton, cartesian product, union, intersection, set difference, inclusion, powerset, bounded comprehension, membership, finite powerset, **cardinality**
Relations	domain, range, relational image, (co-)restriction, (co-)subtraction, converse, composition, overloading, identity
Functions	function spaces (partial/total/injective/surjective/bijective as functional relations), **function application**, λ-abstraction
Sequences	set of sequences (`seq`), **size**
Arithmetic	integers, intervals, usual orderings, basic arithmetic operators (addition, multiplication, etc.), **minimum**, **maximum**

- extending the supported fragment of the B mathematical language and the accompanying lemma library;
- reducing redundancy among generated WD obligations via subsumption;
- enriching automation beyond WD goals for recurring proof-obligation patterns.

6.1 Increasing coverage of B

BARReL currently supports a practically useful fragment of the B mathematical language covering the set/relational core used by typical Atelier B proof obligations, together with common arithmetic and finiteness idioms as summarized in Table 2. Advanced data structuring features and less common operators are currently out of scope, but can be added incrementally.

Extending BARReL's supported fragment typically requires, for each new B symbol:

- a Lean definition and notation close to the B syntax,
- a parsing and an encoding rule,
- a WD predicate and corresponding WD generation rule for partial operators,
- a small supporting lemma kit (equational rules, basic facts and WD lemmas) so that the automation layer remains effective.

Table 2 summarizes the current coverage and the partial operators for which BARReL generates WD obligations. Missing constructs include sequences (beyond size, there are many more operations like concatenation, head, tail, etc.), trees, quantified operators (union, intersection, summation, product), and more advanced operations like permutations, closures, etc.

Implementation metrics

Considering core tooling only, BARReL consists of 1,331 lines of code (LoC), excluding blank and comment lines and including the embedding of the B syntax, the parser, the encoder, and the discharger/meta-programming layer—witnessing a relatively lightweight implementation.

The artifacts also include the previously mentioned library of auxiliary equational and simplification lemmas: the B "builtins" library (`Barrel/Builtins/*`) accounts for 1,282 LoC and contains supporting facts about sets, relations, functions, arithmetic and well-definedness, as well as the mentioned automation tactics.

Finally, the artifacts also contain all examples from the paper and more that exercise BARReL's workflow. These metrics are summarized in Table 3.

6.2 Reducing the number of WD side goals via subsumption

One challenge with BARReL is that it generates a lot of WD side goals compared to Atelier B, as evidenced by Table 1. This stems from a combination of multiple factors:

- Atelier B is able to reason directly on the machine itself, while BARReL only knows about the obligations generated. Thus, Atelier B can insert WD conditions only where needed (at the call sites of partial operators, before generating the goals), once and for all, and share them between sub-goals (or rather not duplicate them).
- When invariants are n-ary conjunctions, Atelier B's PO generator outputs multiple separate obligations (one per conjunct) while duplicating the hypotheses. Since BARReL encodes each obligation individually and separately, WD conditions that come from hypotheses of the obligation (e.g. in invariant preservation goals) are needlessly duplicated.

Although most of these WD side goals are automatically discharged by an internal tactic (see Section 3.3), the remaining ones may still require proving the same WD condition in multiple different—but subsumable—contexts. We leave as future work implementing subsumption of WD conditions to reduce the number of side goals generated, thus improving overall performance of BARReL when importing a B machine.

6.3 Automation beyond WD obligations

The current tactic layer of BARReL is intentionally narrow: it targets WD side conditions and leaves the remaining obligations—typically invariant preservation and refinement simulation—to interactive proof. In practice, these non-WD obligations also exhibit highly regular structures—such as routine set/relational algebra, standard refinement simulation patterns, and repeated use of the same invariants—suggesting that they also admit dedicated, domain-specific automation. Since BARReL operates entirely inside Lean, such automation is *proof-producing*, meaning that such tactics synthesize proof terms that are checked by the kernel. The relevant design goal is to increase proof-search power while keeping proofs readable and predictable.

A concrete plan for extending BARReL's automation layer includes:

- enriching the simplification and rewriting library for B-style set and relational algebra (domain/range laws, restriction/subtraction, relational image and composition) so that obligations normalize to "Mathlib-shaped" goals;
- adding small, focused procedures for functional-relational reasoning (e.g. determinism, domain/range inclusion, extensionality, and application lemmas);
- providing lemmas for common PO families and delegation to standard Mathlib automation and reasoning.

Table 3 Implementation metrics of the artifacts containing BARReL.

Component	LoC
Core tooling	1,331
Lemmas and automation	1,282
Examples and tests	585
Total	**3,198**

This would allow BARReL to discharge a larger fraction of proof obligations automatically, while keeping the overall architecture modular and maintainable. In the same vein towards modularity, one could also expect that BARReL provide only a generic interface for set/relational reasoning, allowing users to plug in their own implementations or strategies as needed.

6.4 Towards a fully verified backend

Although BARReL is a proof-producing backend for the obligations it emits, it currently still relies on Atelier B's `bxml` and `pog` binaries for parsing B machines and generating proof obligations. A natural next step is therefore to implement, *in Lean*, a PO generator for the supported fragment that produces Lean propositions directly over BARReL's embedding—with correct handling of WD conditions—and to establish soundness guarantees stating that discharging these obligations entails invariant preservation and refinement correctness. First, this would eliminate the main external component of the trusted computing base, thereby yielding a more self-contained verification chain. Second, this paves the way to developing an embedded DSL for writing B developments directly in Lean. By reasoning on Lean terms directly, WD conditions can be directly inlined—and even automatically deduced—in the B developments.

In the same direction, one could lean toward soundness of the translation connecting B obligations to their Lean counterparts. While the current translation is intentionally syntax-directed and close to a one-to-one mapping into our embedding, making this correspondence explicit would further reduce the amount of trusted code in the backend, although this would require a significant formalization effort—including giving formal semantics to a fragment of Lean.

7 Conclusion

We presented BARReL, a Lean 4 backend for Atelier B that turns industrial B artifacts into ordinary Lean goals over a natural set-theoretic encoding of the B language. Our central design choice is to make B's ubiquitous partiality explicit: partial operators are represented as total Lean functions parameterized by proofs of well-definedness which are reified as standard additional goals. This yields robust proof scripts that are protected against ill-typed or ill-defined instantiations, while remaining close to the structure of the obligations produced by Atelier B.

We evaluated the approach on a three-level refinement chain for minimum search, showing generated Lean goals, many of which being discharged automatically by BARReL's lightweight automation, leaving refinement and invariant-preservation obligations to be proved interactively. From a trust perspective, BARReL reduces the trusted computing base by implementing the translation pipeline within Lean's metaprogramming framework as a natural, almost identical mapping from B constructs to Lean expressions relying only on Atelier B for proof obligation generation. Future work includes reducing redundancy among well-definedness goals, e.g. via subsumption, extending the covered fragment of B and the accompanying lemma library, and integrating a verified proof obligation generator within Lean itself.

Artifacts availability

A snapshot of BARReL is available at `https://anonymous.4open.science/r/BARReL`. The archive contains the complete Lean 4 implementation of BARReL (B surface embedding, POG reader, encoder, discharger macros, and tactic layer), together with the sample B machines and Lean scripts used in the evaluation.

Declaration of AI Use

We declare that no generative AI tools or large language models (LLMs) were used in the development of the tool, the Lean implementation, or the writing of this paper.

References

1. J. R. Abrial, M. K. O. Lee, D. S. Neilson, P. N. Scharbach, and I. H. Sørensen. The B-method. In Søren Prehn and Hans Toetenel, editors, *VDM '91 Formal Software Development Methods*, pages 398–405, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.
2. Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010. doi:10.1017/CBO9781139195881.
3. Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer*, 12(6):447–466, November 2010. doi:10.1007/s10009-010-0145-y.
4. Anne Baanen, Matthew Robert Ballard, Johan Commelin, Bryan Gin-ge Chen, Michael Rothgang, and Damiano Testa. Growing Mathlib: maintenance of a large scale mathematical library. In Valeria de Paiva and Peter Koepke, editors, *Intelligent Computer Mathematics*, pages 51–70, Cham, 2026. Springer Nature Switzerland.
5. Benoît Ballenghien and Burkhart Wolff. Event-B as DSL in Isabelle and HOL experiences from a prototype. In Silvia Bonfanti, Angelo Gargantini, Michael Leuschel, Elvinia Riccobene, and Patrizia Scandurra, editors, *Rigorous State-Based Methods - 10th International Conference, ABZ 2024, Bergamo, Italy, June 25-28, 2024, Proceedings*, volume 14759 of *Lecture Notes in Computer Science*, pages 241–247. Springer, 2024. doi:10.1007/978-3-031-63790-2_18.
6. Haniel Barbosa et al. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022. doi:10.1007/978-3-030-99524-9_24.
7. Jean-Paul Bodeveix and Mamoun Filali. Type synthesis in B and the translation of B to PVS. In Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors, *ZB 2002:Formal Specification and Development in Z and B*, pages 350–369, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.
8. Clearsy. Atelier B. `https://www.atelierb.eu`.
9. Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.
10. David Déharbe. Integration of SMT-solvers in B and Event-B development environments. *Science of Computer Programming*, 78:310–326, 03 2013. doi:10.1016/j.scico.2011.03.007.
11. Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*, page 625–635, Berlin, Heidelberg, 2021. Springer-Verlag. doi:10.1007/978-3-030-79876-5_37.

12 Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

13 Matthias Schmalz. *Formalizing the logic of Event-B: Partial functions, definitional extensions, and automated theorem proving*. PhD thesis, ETH Zurich, Zürich, Switzerland, 2012. URL: https://hdl.handle.net/20.500.11850/64337, doi:10.3929/ETHZ-A-007577749.

14 The Rocq Development Team. The Rocq prover, September 2025. doi:10.5281/zenodo.17473943.

15 Vincent Trélat. Safely Encoding B Proof Obligations in SMT-LIB. In Michael Leuschel and Fuyuki Ishikawa, editors, *Rigorous State-Based Methods - 11th International Conference, ABZ 2025, Düsseldorf, Germany, June 10-13, 2025, Proceedings*, ABZ 2025, pages 52–69. Springer, 2025. URL: https://doi.org/10.1007/978-3-031-94533-5_4, doi:10.1007/978-3-031-94533-5_4.