Formalization of a Realistic Verification-Condition Generator for an Intermediate Verification Language

Vladimir Gladshtein
National University of Singapore, Singapore

K. Rustan M. Leino
Amazon Web Services, Seattle, WA, USA

Abstract

Intermediate Verification Languages (IVLs) play the same role in verification as Intermediate Representations in compilation, a layer that separates a verifier's language-specific front-end from its logic automation back-end. Successful IVL tools such as Boogie, Why3, and Viper generate Verification Conditions (VCs) that are sent to an SMT solver. The verifier output can be trusted only if these VCs are sound with respect to the formal semantics of the IVL. Formalizing the semantics of IVLs and verifying the soundness of corresponding VC Generators with respect to this semantics is challenging if one wants to model realistic features of IVLs such as mutually recursive definitions, lexical variable and control-flow labeled scopes, interpreted and uninterpreted functions, and unbounded loops.

B3 is a new IVL. This paper presents a formalization of B3's semantics, a VC Generator for the language, and a soundness proof that these two correspond. A key practical contribution of this work is that all three components are authored in the Dafny programming language and verifier. This makes it easy for a tool maintainer to maneuver between the semantic definitions, the proofs, and the VCG's executable code. The key theoretical contribution of the work is a methodology to split the IVL's semantic encodings into two layers of abstraction to cover realistic aspects of the semantics, while keeping the proofs amenable to automation. Optimized for Dafny-style automation, the first layer is used to verify the correctness of the VC Generator procedure. Optimized for expressiveness, the second layer is used to capture the semantics in a natural way.

2012 ACM Subject Classification Software and its engineering → Software verification and validation

Keywords and phrases Intermediate verification language, Soundness, Verification, B3, Dafny, SMT solvers

Digital Object Identifier 10.4230/LIPIcs.ITP.2026.12

Supplementary Material *Software (Dafny implementation)*: `B3 repository`

Funding *Vladimir Gladshtein*: Work done during an internship at Amazon Web Services.

1 Introduction

By deductively verifying a source program to meet a given specification, one obtains a high degree of assurance that the program will always behave as intended when deployed. As is well understood, this assurance is not absolute, because a program verifier checks only the source program itself, not the machine code a compiler generates from it, not the hardware on which the machine code runs, not the physical conditions under which that hardware operates, *etc.* Nevertheless, a software engineer expects the affirmative output of a program verifier to provide some guarantees about the source program itself. For this to be true, the verifier itself must be correct.

In this paper, we prove the correctness of one important component of automated program verifiers, namely the generation of verification conditions (VCs) from an intermediate

verification language (IVL). The IVL is what separates the verifier's front-end, which is specific to a source language, from the verifier's back-end, which attempts to discharge the logical verification conditions. The IVL is used as a program-like notation to prescribe proof obligations in the source language. IVLs used in this way to power popular automated program verifiers include Boogie [2], Why3 [9], and Viper [21]. We target B3, a new IVL whose design is informed by two decades of using Boogie in program verifiers like Dafny [15].

The verification pipeline in which B3 sits has three steps. First, a verifier front-end translates the source program (*e.g.*, in Java or Dafny) into a B3 program. This step uses B3 constructs that are not directly shared by programming languages — such as nondeterminism (useful in representing the behavior of underspecified call targets) and `assume` statements (used to exclude certain program traces) — to encode the source language's proof obligations. The front-end also supplies procedure specifications and loop invariants, either by translation from similar constructs in the source language or by an IVL-to-IVL transform (*e.g.*, Boogie uses an abstract interpreter [2]). Second, the B3 VCG traverses the resulting program and emits a set of VCs; each VC is a B3 language expression. Third, a back-end represents these expressions as `SMT-LIB` formulas and dispatches them to an SMT solver such as Z3.

Of these three steps, only the middle one is the subject of formal study in this paper. We define the semantics of B3, define a generation of VCs from B3, and prove the correctness of that VC generation with respect to the semantics. In other words, we prove the *soundness* of B3's VC generation: if every VC produced by the VCG is logically valid, then the source program meets its specification under the B3 reference semantics. The source-to-B3 and B3-to-SMT translations remain unverified components of any practical deployment.

Proving the soundness of VC generation is in many ways similar to proving the correctness of a compiler for a source programming language (*e.g.*, [20, 14]). Indeed, control-flow statements of an IVL mimic those in programming languages, which makes the IVL a convenient target for verifier front-ends. But the IVL features mentioned above — nondeterminism, `assume`, specifications, invariants — have no direct counterparts in compilers, and the B3 semantics has to take them into account so VC generation can use them. These differences make the semantic definitions different from those used to prove a compiler correct.

In theory, VC generation is straightforward: simply use *weakest preconditions* [7] to obtain the VC formula. In practice, obtaining VCs that an automatic logic engine can process effectively is more involved. Most conspicuously, VC generation introduces new variable names (*incarnations* [10]) for each assignment, which requires substitution operations to be performed during VC generation. In order to avoid computing fix-points, VC generation cuts loop back-edges and uses the supplied loop invariants to check arbitrary repetitions of the loop body. Even in structured source languages, exceptions, loop breaks, and return statements necessitate abrupt exits in the IVL, which a realistic VC generator needs to account for. Finally, the principle of modular verification [23] tells us that verifying each procedure against its own specification, under the assumption that all other procedures—including recursive calls—satisfy theirs, suffices to conclude that the entire program is correct. Soundness of VC generation has to formalize this piece of folklore. We address all of these points in our formalization and soundness theorem.

The B3 tool is implemented in the Dafny programming language [15]. Since Dafny and its automated verifier support lemmas and proofs, we have also used Dafny to define the B3 semantics and to state and prove the soundness theorem.

In defining the semantics of B3 and doing the soundness proof, we encountered a tension between writing semantic rules that are "obviously correct" and semantic rules that are

Variables	$x,\ y,\ \ldots$	alpha-numeric strings
Names	$m,\ \ldots$	alpha-numeric strings
Literals	$l,\ \ldots$	alpha-numeric strings
Functions	$f,\ \ldots$	alpha-numeric strings
Types	$\mathcal{T}$ ::=	bool | int | $t,\ \ldots$ alpha-numeric strings
Constants	i	unbounded integer literals
Expressions	$\mathcal{E}$ ::=	false | true | i | l | x | $\mathcal{E} = \mathcal{E}$ | $\oplus(\mathcal{E},\ldots,\mathcal{E})$ | $f(\mathcal{E},\ldots,\mathcal{E})$ | $\forall x\colon \mathcal{T} \cdot \mathcal{E}$
Statement	$\mathcal{A}$ ::=	`check` $\mathcal{E}$ | `assume` $\mathcal{E}$ | $x := \mathcal{E}$ | $\mathcal{S} + \mathcal{S}$ | `if` $\mathcal{E}$ `then` $\mathcal{S}$ `else` $\mathcal{S}$ | `loop` $\mathcal{S}$ | `vars` $x_1,\ldots,x_n$ `in` $\mathcal{S}$ | `escape` i | `call` $m(y_1,\ldots,y_n)$
Statements	$\mathcal{S}$ ::=	$\cdot$ | $\mathcal{A}$; $\mathcal{S}$
procedure	$\mathcal{P}$ ::=	`proc` $m(x_1,\ldots,x_n)$ `requires` $\mathcal{E}$ `ensures` $\mathcal{E}$ { $\mathcal{S}$? }
program		$t,\ldots,t,\mathcal{P},\ldots,\mathcal{P}$

Figure 1 An abstract representation of the B3 language. For brevity in this paper, we have combined B3's `var`, `break`, and labeled-block statements into statements that we call `vars` and `escape`. For clarity in the paper, we have added a `call` keyword to the procedure-call statement.

more amenable to automation for the soundness proof itself. This tension was particularly noticeable for loops and procedure calls, both of which can give rise to infinite traces. To take advantage of both kinds of semantic rules, we use a two-layered approach whereby we first prove the soundness of VC generation w.r.t. a structural "exists-invariant" semantics and then prove the soundness of that semantics w.r.t. an "obviously correct" reference semantics.

Our reference semantics is coinductive. This lets it capture infinite program traces (*i.e.*, program runs that go on forever), which is natural because B3 allows such programs. During the course of our work, we several times asked ourselves if we couldn't just use an inductive semantics instead, since that would permit our results to be replicated in proof systems without support for coinduction. But the main theorem does not hold for an inductive semantics, as we'll explain in Sec. 3.1. Luckily, Dafny has the same support for coinductive predicates as it does for inductive predicates [19, 18]. Moreover, the coinductive proofs we did were remarkably simple, requiring only about 100 lines of manually supplied proofs outside what's already provided through Dafny's automation.

In summary, our contributions in this paper are:

- A formal semantics for B3, a VC generator for B3, and an accompanying soundness proof.
- A showcase that an auto-active verifier like Dafny is a good vehicle for defining programming language meta-theory and proofs thereof, even when such proofs use advanced features such as coinductive predicates defined by greatest fix-points.
- A methodology for breaking the IVL semantics into two layers, where the first layer more closely follows the form of the VC generator and the second layer more closely follows a program-evaluation semantics.

2 Overview of B3 Semantics

We start by giving a formal definition of the B3 language and providing a formal semantics for it.

2.1 B3 language

The B3 language is designed to be generic enough to model simple computations annotated with assertions that guide the verification process. We show the parts relevant to the paper in Fig. 1. Our full formalization is found in our accompanying Dafny implementation.

Expressions in B3 include standard mathematical operations (abbreviated by $\oplus$) along with interpreted and uninterpreted functions. Each expression can be $\forall/\exists$-quantified (in this paper, $\exists$ can be expressed via negation and $\forall$). For brevity in the paper, we omit interpreted functions, as their treatment is similar to the treatment of B3 procedures.

Statements in B3 include standard assignments, sequential composition, and conditionals. Since B3 is an IVL, it also includes `check` and `assume`, which verify or assume a certain condition at the given program point, and a nondeterministic operator of choice, $+$.

The statement `vars` $x_1, \ldots, x_n$ `in` $\mathcal{S}$ serves two purposes: it introduces (a possibly empty list of) new mutable local variables for use in $\mathcal{S}$, and it introduces a new control-flow scope that can be exited abruptly by `escape` statements. The statement `escape` i exits i nested scopes from the current program point (so, `escape` 0 is a no-op). In this paper, we assume all variable declarations are unique (*e.g.*, there is no shadowing). The `loop` statement repeats its body indefinitely. The loop can be exited by nesting it inside a `vars` statement and using an `escape` statement in the loop body to jump outside of that scope.

We play fast and loose with the notation for sequential composition: we may write $\mathcal{S}\ ;\ \mathcal{S}'$, leaving the necessary reassociation implicit, and may write $\mathcal{A}$ when we mean $\mathcal{A}\ ;\ \cdot$.

Procedures take parameters and are declared with pre- and postconditions and an optional body statement. In the body, the procedure's formal parameters are used as local variables. B3 has in-, inout-, and out-parameters, which are passed by value from caller to callee (in and inout) and back (inout and out). For simplicity in this paper, we only show inout-parameters, so call arguments are distinct variables whose values can be modified by the callee.[1] Our accompanying Dafny implementation handles all parameter kinds.

2.2 Structural rules

To capture the semantics, we adopt the *omnisemantics* approach, which has been shown to be convenient for handling nondeterminism and is equivalent to the standard operational semantics for nondeterministic computations [3]. More precisely, omnisemantics (denoted as $\Downarrow$) relates a pair $\langle \mathcal{S}, st \rangle$ of a program statement and a starting state to a set Q overapproximating the set of its possible outcomes, *i.e.*, final states: $\langle \mathcal{S}, st \rangle \Downarrow Q$ expresses that every terminating trace of $\mathcal{S}$ starting in st ends in some state in Q.

The rules of omnisemantics for basic B3 statements (such as `check`, `assume`, `if-then-else`, $:=$, and $;$) are standard and are defined recursively on the structure of a program. We show three of those rules in Fig. 2a.

Defining the omnisemantics for non-local control-flow statements is more challenging. The statement `escape` i causes the program to jump out of i enclosing scopes. Our semantics therefore needs to relate `escape` i to the same set of final states as the `vars` statement i nestings away. To achieve this, intuitively, we will track the postcondition of every active scope as a stack: entering a `vars` pushes a fresh frame, and `escape` i discharges the i-th frame from the top. Concretely, our $\Downarrow$ relation associates a pair $\langle \mathcal{S}, st \rangle$ with a *sequence* of post-state sets $\overline{Q}$, where element i of $\overline{Q}$ (denoted as $\overline{Q}_i$) contains the set of final states for the scope that is i `vars`-nestings away (*cf.* [5]). Notationally, we sometimes decorate an identifier with an overline (like $\overline{Q}$ here) to emphasize that it denotes a sequence.

The two rules that manipulate $\overline{Q}$ are shown in Fig. 2b. Rule Scope *pushes* a frame. To see this, consider first the simpler case in which the scope introduces no variables. The

[1] This may appear restrictive, but note that a caller can easily pass an expression by first assigning it to a fresh local variable and passing that variable instead.

$$\textsc{Check}\ \frac{[\![\mathcal{E}]\!]_{st} = \mathsf{true} \quad st \in Q}{\langle \texttt{check}\ \mathcal{E}, st\rangle \ \Downarrow\ Q} \qquad \textsc{Assume}\ \frac{[\![\mathcal{E}]\!]_{st} = \mathsf{true} \implies st \in Q}{\langle \texttt{assume}\ \mathcal{E}, st\rangle \ \Downarrow\ Q}$$

$$\textsc{Seq}\ \frac{\langle \mathcal{S}_1, st\rangle \ \Downarrow\ Q' \quad \forall\, st' \in Q' \cdot \langle \mathcal{S}_2, st'\rangle \ \Downarrow\ Q}{\langle \mathcal{S}_1\ ;\ \mathcal{S}_2, st\rangle \ \Downarrow\ Q}$$

(a) Structural rules for three basic B3 statements, shown with a simple post-set Q. To port these to the full rules, change $st \in Q$ to $st \in \overline{Q}_0$ and change the other occurrences of Q to $\overline{Q}$, and likewise for Q'.

$$\textsc{Scope}\ \frac{\forall\, st' \cdot \mathsf{Dom}(st') = \{x_1, \ldots, x_n\} \implies \langle \mathcal{S}, st + st'\rangle \ \Downarrow\ (\overline{Q}_0 \cdot \overline{Q}) + x_1, \ldots, x_n}{\langle \texttt{vars}\ x_1, \ldots, x_n\ \texttt{in}\ \mathcal{S}, st\rangle \ \Downarrow\ \overline{Q}}$$

$$\textsc{Escape}\ \frac{st \in \overline{Q}_i}{\langle \texttt{escape}\ i, st\rangle \ \Downarrow\ \overline{Q}}$$

(b) Structural rules for B3 scope-related statements.

$$\textsc{LoopUnroll}\ \frac{\langle \mathcal{S}\ ;\ \texttt{loop}\ \mathcal{S},\ st\rangle \overset{\text{ref}}{\Downarrow}\ \overline{Q}}{\langle \texttt{loop}\ \mathcal{S},\ st\rangle \overset{\text{ref}}{\Downarrow}\ \overline{Q}}$$

(c) Reference semantics for `loop`, which unrolls the loop in a readily understood manner. This and other rules for $\overset{\text{ref}}{\Downarrow}$ are to be understood coinductively.

$$\textsc{LoopInv}\ \frac{st \in \mathit{Inv} \quad \forall\, st' \in \mathit{Inv} \cdot \langle \mathcal{S}, st'\rangle \ \Downarrow\ \mathit{Inv} \cdot \overline{Q}}{\langle \texttt{loop}\ \mathcal{S}, st\rangle \ \Downarrow\ Q' \cdot \overline{Q}}$$

(d) Structural semantics of `loop`. Note that Q' plays no role, since the loop repeats indefinitely, exited only by `escape` statements that answer to the post-state sets in $\overline{Q}$.

Figure 2 Fragment of B3 semantics rules

scope falls through to the same postcondition as its enclosing context, so the body is verified against $\overline{Q}_0 \cdot \overline{Q}$, with each original $\overline{Q}_i$ shifted to position $i+1$:

$$\frac{\langle \mathcal{S}, st\rangle \ \Downarrow\ \overline{Q}_0 \cdot \overline{Q}}{\langle \texttt{vars in}\ \mathcal{S}, st\rangle \ \Downarrow\ \overline{Q}}$$

The full rule extends this with the freshly bound locals $x_1, \ldots, x_n$: the starting state st is concatenated with an arbitrary state st' on those variables, and every set in the sequence is lifted to states whose domain includes them. The lifting notation $+\, x_1, \ldots, x_n$ distributes over the sequence and, for each post-state set Q, has the meaning

$$st \in Q + x_1, \ldots, x_n \quad \overset{\text{def}}{=} \quad \{x_1, \ldots, x_n\} \subseteq \mathsf{Dom}(st) \wedge st - \{x_1, \ldots, x_n\} \in Q \tag{1}$$

Rule Escape *consumes* a frame: `escape` i succeeds in state st exactly when st already lies in $\overline{Q}_i$. An `escape` i unwinds i scopes at once, relating $\langle \texttt{escape}\ i, st\rangle$ to $\overline{Q}_i$, as shown in the bottom part of Fig. 2b.

For example, take `vars in` $(\mathcal{S}_1\ ;\ (\texttt{escape}\ 1\ ;\ \mathcal{S}_2))$ under enclosing postcondition $\overline{Q}$. Applying ScopeNoVars prepends $\overline{Q}_0$, so the body is verified against $\overline{Q}_0 \cdot \overline{Q}$. After $\mathcal{S}_1$ produces some intermediate state st', rule Escape fires with $i = 1$ and requires $st' \in (\overline{Q}_0 \cdot \overline{Q})_1$. The latter is equivalent to $st' \in \overline{Q}_0$, where $\overline{Q}_0$ is the post-condition before the scope. Thus, `escape` breaks out of the surrounding scope, skipping $\mathcal{S}_2$.

The rules for the other basic statements can easily be ported from the literature's standard definitions to a semantics relation with a $\overline{Q}$ sequence (*e.g.*, see caption of Fig. 2a). The full definitions are found in the accompanying Dafny implementation.

2.3 Two different semantics for loops

The standard way to define loop semantics is through unrolling, as shown in Fig. 2c (we will explain the $^{\text{ref}}$ superscript shortly). While this unrolling rule is easy to understand, it is inconvenient to work with in practice, because it is not structural on the program term. Specifically, the term in its assumption ($\mathcal{S}$; `loop` $\mathcal{S}$) is structurally larger than the term in its conclusion (`loop` $\mathcal{S}$). Such a rule cannot be part of a simple predicate definition whose evaluation terminates. Instead, we have to define these semantics rules by a least fix-point (inductive predicate) or by a greatest fix-point (coinductive predicate). Since the B3 language is designed for partial semantics (allowing non-terminating runs), we will use a coinductive predicate. That is, the rules we give for defining $\overset{\text{ref}}{\Downarrow}$ are to be interpreted coinductively, as greatest fix-points.

Dafny supports defining and reasoning about coinductive predicates, but its automated reasoning performs significantly better with the simpler terminating predicates. To get the best of both worlds, we split our B3 semantics into two layers, denoted by $\Downarrow$ and $\overset{\text{ref}}{\Downarrow}$. The first layer is structurally recursive: we will use it in most of the proofs for the soundness of the VC generator. We'll refer to this layer as the *structural semantics*. The second layer gives our *reference semantics* (hence the $^{\text{ref}}$ superscript), which is used in the statement of the main soundness theorem. In order for the proof of the main soundness theorem to use the theorems about the first layer, we need to show that the reference semantics follows from the structural semantics, as will be our focus in Sec. 4.

The structural and reference semantics are defined in the same way for the basic and scope-related statements we described in Sec. 2.2. The only statements where the two semantics differ are `loop` and `call`[2]. While the reference semantics for `loop` is based on unrolling, the structural semantics for `loop` comes down to the existence of a sufficiently strong loop invariant (Fig. 2d). Specifically, to prove it, one has to find a set of states *Inv* such that for all states $st \in Inv$, if the loop body terminates normally from st, then it terminates in a state in *Inv*. Note that this rule is structural, since its assumption calls $\Downarrow$ on $\mathcal{S}$, which is structurally smaller than the `loop` $\mathcal{S}$ in the conclusion.

Note that both loop rules leave $\overline{Q}$ unchanged in the premise: a loop neither pushes nor consumes scope frames. Consequently, an `escape` i inside a loop body targets exactly the same scope as it would outside the loop — in particular, the standard idiom of breaking out of a loop is to wrap it in a `vars` and use `escape` 1 to jump to the postcondition of that wrapper.

2.4 Procedures and calls

The remaining statement to explain is procedure call. As we mentioned in Sec. 2.1, a B3 procedure includes a specification and may omit an implementation. The semantics of a call has two parts, one for the caller side and one for the callee side. (1) On the caller side, the semantics checks the precondition and then continues in an arbitrary state satisfying the postcondition. (2) On the callee side, when the procedure has a body, the semantics starts from any state satisfying the precondition, then proceeds as the procedure body, and finally checks the postcondition.

Rule CallRef in Fig. 3a captures these two parts for the reference semantics, the

[2] and also in expressions that call interpreted functions, but as we mentioned earlier, we omit those from the paper since they are similar to procedure calls

$$\textsc{Call}\ \textsc{Ref}\quad \frac{\mathcal{P} = \texttt{proc}\ m(x_1,\ldots,x_n)\ \texttt{requires}\ req\ \texttt{ensures}\ ens\ \{\ \mathcal{S}?\ \} \qquad [\![\texttt{subst}(req,\ [x_i \mapsto y_i])]\!]_{st} = \mathsf{true} \qquad \mathsf{MeetsSpec}_{\mathrm{ref}}(\mathcal{P}) \qquad \forall\, st' \cdot\ \mathsf{Dom}(st') = \mathsf{Dom}(st) \wedge (\forall\, x \in \mathsf{Dom}(st) \setminus \{y_1,\ldots,y_n\} \cdot\ st[x] = st'[x]) \wedge [\![\texttt{subst}(ens,\ [x_i \mapsto y_i])]\!]_{st'} = \mathsf{true} \implies st' \in \overline{Q}_0}{\langle \texttt{call}\ m(y_1,\ldots,y_n), st\rangle \overset{\text{ref}}{\Downarrow} \overline{Q}}$$

(a) Rule CallRef shows the reference semantics of `call` statements. Omitting the non-*gray* parts, rule Call shows the structural semantics.

$$\textsc{Procedure}\ \textsc{Ref}\quad \frac{Q = \left\{st' \mid [\![ens]\!]_{st'} = \mathsf{true}\right\} \qquad \forall\, st \cdot\ \mathsf{Dom}(st) = \{x_1,\ldots,x_n\} \wedge [\![req]\!]_{st} = \mathsf{true} \implies \langle \mathcal{S}, st\rangle \overset{\text{ref}}{\Downarrow} \langle Q\rangle}{\mathsf{MeetsSpec}_{\mathrm{ref}}(\texttt{proc}\ m(x_1,\ldots,x_n)\ \texttt{requires}\ req\ \texttt{ensures}\ ens\ \{\ \mathcal{S}\ \})}$$

(b) Procedure soundness rules. For a procedure $\mathcal{P}$ without a body, $\mathsf{MeetsSpec}_{\mathrm{ref}}(\mathcal{P})$ always holds.

Figure 3 Semantic rules for procedures and calls

second part being the $\mathsf{MeetsSpec}_{\mathrm{ref}}(\mathcal{P})$ predicate, which is defined in Fig. 3b. Since the precondition *req* and postcondition *ens* are defined over the formal parameters $x_1,\ldots,x_n$ of the procedure, whereas the caller state *st* and the post-state st' are defined over the caller's variables $y_1,\ldots,y_n$, we need to substitute the formal parameters with the actual arguments before evaluation. We use $\texttt{subst}(\mathcal{E},\ [x_i \mapsto y_i])$ to denote the substitution of each occurrence of x_i with y_i in expression $\mathcal{E}$. With this substitution, the precondition $\texttt{subst}(req,\ [x_i \mapsto y_i])$ is evaluated in the caller state *st*. For the postcondition, st' ranges over all states with the same domain as *st* where only the argument variables $y_1,\ldots,y_n$ may differ from *st*, and the postcondition $\texttt{subst}(ens,\ [x_i \mapsto y_i])$ is evaluated in st'. Note that the CallRef rule is not structural: it calls $\mathsf{MeetsSpec}_{\mathrm{ref}}(\mathcal{P})$, which, in turn, calls the reference semantics on the procedure body, a statement that is typically larger than the call statement itself.

To obtain a structurally recursive definition, we define our structural-semantics rule for the `call` statement to omit the $\mathsf{MeetsSpec}_{\mathrm{ref}}$ predicate. This Call rule is obtained by eliding the *gray* parts in Fig. 3a. At first glance, the CallRef rule may seem strictly weaker than Call, but by the coinductive nature of the reference semantics, we can prove (in Sec. 4) that the soundness of any closed[3] set of procedures under the structural semantics implies soundness under the reference semantics.[4]

3 B3 Verification Condition Generator

To check that procedures obey their specifications, the B3 tool implements a VCG. At the top level, the VCG takes a set of B3 procedures and generates a set of VC formulas that should be checked for validity. In this section, we define the VCG and state a soundness

[3] A set of procedures *ps* is *closed* if every procedure call in *ps* goes to some procedure in *ps*.
[4] Our soundness theorem thus also formalizes the modular-verification ideas of Parnas [23].

theorem that shows that the validity of the VCs implies the reference semantics.

3.1 VCG for procedures

The VC generator works one procedure at a time. Whenever it encounters a procedure call, it generates VCs purely based on the specification of the callee, not the callee's body. In this way, the VCG performs *modular verification*.

In contrast to the modular VCG, the reference semantics for calls coinductively also unfolds the body of callees and checks the callee's postcondition (Fig. 3). To make up for the omitted body unfolding, the VCG needs to be applied, separately, to all procedures.

We use the notation $\mathcal{P} \implies \mathcal{VC}$ to say that the VCG generates the set of VCs $\mathcal{VC}$ for a given procedure $\mathcal{P}$. We can now state the main soundness theorem as follows:

▶ **Theorem 1** (Soundness of VCG for procedures). *For any closed set of procedures* $\overline{\mathcal{P}}$*, if* $\forall\, \mathcal{P} \in \overline{\mathcal{P}} \cdot\ \forall\, \mathcal{VC} \cdot\ \mathcal{P} \implies \mathcal{VC} \implies \forall\, vc \in \mathcal{VC} \cdot$ *(*vc *is valid), then* $\forall\, \mathcal{P} \in \overline{\mathcal{P}} \cdot\ \mathsf{MeetsSpec}_{ref}(\mathcal{P})$.

In words, this theorem says that by using the VCG to check each procedure separately, we can conclude that all the procedures are correct w.r.t. the reference semantics. We'll define the procedure-level VCG $\mathcal{P} \implies \mathcal{VC}$ in terms of a statement-level VCG in Sec. 3.2.

As we have explained in Sec. 2, our strategy for proving the main soundness theorem is first to prove the soundness of the VCG w.r.t. the structural semantics $\Downarrow$ and then to prove that the structural semantics $\Downarrow$ implies the reference semantics $\overset{\text{ref}}{\Downarrow}$. The first part of our strategy is captured by the following lemma, which is stated for each procedure (whose body may call other procedures):

▶ **Lemma 2** (Soundness of VCG for one procedure w.r.t. structural semantics). *For a procedure* $\mathcal{P}$*, if* $\mathcal{P} \implies \mathcal{VC}$ *and* vc *is valid for each* $vc \in \mathcal{VC}$*, then* $\mathsf{MeetsSpec}(\mathcal{P})$.

Note that, whereas Theorem 1 uses $\mathsf{MeetsSpec}_{\text{ref}}$ from the reference semantics, this lemma uses MeetsSpec from the structural semantics. The definition of the predicate MeetsSpec is shown in the non-*gray* parts of Fig. 3b. Proving Theorem 2 requires a more general statement, which we'll give in Theorem 4 in Sec. 3.3.

The second part of our strategy is made precise in the following lemma, which we prove in Sec. 4:

▶ **Lemma 3** (Semantic refinement between $\Downarrow$ and $\overset{\text{ref}}{\Downarrow}$). *For a closed set of procedures* $\overline{\mathcal{P}}$*, if* $\forall\, \mathcal{P} \in \overline{\mathcal{P}} \cdot\ \mathsf{MeetsSpec}(\mathcal{P})$*, then* $\forall\, \mathcal{P} \in \overline{\mathcal{P}} \cdot\ \mathsf{MeetsSpec}_{ref}(\mathcal{P})$.

Before diving into more details, let us pause to reflect on the main theorem and our reference semantics. B3 allows an infinite loop like `loop ·`, so we expect any VCs for such a loop to hold trivially (indeed, as we'll soon see, our VCG generates no VCs for such a loop). A 1-procedure program with an infinite loop thus trivially satisfies the antecedent of Theorem 1. The theorem then implies $\langle \texttt{loop}\ \cdot,\ st\rangle \overset{\text{ref}}{\Downarrow} \langle \top \rangle$, which according to rule LoopUnroll (Fig. 2c) is a judgment that follows from itself. If we interpreted the reference-semantics rules *inductively*, then this judgment would be false, which means there's no hope for Theorem 1. But we interpret the rules *coinductively*, so the judgment is true, as indicated by the theorem.

We have that Theorem 1 follows from Theorem 2 (applied to each procedure in $\overline{\mathcal{P}}$) and Theorem 3. Since, as we just argued, Theorem 1 depends on the reference semantics being coinductive, one (or both) of these lemmas must also depend on coinduction. In the next subsection, we'll define the VCG over the structure of statements, so there is nothing about

$$\textsc{Proc}\ \frac{\langle \texttt{assume}\ req;\ \mathcal{S};\ \texttt{check}\ ens,\ \texttt{initial}(x_1,\ldots,x_n),\ \varnothing,\ \varnothing\rangle \implies \mathcal{VC}}{\texttt{proc}\ m(x_1,\ldots,x_n)\ \texttt{requires}\ req\ \texttt{ensures}\ ens\ \{\ \mathcal{S}\ \} \implies \mathcal{VC}}$$

(a) VCG for procedures. The function $\texttt{initial}(x_1,\ldots,x_n)$ constructs an incarnation map that maps variables $x_1,\ldots,x_n$ to themselves.

$$\textsc{Check}\ \frac{\mathcal{VC} = \{\mathcal{C} \implies \texttt{subst}(\mathcal{E},\mathcal{I})\}}{\langle \texttt{check}\ \mathcal{E},\ \mathcal{I},\ \mathcal{C}\rangle \implies \langle \mathcal{VC},\ \mathcal{I},\ \mathcal{C}\rangle} \qquad \textsc{Assume}\ \frac{\mathcal{C}' = (\mathcal{C} \wedge \texttt{subst}(\mathcal{E},\mathcal{I}))}{\langle \texttt{assume}\ \mathcal{E},\ \mathcal{I},\ \mathcal{C}\rangle \implies \langle \varnothing,\ \mathcal{I},\ \mathcal{C}'\rangle}$$

$$\textsc{Assign}\ \frac{x' \text{ is fresh} \quad \mathcal{I}' = \mathcal{I}[x \mapsto x'] \quad \mathcal{C}' = (\mathcal{C} \wedge x' = \texttt{subst}(\mathcal{E},\mathcal{I}))}{\langle x := \mathcal{E},\ \mathcal{I},\ \mathcal{C}\rangle \implies \langle \varnothing,\ \mathcal{I}',\ \mathcal{C}'\rangle}$$

(b) VCG for a fragment of atomic statements

$$\textsc{Seq}\ \frac{\langle \mathcal{A},\ \mathcal{I},\ \mathcal{C}\rangle \implies \langle \mathcal{VC},\ \mathcal{I}',\ \mathcal{C}'\rangle \quad \langle \mathcal{S},\ \mathcal{I}',\ \mathcal{C}',\ \mathcal{B}\rangle \implies \mathcal{VC}'}{\langle \mathcal{A}\ ;\ \mathcal{S},\ \mathcal{I},\ \mathcal{C},\ \mathcal{B}\rangle \implies \mathcal{VC} \cup \mathcal{VC}'}$$

$$\textsc{SeqIf}\ \frac{\langle \mathcal{S}\ ;\ \mathcal{S}'',\ \mathcal{I},\ \mathcal{C} \wedge \texttt{subst}(\mathcal{E},\mathcal{I}),\ \mathcal{B}\rangle \implies \mathcal{VC} \quad \langle \mathcal{S}'\ ;\ \mathcal{S}'',\ \mathcal{I},\ \mathcal{C} \wedge \texttt{subst}(\neg\mathcal{E},\mathcal{I}),\ \mathcal{B}\rangle \implies \mathcal{VC}'}{\langle (\texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}\ \texttt{else}\ \mathcal{S}')\ ;\ \mathcal{S}'',\ \mathcal{I},\ \mathcal{C},\ \mathcal{B}\rangle \implies \mathcal{VC} \cup \mathcal{VC}'}$$

$$\textsc{SeqScope}\ \frac{x_1',\ldots,x_n' \text{ are fresh} \quad \langle \mathcal{S},\ \mathcal{I}[x_1 \mapsto x_1',\ldots,x_n \mapsto x_n'],\ \mathcal{C},\ (\{x_1,\ldots,x_n\},\mathcal{S}')\cdot\mathcal{B}\rangle \implies \mathcal{VC}}{\langle (\texttt{vars}\ x_1,\ldots,x_n\ \texttt{in}\ \mathcal{S})\ ;\ \mathcal{S}',\ \mathcal{I},\ \mathcal{C},\ \mathcal{B}\rangle \implies \mathcal{VC}}$$

$$\textsc{EndScope}\ \frac{\langle \mathcal{S},\ \mathcal{I} \setminus \nu,\ \mathcal{C},\ \mathcal{B}\rangle \implies \mathcal{VC}}{\langle \cdot,\ \mathcal{I},\ \mathcal{C},\ (\nu,\mathcal{S})\cdot\mathcal{B}\rangle \implies \mathcal{VC}} \qquad \textsc{EscapeScope}\ \frac{i = j+1 \quad \langle \texttt{escape}\ j\ ;\ \mathcal{S}',\ \mathcal{I} \setminus \nu,\ \mathcal{C},\ \mathcal{B}\rangle \implies \mathcal{VC}}{\langle \texttt{escape}\ i\ ;\ \mathcal{S},\ \mathcal{I},\ \mathcal{C},\ (\nu,\mathcal{S}')\cdot\mathcal{B}\rangle \implies \mathcal{VC}}$$

$$\textsc{EscapeNoOp}\ \frac{\langle \mathcal{S},\mathcal{I},\mathcal{C},\mathcal{B}\rangle \implies \mathcal{VC}}{\langle \texttt{escape}\ 0\ ;\ \mathcal{S},\ \mathcal{I},\ \mathcal{C},\ \mathcal{B}\rangle \implies \mathcal{VC}} \qquad \textsc{Empty}\ \frac{}{\langle \cdot,\ \mathcal{I},\ \mathcal{C},\ \varnothing\rangle \implies \varnothing}$$

(c) VCG for sequential compositions

Figure 4 Fragment of VCG for B3

Theorem 2 that depends on coinduction. Thus, the part of our proof that uses coinduction is Theorem 3. This is the strength of our 2-layered approach—it concentrates all coinductive reasoning into Theorem 3, making the rest of our proofs stay within the simpler, and better automatable, structural definitions.

3.2 VCG for statements

Intuitively, VCG for a procedure with body $\mathcal{S}$, precondition *req*, and postcondition *ens* is equivalent to the VCG of a B3 statement that combines *req*, *ens*, and $\mathcal{S}$ into one statement. We make this idea precise in Fig. 4a, which defines the VCG for a procedure in terms of a VCG for statements. The statement-level VCG is denoted $\langle \mathcal{S},\mathcal{I},\mathcal{C},\mathcal{B}\rangle \implies \mathcal{VC}$, where in addition to the statement $\mathcal{S}$ and set of VCs $\mathcal{VC}$, there are three components: The map of variable incarnations $\mathcal{I}$ stores a representation for each variable. The context $\mathcal{C}$ is a formula, over the variable incarnations, that is assumed to hold at the start of $\mathcal{S}$. The sequence of continuations $\mathcal{B}$ contains the statements that follow $\mathcal{S}$, one for each enclosing scope.

We use the notation $\langle \mathcal{A},\mathcal{I},\mathcal{C}\rangle \implies \langle \mathcal{VC},\mathcal{I}',\mathcal{C}'\rangle$ to say that, for an atomic statement $\mathcal{A}$,

incarnation map $\mathcal{I}$, and context $\mathcal{C}$, VCG outputs a set of VCs $\mathcal{VC}$ along with an updated incarnation map $\mathcal{I}'$ and context $\mathcal{C}'$. Fig. 4b shows the details for the atomic statements `check`, `assume`, and assignment. Rule Check generates a VC that checks $\mathcal{E}$ to hold in the given context and rule Assume adds expression $\mathcal{E}$ to the context. In both cases, the program expression $\mathcal{E}$ is converted into a logical formula by applying the substitutions stored in $\mathcal{I}$. Rule Assign shows how the incarnation map is updated: each time a procedure updates a variable x, VCG creates a new fresh name x' (known as an *incarnation* of the variable), updates the context to equate x' with the new value of x, and updates the incarnation map to remember that x is henceforth represented by x'. Similarly to the substitution used in the Call rule (Fig. 3a), the formula $\texttt{subst}(\mathcal{E}, \mathcal{I})$ replaces all the variables in $\mathcal{E}$ by their current incarnations according to $\mathcal{I}$.

Next, let's consider sequential composition where the left-hand side is an atomic statement or an `if-then-else` statement, see Fig. 4c. For an atomic statement on the left, rule Seq straightforwardly composes the updated $\mathcal{I}'$ and $\mathcal{C}'$ with the VCG of $\mathcal{S}'$, outputting a union of the two VC sets. Rule SeqIf generates VCs for $(\texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}\ \texttt{else}\ \mathcal{S}')\ ;\ \mathcal{S}''$ by combining the VCs of $\mathcal{S}\ ;\ \mathcal{S}''$ and $\mathcal{S}'\ ;\ \mathcal{S}''$ under the context assuming the guard condition $\mathcal{E}$ and its negation, respectively. This kind of VC generation is sometimes referred to as *symbolic execution* [13], which in practice gives more stable performance for automated logic engines, at the expense of a larger number of VCs (*cf.* [8]).

VCG for control flow statements like `vars` and `escape` is more complex and explains the need for the block continuation $\mathcal{B}$. Analogously to the SeqIf rule, we would want to define VCG for $(\texttt{vars}\ x_1, \ldots, x_n\ \texttt{in}\ \mathcal{S})\ ;\ \mathcal{S}'$ in terms of VCG for the sequence $\mathcal{S}\ ;\ \mathcal{S}'$. However, we must separate the two, since $\mathcal{S}$ may contain `escape` statements that direct control flow to $\mathcal{S}'$. So, in rule SeqScope (Fig. 4c), we record $\mathcal{S}'$ in the block continuation sequence $\mathcal{B}$. With each element of that sequence, we also record the set of variables introduced in the `vars` statement, which helps us reduce bloat in the incarnation maps when scopes are exited. Rule EndScope exits a scope after its last statement and rule EscapeScope exits a scope abruptly. In more detail, those rules remove the variables introduced in the current scope and then continue to the next enclosing scope. Base cases for `escape` and for the outermost statement sequence of a procedure body are handled by EscapeNoOp and Empty, respectively.

3.3 Soundness of VCG for statements

Recall, the soundness of our VCG w.r.t. the structural semantics is stated in Theorem 2. That lemma needs a way to go from a particular form of the VCG judgment $\langle \mathcal{S}, \mathcal{I}, \mathcal{C}, \mathcal{B} \rangle \implies \mathcal{VC}$ to a particular form of the structural-semantics judgment $\langle \mathcal{S}, st \rangle \Downarrow \overline{Q}$. To that end, we prove a more general property, stated by the following theorem:

▶ **Theorem 4** (Soundness of VCG for statements). *For any statement $\mathcal{S}$, incarnation map $\mathcal{I}$, context $\mathcal{C}$ over the incarnations, block continuation sequence $\mathcal{B}$, set of VCs $\mathcal{VC}$, and incarnation-variable state ist (that is, a mapping from incarnation variables to values), if*

- *free variables in $\mathcal{S}$ and $\mathcal{B}$ are in the domain of $\mathcal{I}$,*
- *jump depth of $\mathcal{S}$ is less than or equal to the length of $\mathcal{B}$, and for each continuation $\mathcal{S}'$ in $\mathcal{B}$, the jump depth of $\mathcal{S}'$ is less than or equal to the length of its suffix in $\mathcal{B}$,*
- *$\langle \mathcal{S}, \mathcal{I}, \mathcal{C}, \mathcal{B} \rangle \implies \mathcal{VC}$ holds,*
- *for each $vc \in \mathcal{VC}$, vc is valid, and*
- *ist satisfies the context formula $\mathcal{C}$,*

then $\langle \texttt{mkScope}(\mathcal{S}, \mathcal{B}),\ \texttt{AdjustState}(ist, \mathcal{I}) \rangle \Downarrow \langle \top \rangle$

The theorem uses two functions. Function $\texttt{mkScope}(\mathcal{S}, \mathcal{B})$ reconstructs a scoped statement from any $\mathcal{S}$ and $\mathcal{B}$ as follows:

$$\texttt{mkScope}(\mathcal{S}, \mathcal{B}) \stackrel{\text{def}}{=} \begin{cases} \texttt{mkScope}(\mathcal{B}',\ (\texttt{vars}\ \nu\ \texttt{in}\ \mathcal{S})\ ;\ \mathcal{S}') & \text{if } \mathcal{B} = (\nu, \mathcal{S}') \cdot \mathcal{B}' \\ \mathcal{S} & \text{if } \mathcal{B} = \varnothing \end{cases} \tag{2}$$

Function $\texttt{AdjustState}(ist, \mathcal{I})$ projects an incarnation-variable state (which may contain values of incarnations that stand for previous values of program variables) to a state where the program variables map to the values of the incarnations according to $\mathcal{I}$. In symbols, we have $\mathsf{Dom}(\texttt{AdjustState}(ist, \mathcal{I})) = \mathsf{Dom}(\mathcal{I})$ and for any variable x, $\texttt{AdjustState}(ist, \mathcal{I})(x) = ist(\mathcal{I}(x))$

Now, let us explain the statement of the theorem in more detail. The first two conditions of Theorem 4 are technical and ensure that the VCG is well-defined: VCG turns each variable use in $\mathcal{S}$ and $\mathcal{B}$ into an incarnation, so all of those variables must be accounted for by $\mathcal{I}$. The restriction on jump depth ensures that every `escape` statement in $\mathcal{S}$ and in the continuations of $\mathcal{B}$ has a target continuation in $\mathcal{B}$. The third condition says that VCG has generated the set of VCs $\mathcal{VC}$, and the fourth that all those VCs are logically valid. The fifth condition says that the values of the incarnation variables in ist satisfy the context condition $\mathcal{C}$.

Assuming the five conditions, Theorem 4 concludes that the semantics of the scoped statement $\texttt{mkScope}(\mathcal{S}, \mathcal{B})$ is over-approximated by the set $\top$ of all states. This means two things: (1) all jumps in $\texttt{mkScope}(\mathcal{S}, \mathcal{B})$ stay within this statement, and (2) all the checks inside $\texttt{mkScope}(\mathcal{S}, \mathcal{B})$ are sound—if there was a failing check, according to rule Check in Fig. 2a, the semantics of the overall statement would not relate to any set of final states.

Note that the conclusion of the theorem does not speak about the incarnation-variable state ist directly, but rather about the program state $\texttt{AdjustState}(ist, \mathcal{I})$, which is the state adjusted according to the incarnations map $\mathcal{I}$. This is because $\mathcal{C}$ is defined not over the program variables, but over the incarnations of these variables. Hence, state ist contains values for all incarnations of variables. With $\texttt{AdjustState}(ist, \mathcal{I})$, we select only the values for the latest ones, corresponding to the current state of the program.

The proof of Theorem 4 is done by induction on the structure of the statement $\mathcal{S}$ and the block continuation $\mathcal{B}$. We will not elaborate on the proof here, but refer the reader to the accompanying Dafny implementation.

Using Theorem 4, we can prove Theorem 2 (which we used in the proof of the main soundness result, Theorem 1) as follows:

Proof of Theorem 2. Using rule Proc in Fig. 4a, we run VCG on the empty context $\mathcal{C}$, the empty block-continuation sequence $\mathcal{B}$, and the initial incarnations map $\texttt{initial}(x_1, \ldots, x_n)$. From this, Theorem 4 gives us

$$\langle \texttt{assume}\ req;\ \mathcal{S};\ \texttt{check}\ ens,\ \texttt{AdjustState}(ist,\ \texttt{initial}(x_1, \ldots, x_n)) \rangle \Downarrow \langle \top \rangle \tag{3}$$

for any incarnation-variable state ist (since any such state satisfies the empty context $\mathcal{C}$).

Note that here we have applied the second equation from (2) to reduce the scoped statement with the empty block continuation to just $\mathcal{S}$. Also, note that

$$\texttt{AdjustState}(ist, \texttt{initial}(x_1, \ldots, x_n))$$

turns the arbitrary incarnation-variable state ist into an arbitrary state on the program variables $x_1, \ldots, x_n$. Now, remember that for a procedure $\mathcal{P}$ with body $\mathcal{S}$, precondition req, and postcondition ens, $\mathsf{MeetsSpec}(\mathcal{P})$ says (rule Procedure in Fig. 3b) that for any state st defined on variables $x_1, \ldots, x_n$, if st satisfies the precondition req, then the semantics $\mathcal{S}$ is over-approximated by ens. Using the rules for the `assume` and `check` statements, we ensure that that is exactly what (3) guarantees. ◀

```
ghost predicate Sem(s: Stmt, st: State,
                    posts: seq<iset<State>>)
  requires |posts| ≠ 0
{
  match s
  case Check(e)          ⇒
    ∧ e.HoldsOn(st)
    ∧ st ∈ posts[0]
  case Assume(e)         ⇒
    e.HoldsOn(st)
    ⟹ st ∈ posts[0]
  case Loop(_, body)     ⇒
    ∃ inv: iset<State> ·
      ∧ st ∈ inv
      ∧ ∀ st': State ·
        st' ∈ inv ⟹
        Sem(body, st', posts[0 := inv])
  case Call(proc, args) ⇒
      ∧ (∀ e ∈ proc.Pre ·
           e.Subst(args).HoldsOn(st))
      ∧ ∀ st': State ·
          ∧ st'.Keys = st.Keys
          ∧ (∀ x ∈ st.Keys − args.Keys ·
               st[x] = st'[x])
          ∧ (∀ e ∈ proc.Post ·
               e.Subst(args).HoldsOn(st'))
          ⟹ st' ∈ posts[0]
  ...
}
```

(a) Structural semantics

```
0 greatest predicate RefSem(s: Stmt, st: State,
1                           posts: seq<iset<State>>)
2   requires |posts| ≠ 0
3 {
4   match s
5   case Check(e)          ⇒
6     ∧ e.RefHoldsOn(st)
7     ∧ st ∈ posts[0]
8   case Assume(e)         ⇒
9     e.RefHoldsOn(st)
10    ⟹ st ∈ posts[0]
11  case Loop(inv, body)   ⇒
12    RefSem(Seq(
13        body,
14        Loop(inv, body)),
15      st, posts)
16  case Call(proc, args) ⇒
17    RefMeetsSpec(proc) ∧
18      ∧ (∀ e ∈ proc.Pre ·
19           e.Subst(args).RefHoldsOn(st))
20      ∧ ∀ st': State ·
21          ∧ st'.Keys = st.Keys
22          ∧ (∀ x ∈ st.Keys − args.Keys ·
23               st[x] = st'[x])
24          ∧ (∀ e ∈ proc.Post ·
25               e.Subst(args).RefHoldsOn(st'))
26          ⟹ st' ∈ posts[0]
27  ...
28 }
```

(b) Reference semantics

Figure 5 Definition of B3 semantics in Dafny

4 Implementation

In this section, we show a representative portion of our definitions of the B3 semantics and VCG in Dafny. Then, we comment on the main parts of the proofs that connect the structural semantics $\Downarrow$ and the reference semantics $\overset{\text{ref}}{\Downarrow}$ as well as on the soundness of VCG for statements.

4.1 Dafny definition of B3 semantics

To define the structural and reference semantics of B3 in Dafny, we use recursive and coinductive predicates, respectively. Fig. 5 shows the Dafny definitions for fragments of the two semantics. Both predicates take a statement, state, and sequence of sets of states as arguments. We use `posts` to name the latter variable in the Dafny code, opposed to $\overline{Q}$ in math formlas. This way, `Sem(s, st, posts)` corresponds to the math semantics judgment $\langle s, st\rangle \Downarrow \overline{Q}$, and `RefSem(s, st, posts)` corresponds to $\langle s, st\rangle \overset{\text{ref}}{\Downarrow} \overline{Q}$. The latter is declared coinductive via Dafny's `greatest` keyword (line 0 at Fig. 5b).

Both predicates proceed by pattern-matching on the statement s, with the body of each matching `case` being a boolean expression that encodes the premises of the corresponding rule from Sec. 2. We illustrate this encoding on the cases for `check` and `assume` in the `Sem` predicate. Here, the post-state sequence $\overline{Q}$ becomes the `posts` argument, $\overline{Q}_0$ becomes `posts[0]`, and the math's $[\![\mathcal{E}]\!]_{st} = \mathsf{true}$ becomes the boolean predicate `e.HoldsOn(st)`. With these mappings, CHECK becomes the conjunction `e.HoldsOn(st) ∧ st in posts[0]`, and ASSUME becomes the implication `e.HoldsOn(st) ⟹ st in posts[0]`.

The cases for `check` and `assume` in `RefSem` (Fig. 5b) are identical to those above, except that `HoldsOn` is replaced by `RefHoldsOn`. This difference does not matter for the paper, but it does in our accompanying Dafny implementation, where we also support recursive interpreted functions. Supporting them runs into the same issue as with recursive procedures, because the function body is typically structurally larger than the function call itself. To overcome this issue, we use the same trick as we did with procedures, *i.e.*, we define a separate

predicate for the coinductive reference semantics of expressions (`RefHoldsOn`). For brevity, we omit the implementation of the expression semantics here.

The interesting parts of Fig. 5b are the cases for the `loop` and `call` statements: for both, the structural and reference semantics differ, as discussed in Sec. 2.

For `loop`, the reference case (lines 11–16) is a direct transcription of LoopUnroll (Fig. 2c): `RefSem` on `Loop(inv, body)` is defined to hold whenever `RefSem` holds on the unrolled body `Seq(body, Loop(inv, body))` (with the same `st` and `posts`). The structural case is more subtle. It realizes the math rule LoopInv (Fig. 2d). The invariant *Inv*, which appears *unbound* in the math premise, corresponds directly to the existentially quantified ∃ `inv: iset<State>` in the Dafny case. Both encode the obligation that *some* invariant exists, leaving the choice of witness to whichever client invokes the rule (in our case, the VCG). Next, in the rule's conclusion $\langle \texttt{loop}\ \mathcal{S}, st\rangle \Downarrow Q' \cdot \overline{Q}$, the entire post-state sequence is represented in the Dafny code by the single `posts` argument: Q' corresponds to `posts[0]` and $\overline{Q}$ to the tail `posts[1..]`. The body's premise $\langle \mathcal{S}, st'\rangle \Downarrow \mathit{Inv} \cdot \overline{Q}$ differs from the conclusion's sequence only at the head, replacing Q' with *Inv*. In Dafny, this is written `posts[0 :=inv]`, which updates position 0 of `posts` with `inv` and leaves the rest unchanged. Note also that the syntactic invariant annotation on the AST, written `Loop(_, body)` in the case pattern, is ignored by `Sem`: the structural semantics commits to no particular invariant, leaving the supplied annotation for the VCG to validate.

For `call`, the reference semantics contains one more conjuct, `RefMeetsSpec(proc)` (line 17 at Fig. 5b), compared to the structural one. The `RefMeetsSpec(proc)` predicate is a straightforward Dafny encoding of $\mathsf{MeetsSpec}_{\mathrm{ref}}(\mathcal{P})$ from Fig. 3b. Since `RefMeetsSpec` is mutually recursive with `RefSem` (because of `call` statements), Dafny requires us to mark it as a `greatest` predicate, too.

We omit the implementation of predicates `Sem` and `RefSem` for other statements and instead focus on the proof of Theorem 3, next.

4.2 Connecting B3 semantics layers

To connect the structural and reference semantics, we prove that the latter is a refinement of the former (Theorem 3 in Sec. 3.1). This proof relies on two lemmas: the refinement of semantics for statements (Fig. 6a) and for procedures (Fig. 6b). As discussed in Sec. 4.1, the definitions of `RefSem` and `RefMeetsSpec` are coinductive and mutually recursive. Hence, the refinement lemmas must also utilize coinductive reasoning (obtained in Dafny using a `greatest lemma` declaration) and mutual recursion.

For basic statements such as `check` and `assume`, the proof of `SemRefine` (Fig. 6a) is straightforward and follows from a similar lemma regarding the refinement of expression semantics, which we omit here.

For the `loop` statement, the proof utilizes the loop unrolling lemma `SemLoopUnroll`, which demonstrates that the invariant-based structural semantics of a `loop` admits unrolling. In more detail, the proof of `SemRefine` calls `SemLoopUnroll` (line 12 in Fig. 6a) to obtain `Sem(Seq([body, Loop(inv, body)]), st, posts)`. By Dafny's automatically applied coinductive hypothesis [16, 17], this implies `RefSem(Seq([body, Loop(inv, body)]), st, posts)`, which is the unfolded equivalent of `RefSem(Loop(inv, body), st, posts)`, our proof goal.

Finally, for the `call` statement, the proof invokes the `MeetsSpecRefine` lemma (line 14 in Fig. 6a). In turn, if the given procedure `proc` has an implementation, `MeetsSpecRefine` invokes `SemRefine` on the procedure body for all states satisfying the precondition.

```
greatest lemma SemRefine(s: Stmt, st: State,
    posts: seq<iset<State>>, procs: set<Procedure>)
  requires Sem(s, st, posts)
  requires s.ProceduresCalled() ⊆ procs
  requires procs.IsClosed()
  requires ∀ p ∈ procs · MeetsSpec(p)
  ensures RefSem(s, st, posts)
{
  match s
  case Check(e) ⇒ ...
  case Assume(e) ⇒ ...
  case Loop(inv, body) ⇒
    SemLoopUnroll(inv, body, st, posts);
  case Call(proc, args) ⇒
    MeetsSpecRefine(proc, procs);
    ...
}
```

(a) Semantics soundness proof for statements. Notice that the coinductive lemmas `SemRefine` and `MeetsSpecRefine` (Fig. 6b) are mutually recursive and, like the corresponding definitions in Fig. 3, may not be terminating calls.

```
greatest lemma MeetsSpecRefine(proc: Procedure,
    procs: set<Procedure>)
  requires MeetsSpec(proc)
  requires proc ∈ procs
  requires proc.ProceduresCalled() ⊆ procs
  requires procs.IsClosed()
  requires ∀ p ∈ procs · MeetsSpec(p)
  ensures RefMeetsSpec(proc)
{
  if proc.Body.Some? {
    forall st ∈ proc.PreSet
      ensures RefSem(proc.Body.value, st, [proc.PostSet])
    {
      SemRefine(proc.Body.value, st, [proc.PostSet], procs);
    }
  }
}
```

(b) Semantics soundness proof for procedures

```
lemma SemLoopUnroll(inv: Expr, body: Stmt,
    st: State, posts: seq<iset<State>>)
  requires Sem(Loop(inv, body), st, posts)
  ensures Sem(Seq([body, Loop(inv, body)]), st, posts)
{
  ...
}
```

(c) Loop unrolling lemma, in terms of the structural semantics (`Sem`)

```
lemma ProcsSoundRefine(procs: set<Procedure>)
  requires procs.IsClosed()
  requires ∀ p ∈ procs · MeetsSpec(p)
  ensures ∀ p ∈ procs · RefMeetsSpec(p)
{
  forall proc ∈ procs { MeetsSpecRefine(proc, procs); }
}
```

(d) Procedures soundness refinement lemma

Figure 6 Soundness proofs for B3 semantics

4.3 Soundness of VCG

In Dafny, we define our VCG for procedures and statements as methods that mimic the $\langle \mathcal{S}, \mathcal{I}, \mathcal{C}, \mathcal{B} \rangle \implies \mathcal{VC}$ judgment in Fig. 4. Like the corresponding rules, the recursive Dafny methods are proved to terminate via size measures for $\mathcal{S}$ and $\mathcal{B}$. To obtain a proof of Theorem 1 and Theorem 4, we define the VCG methods with pre- and postconditions (`requires` and `ensures` clauses in Dafny) and amend the implementation with ghost proof code.

4.4 Mechanization

Our model of B3 — the formal AST, semantics, and VCG together with their soundness proofs — comprises about 4.7 kLoC of Dafny: roughly 2.0 kLoC for the formal AST and state, 660 LoC for the omnisemantics, and a further ∼2.0 kLoC for the formal VCG and its soundness proofs. Around 1.5 kLoC of total lines are actual code, including the B3 AST, semantics, and VCG. The other 3 kLoC are code annotations and proofs. The whole effort was completed in 4 month by a person with no previous experience in Dafny proofs.

Across the proofs, Dafny's SMT-based automation was very helpful in closing lemmas and subgoals. Manual guidance was concentrated in three places. First, a cluster of incarnation- and substitution-bookkeeping lemmas in the VCG soundness chain accounts for most of the `calc` chains in the formalization, since the chain of substitutions through the variable-incarnation map is too detailed for SMT to discharge directly. Second, reasoning about jumps through block continuations: continuations are sequences, and the VCG repeatedly folds over them with sequence-aggregating operations (for instance, to compute the set of variables to remove from the scope when an `escape` unwinds several frames at once); these fold-like recursions need explicit inductive lemmas to be discharged. Third, the soundness of

the expression semantics: B3 expressions include quantifiers and uninterpreted functions, so the resulting semantics is effectively higher-order, and SMT cannot deal with it efficiently.

5 Related Work

Verified verification condition generators

A prominent approach to building trustworthy verification tools is the foundational verification of the Verification Condition Generator (VCG) itself. This involves proving the VCG's soundness within a proof assistant and is typically expressed as a property that if the generated VCs are valid, then the program is safe with respect to its operational semantics.

Early pioneering work by Homeier and Martin presented a mechanically verified VCG in the HOL system for the Sunrise language [12]. While foundational, this work targets a very limited language, focusing on a basic "While" syntax with simple assignments and structured control flow. Crucially, it lacks support for non-determinism and modern verification features such as uninterpreted types or non-local control flow, which are essential for modeling low-level systems and abstract specifications.

More recently, Nezamabadi *et al.* developed a verified VCG for a realistic subset of the Dafny language [22] along with a verified compiler from Dafny to the CakeML [14]. Their work represents a significant leap forward by formalizing a feature-rich language in HOL4 and proving the correctness of its VCG and compiler in tandem. However, despite its sophistication, their approach maintains a deterministic operational semantics.

While both of these efforts provide strong correctness guarantees, they do not yet cover several features critical to the expressivity of modern verification languages. Specifically, neither framework supports non-determinism at the semantics level. Furthermore, unlike our work, they do not support non-local control flow or uninterpreted types and functions, which are standard in production-grade verifiers like Dafny to allow for modular and underspecified reasoning.

Formalization of the Core Why3 logic in Coq [4] includes more advanced features such as support for inductive types in the semantics. However, their approach differs fundamentally from ours in their treatment of termination: their semantics is total, which requires encoding termination checks and restricting recursion to structural recursion. Our semantics is partial, thanks to coinduction, which frees us from termination obligations and allows more general recursive definitions. Finally, [4] covers only two key Why3 transformations, leaving the rest for the future work.

Another aspect of verifier verification is verifying the translation from a target language to an IVL. Dardinier *et al.* [6] formalize the operational semantics of a core IVL and formally connect it to two different back-end verifiers, proving the soundness of front-end translations from separation logic to the IVL. Their work is mechanized in Isabelle/HOL and instantiated with elements of the Viper verification infrastructure. These two approaches are complementary and could together form a fully verified pipeline: our work verifies the IVL back-end (from IVL to verification conditions), while theirs verifies the front-end translation (from the target language to the IVL).

Validation of verifiers

In contrast to foundational verification, which proves the VCG correct for all possible inputs, an alternative approach is Result Certification. Instead of verifying the entire tool—which can be prohibitively complex for industrial verifiers—this method generates a formal

certificate for each specific run of the VCG. This certificate is then validated by a small, trusted kernel or a proof assistant.

A pioneer example is the framework by Parthasarathy *et al.* for certifying the core transformations of the Boogie verifier [24]. By formalizing a subset of the Boogie intermediate language in Isabelle/HOL, they provide a means to validate that specific passes, such as loop elimination and passification, are semantics-preserving.

While Result Certification is more flexible and easier to adapt to evolving tools like Boogie or Dafny, it introduces a constant computational overhead for every verification task and requires a "translation bridge" to the proof assistant for every certificate generated. Furthermore, if a certificate fails to validate due to a bug in the unverified VCG, the user is left without a formal guarantee for that specific program. Our work follows the foundational approach: by proving the VCG sound once and for all against a coinductive semantics, we eliminate the need for per-run certificates and provide a unified guarantee that covers the entire language's expressive power, including non-deterministic and recursive behaviors.

Semantic support for advanced language features

Another key novelty of our work lies in our formal semantics, which supports a language featuring arbitrary forward jumps, uninterpreted types, and both interpreted and uninterpreted functions.

Vogels *et al.* [25] present Featherweight VeriFast, a formally defined and mechanically verified (in Coq) core subset of the VeriFast program verifier. While their work provides a soundness proof for a verifier, the target language is deliberately kept small and does not support features such as non-local control flow, uninterpreted types, or higher-order functions.

Another related line of work is the CompCert project [20] for the Clight language. CompCert provides a comprehensive and landmark semantics for a realistic language (Clight). However, its design goals differ significantly from ours. CompCert targets a realistic executable language intended for compilation to machine code. Consequently, it does not include features essential for an IVL, such as code annotations, uninterpreted types and (un)interpreted functions, which are used to abstract away implementation details during the verification process.

Furthermore, CompCert's handling of control flow is tailored toward standard C constructs (such as `break` and `continue` in loops), rather than our arbitrary forward jumps.

6 Conclusion and Future Work

In summary, we have formalized the semantics of the intermediate verification language B3. We did so in two layers.

One layer, which we called the *structural semantics*, defines the semantics of each statement in terms of the semantics of the statement's structurally smaller sub-statements. This gives a simple mathematical definition in terms of well-founded recursive predicates. It also has the advantage that the definition stays closer to how one generates logical verification conditions to check the correctness of a given program. The drawback of this semantics definition is that it does not "connect all the dots". In particular, the semantics describes arbitrary loop iterations using loop invariants, which it asserts exists, but it does not show any signs of repeating the loop body. Also, for calls, it uses a pre/post specification, but does not directly check the callee's implementation to satisfy the specification.

The other layer, which we called the *reference semantics*, is defined coinductively. This gives a natural way to define a loop in terms of the indefinite unrolling of its body and it can hold a callee responsible for meeting its procedure specification. Such an infinitely tangled definition still makes sense mathematically as a least or greatest fix-point. As we elaborated on in Sec. 3.1, we have chosen the greatest fix-point, that is, we use a coinductive definition.

Our interest in studying the semantics of B3 is to prove its VC generation to be correct. We have done so in this paper (and the accompanying Dafny implementation) by defining a VC generator (VCG), proving that the VCG respects the structural semantics, and proving that the structural semantics implies the reference semantics. We had a good experience with the approach of splitting the semantics into two layers, because most of the proofs are done w.r.t. the structural semantics, for which Dafny provides good automation. Moreover, the proofs that connect the two semantics were not laborious (about 100 LOC).

The semantic definitions make the meaning of the B3 language precise and our soundness proofs give us high confidence in the correctness of the B3 VC generator. However, there is more work to be done. We mention the two most conspicuous opportunities for future work:

One is that, although the VC generation we defined in this paper (and that is found in our accompanying Dafny implementation) stays close to the VCG that is implemented in the public B3 tool, the public tool sends its VCs to an SMT solver in an incremental fashion. For example, the context $\mathcal{C}$ in rule SeqIf (Fig. 4c) is pushed to the SMT solver and reused for each of the branches in the premise of the rule. Moreover, the actual B3 VCG has evolved and we would like to integrate our formalization and proofs into the B3 release.

The other conspicuous opportunity for future work is that, so far, we have ignored B3's `axiom` definitions. These user-defined conditions serve as a backdrop for each of the VCs to be checked. In principle, this seems like a simple addition to our theorems and proofs—just conjoin the axioms as a big antecedent of both the semantics and the VCs. However, the actual operation of the public B3 tool is to supply the SMT solver with these definitions (and also the definitions for interpreted functions) lazily. We hope to address this in future work.

In conclusion, we have shown the feasibility of using an auto-active verifier like Dafny for programming-language meta-theory in a practically important setting. The semantics, theorems, proofs, and implementation are are given in the same programming language. Unlike many meta-theory proofs in the literature, the proofs are declarative, not procedural [11]. The layering of the semantics into a structural semantics and a reference semantics provided a good trade-off that we recommend.

6.1 AI Declaration

For this project, we have only used Cursor [1] auto-completion.

References

1 Anysphere. Cursor: The AI code editor. https://cursor.com, 2026. Accessed: 2026-05-04.

2 Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005. doi:10.1007/11804192_17.

3 Arthur Charguéraud, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. Omnisemantics: Smooth handling of nondeterminism. *ACM Trans. Program. Lang. Syst.*, 45(1), March 2023. doi:10.1145/3579834.

4 Joshua M. Cohen and Philip Johnson-Freyd. A formalization of core Why3 in Coq. volume 8, New York, NY, USA, January 2024. Association for Computing Machinery. doi:10.1145/3632902.

5 Flaviu Cristian. Correct and robust programs. *IEEE Trans. Software Eng.*, 10(2):163–174, 1984. doi:10.1109/TSE.1984.5010218.

6 Thibault Dardinier, Michael Sammler, Gaurav Parthasarathy, Alexander J. Summers, and Peter Müller. Formal foundations for translational separation logic verifiers. *Proc. ACM Program. Lang.*, 9(POPL), January 2025. doi:10.1145/3704856.

7 Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

8 Marco Eilers, Malte Schwerhoff, and Peter Müller. Verification algorithms for automated separation logic verifiers. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification — 36th International Conference, CAV 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 362–386. Springer, 2024. doi:10.1007/978-3-031-65627-9_18.

9 Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — Where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

10 Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In Chris Hankin and Dave Schmidt, editors, *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 193–205. Association for Computing Machinery, 2001. doi:10.1145/360204.360220.

11 John Harrison, Josef Urban, and Freek Wiedijk. History of interactive theorem proving. In Jörg H. Siekmann, editor, *Computational Logic*, volume 9 of *Handbook of the History of Logic*, pages 135–214. Elsevier, 2014. doi:10.1016/B978-0-444-51624-4.50004-6.

12 P. V. Homeier and D. F. Martin. A mechanically verified verification condition generator. *The Computer Journal*, 38(2):131–141, 01 1995. arXiv:https://academic.oup.com/comjnl/article-pdf/38/2/131/1051248/380131.pdf, doi:10.1093/comjnl/38.2.131.

13 James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

14 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 179–191. Association for Computing Machinery, 2014. doi:10.1145/2535838.2535841.

15 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning — 16th International Conference, LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, April 2010. doi:10.1007/978-3-642-17511-4_20.

16 K. Rustan M. Leino. Automating induction with an SMT solver. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation — 13th International Conference, VMCAI 2012*, volume 7148 of *Lecture Notes in Computer Science*, pages 315–331. Springer, January 2012.

17 K. Rustan M. Leino. Automating theorem proving with SMT. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving — 4th International Conference, ITP 2013*, volume 7998 of *Lecture Notes in Computer Science*, pages 2–16. Springer, July 2013.

18 K. Rustan M. Leino. Well-founded functions and extreme predicates in Dafny: A tutorial. In Boris Konev, Stephan Schulz, and Laurent Simon, editors, *IWIL-2015. 11th International Workshop on the Implementation of Logics*, volume 40 of *EPiC Series in Computing*, pages 52–66. EasyChair, 2016. URL: /publications/paper/vHsB, doi:10.29007/v2m3.

19 K. Rustan M. Leino and Michał Moskal. Co-induction simply — automatic co-inductive proofs in a program verifier. In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods — 19th International Symposium*, volume 8442 of *Lecture Notes in Computer Science*, pages 382–398. Springer, May 2014.

20 Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. doi:10.1145/1538788.1538814.

21 Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation — 17th International Conference, VMCAI 2016*, volume 9583 of *Lecture Notes in Computer Science*, pages 41–62. Springer, 2016.

22 Daniel Nezamabadi, Magnus O. Myreen, and Yong Kiam Tan. Verified VCG and verified compiler for Dafny. In *CPP 2026*, 2026. URL: https://api.semanticscholar.org/CorpusID:283672227.

23 David Lorge Parnas. A technique for software module specification with examples. *Commun. ACM*, 15(5):330–336, 1972. doi:10.1145/355602.361309.

24 Gaurav Parthasarathy, Peter Müller, and Alexander J. Summers. Formally validating a practical verification condition generator. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Proceedings, Part II*, pages 704–727, Berlin, Heidelberg, 2021. Springer-Verlag. doi:10.1007/978-3-030-81688-9_33.

25 Frédéric Vogels, Bart Jacobs, and Frank Piessens. Featherweight VeriFast. *Logical Methods in Computer Science*, 11(3), 2015. doi:10.2168/LMCS-11(3:19)2015.