Formally Validating a Practical Verification Condition Generator

No Author Given

No Institute Given

Abstract. A program verifier produces reliable results only if both the *logic* used to justify the program's correctness is sound, and the *implementation* of the program verifier is itself correct. Whereas it is common to formally prove soundness of the logic, the implementation of a verifier typically remains unverified. Bugs in verifier implementations may compromise the trustworthiness of successful verification results. Since program verifiers used in practice are complex, evolving software systems, it is generally not feasible to formally verify their implementation.

In this paper, we present an alternative approach: we *validate successful runs* of the widely-used Boogie verifier by producing a *certificate* which proves correctness of the obtained verification result. Boogie performs a complex series of program translations before ultimately generating a verification condition whose validity should imply the correctness of the input program. We show how to certify three of Boogie's core transformation phases: the elimination of cyclic control flow paths, the (SSA-like) replacement of assignments by assumptions using fresh variables (passification), and the final generation of verification conditions. Similar translations are employed by other verifiers. Our implementation produces certificates in Isabelle, based on a novel formalisation of the Boogie language.

1 Introduction

Program verifiers are tools which attempt to prove the correctness of an implementation with respect to its specification. A successful verification attempt is, however, only meaningful if both the *logic* used to justify the program's correctness is sound, and the *implementation* of the program verifier is itself correct. It is common to formally prove soundness of the logic, but the implementations of program verifiers typically remain unverified. As is standard for complex software systems, bugs in verifier implementations can and do arise, potentially raising doubts as to the trustworthiness of successful verification results.

One way to close this gap is to prove a verifier's implementation correct. However, such a *once-and-for-all* approach faces serious challenges. Verifying an existing implementation bottom-up is not practically feasible because such implementations tend to be large and complex (for instance, the Boogie verifier [27] consists of over 30K lines of imperative C# code), use a variety of libraries, and are typically written in efficient mainstream programming languages which themselves lack a formalisation. Alternatively, one could develop a verifier that is correct by construction. However, this approach requires the verifier to be (re-)implemented in an interactive theorem prover (ITP) such as Coq [13] or Isabelle [23]. This precludes the free choice of implementation language and paradigm, exploitation of concurrency, and possibility of tight integration with standard compilers and IDEs, which is often desirable for program verifiers [3, 4, 12, 24]. Both verification approaches substantially impede software maintenance, which is problematic since verifiers are often rapidly-evolving software projects (for instance, the Boogie repository [1] contains more than 5000 commits). It is, thus, not surprising that once-and-for-all verification of program verifiers has been restricted to idealised implementations, omitting for instance challenging optimisations [2, 40]; the gap between those implementations and the tools used in practice remains significant.

To address these challenges, in this work we employ a different approach. Instead of verifying the implementation once and for all, we *validate specific runs* of the verifier by producing a *certificate* which proves the correctness of the obtained verification result. Our certificate generation formally relates the input and output of the verifier, but does so largely independently of its implementation, which can freely employ complex languages, algorithms, or optimisations. Our certificates are formal proofs in Isabelle, and so checkable by an independent trusted tool; their guarantees for a certified run of the verifier are as strong as those provided by a (hypothetical) verified verifier.

We apply our novel verifier validation approach to the widely-used Boogie verifier, which verifies programs written in the intermediate verification language Boogie. The Boogie verifier is a *verification condition generator*: it verifies programs by generating a verification condition (VC), whose validity is then discharged by an SMT solver. Certifying a verifier run requires proving that validity of the VC implies the correctness of the input program. Certification of the validity-checking of the VC is an orthogonal concern; our results can be combined with work in that area [10, 14, 18] to obtain end-to-end guarantees.

Like many automatic verifiers, Boogie is a *translational verifier*: it performs a sequence of substantial Boogie-to-Boogie translations (*phases*), simplifying the task and output of the final efficient VC computation [5,17]. The key challenges in certifying runs of the Boogie tool are to certify each of these phases, including final VC generation. In particular, we present novel techniques for making the following three key phases (and many smaller ones) of Boogie's tool chain certifying:

1. The elimination of loops (more precisely, cycles in the CFG) by reducing the correctness of loops to checking loop invariants *(CFG-to-DAG phase)*
2. The replacement of assignments by (SSA-style) introduction of fresh variables and suitable `assume` statements *(passification phase)*
3. The final generation of the VC, which includes the erasure and logical encoding of Boogie's polymorphic type system [31] *(VC phase).*

The certification of such verifier phases is related to existing work on compiler verification [32] and validation [7, 37, 38]. However, the translations and the certified property we tackle here are fundamentally different from those in compilers. Compilers typically require that each execution of the target program corresponds to an execution of the source program. In contrast, the encoding of a program in a translational verifier typically has intentionally more executions (for instance, allows more non-determinism). Moreover, translational verifiers need to handle features not present in standard programming languages such as **assume** statements and background theories. Prior work on validating such verifier phases has been limited in the supported language and extent of the formal guarantee; we discuss comparisons in detail in Sec. 8.

Contributions. Our paper makes the following technical contributions.

1. The first formal semantics for a significant subset of Boogie (including axioms, polymorphism, type constructors), mechanised in Isabelle.
2. A validation technique for two core program-to-program translations occurring in verifiers (CFG-to-DAG and passification).
3. A validation technique for the VC phase, handling polymorphism erasure and Boogie's type system encoding [29], for which no prior formal proof exists.
4. A version of the Boogie implementation that produces certificates for a significant subset of Boogie.

Making the Boogie verifier certifying is an important result, reducing the trusted code base for a wide variety of verification tools implemented via encodings into Boogie, e.g. Dafny [29], VCC [12], Corral [26], and Viper [33]. Moreover, the technical approach we present here can in future be applied to the certification of the translations performed by these tools, and those based on comparable intermediate verification languages such as Frama-C [24] and Krakatoa [16] based on Why3 [15] and Prusti [3] and VerCors [9] based on Viper [33].

Outline. Sec. 2 explains at a high-level, how our validation approach is structured for the different phases. Sec. 3 introduces a formal semantics for Boogie. Secs. 4, 5 and 6 present our validation of the passification, CFG-to-DAG, and VC phases, respectively. Sec. 7 evaluates our certificate-producing version of Boogie. Sec. 8 discusses related work Sec. 9 concludes.

2 Approach

A Boogie program consists of a set of procedures, each with a specification and a procedure body in the form of a (reducible) control-flow-graph (CFG), whose blocks contain basic commands; we present the formal details in the next section. Boogie verifies each procedure modularly, desugaring procedure calls according to their specifications. Verification is implemented via a series of phases: program-to-program translations and a final computation of a VC to be checked by an SMT solver. Our goal is to formally certify (per run of Boogie) that validity of this VC implies the correctness of the original procedure.

To keep the complexity of certificates manageable, our technical approach is *modular* in three dimensions: decomposing our formal goal per *procedure* in the

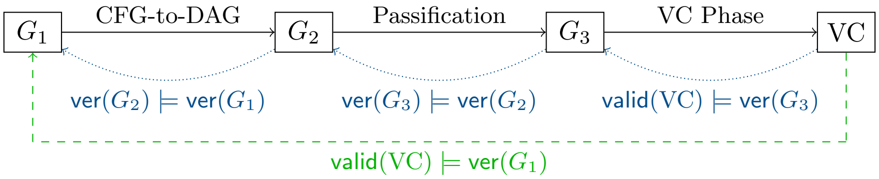

G_1
CFG-to-DAG
G_2
Passification
G_3
VC Phase
VC
$\mathsf{ver}(G_2) \models \mathsf{ver}(G_1)$
$\mathsf{ver}(G_3) \models \mathsf{ver}(G_2)$
$\mathsf{valid}(\mathrm{VC}) \models \mathsf{ver}(G_3)$
$\mathsf{valid}(\mathrm{VC}) \models \mathsf{ver}(G_1)$

Fig. 1. Key phases of verification in Boogie and their certification. The solid edges show Boogie's transformations on a procedure body; each node G_i represents a control-flow-graph. Our final certificate (green edge) is constructed by formally linking the three phase certificates represented by the blue edges. Each of three phase certificates additionally incorporates extra smaller transformation phases that we do not show here.

Boogie program, per *phase* of the Boogie verification, and per *block* in the CFG of each procedure. This modularity makes the full automation of our certification proofs in Isabelle practical. In the following, we give a high-level overview of this modular structure; the details are presented in subsequent sections.

Procedure decomposition. Boogie has no notion of a main program or an overall program execution. A Boogie program is correct if each of its procedures is individually correct (that is, the procedure body has no failing traces, as we make precise in the next section). Boogie computes a separate VC for each procedure, and we correspondingly validate the verification of each procedure separately.

Phase decomposition. We break our overall validation efforts down into per-phase sub-problems. In this paper, we focus on the following three most substantial and technically-challenging of these sequential phases, illustrated in Fig. 1. (1) The *CFG-to-DAG phase* translates a (possibly-cyclic) CFG to an acyclic CFG (*cf.* Sec. 4). This phase substantially alters the CFG structure, cutting loops using annotated loop invariants to over-approximate their executions. (2) The *passification phase* eliminates imperative updates by transforming the code into static single assignment (SSA) form and then replacing assignments with *constraints* on variable versions (*cf.* Sec. 5). Both of these phases introduce extra non-determinism and **`assume`** statements (which, if implemented incorrectly could make verification unsound by masking errors in the program). (3) The final *VC phase* translates the acyclic, passified CFG to a verification condition that, in addition to capturing the weakest precondition, encodes away Boogie's polymorphic type system [31].

We construct certificates for each of these key phases separately (depicted by the blue lines in Fig. 1). For each phase, we certify that *if* the target of the translation phase is correct (a correct Boogie program for the first two phases; a valid VC for VC phase) then the source (program) of the phase is correct. This modular approach lets us focus the proof strategy for each phase on its conceptually-relevant concerns, and provides robustness against *changes* to the verifier since at most the certification of the changed phases may need adjustment. Logically, our per-phase certificates are finally glued together to guarantee the analogous end-to-end property for the entire pipeline, depicted in green in Fig. 1.

The certificates of the key phases also incorporate various smaller transformations between the key phases such as peephole optimisation. Our work covers these small transformations, but we focus on the key phases in the paper. Boogie also performs several smaller translation steps *prior* to the CFG-to-DAG phase. These include transforming ASTs to corresponding CFGs, optimisations such as dead variable elimination, and desugaring procedure calls using their specifications (via explicit **assert**, **assume**, and **havoc** statements). Our approach applies analogously to these initial smaller phases, but our current implementation certifies only the pipeline of all phases from the input to the CFG-to-DAG phase onwards.

CFG decomposition. When tackling the certification of *each* phase, we further break down validation of a procedure's CFG in a two-tiered manner:

1. *Local block lemmas:* We prove independent lemmas per CFG block, relating the executions through a block in the source program with the corresponding block(s) in the target program. In particular, these lemmas imply that if the target block(s) have no failing executions (or the VC generated for that block holds, for the VC phase), neither did the source block.
2. *Global block theorems:* We show analogous per-block results concerning all executions *from this block onwards*; we build these compositionally by reverse-topological traversal of either the source or target CFGs, as appropriate.

This decomposition over program structure separates command-level reasoning (local block lemmas) from CFG-level reasoning (global block theorems). It enables concise lemmas and proofs in Isabelle and makes each comprehensible to a human.

3 A Formal Semantics for Boogie

Our certificates prove that the validity of a VC generated by Boogie formally implies correctness of the Boogie program to be verified. This proof relies crucially on a formal semantics for Boogie itself. Our first contribution is the first such formal semantics for a significant subset of Boogie, mechanised in Isabelle. Our semantics uses the Boogie reference manual [27], the presentation of its type system [31], and the Boogie implementation for reference; none of those provide a formal account of the language. For space reasons, we explain only the key concepts of our detailed formalisation here; we will make the full Isabelle mechanisation available as part of our accompanying artifact.

3.1 The Boogie Language

Boogie programs consist of a set of top-level declarations of global variables and constants (the *global data*), axioms, uninterpreted (polymorphic) functions, type constructors, and procedures. A procedure declaration includes parameter, local-variable, and result-variable declarations (the *local data*), a pre- and postcondition, and a procedure body given as a CFG[1]. CFGs are formalised as usual in terms of basic blocks (containing a possibly-empty list of *basic commands*), and edges; semantically, execution after a basic block continues via any of its successors non-deterministically.

The types, expressions, and basic commands in our Boogie subset are shown in Fig. 5 in App. A. We support the primitive types *Int* and *Bool*; types obtained via declared type constructors are *uninterpreted types*; the sets of values such types denote are constrained only via Boogie axioms and **assume** commands.

Boogie expression syntax is largely standard (e.g. including typical arithmetic and boolean operations). Old-expressions **old**(e) evaluate the expression e w.r.t. the current local data and the global data as it *was* in the pre-state of the procedure execution. Boogie expressions also include universal and existential *value* quantification, as well as universal and existential *type* quantification.

Basic commands form the single-steps of traces through a Boogie CFG; sequential composition is implicit in the lists of basic commands in a CFG basic block and further control flow (including loops) is prescribed by CFG edges. Boogie's basic commands are assumes, asserts, assignments, and havocs; **havoc** x non-deterministically assigns a value matching the type of variable x to x.

The main Boogie features *not* supported by our subset are maps and other primitive types such as bitvectors. Boogie maps are polymorphic and impredicative, i.e. one can define maps that contain themselves in their domain. Giving a semantic model for such maps in a proof assistant such as Isabelle or Coq is non-trivial; we aim to tackle this issue in the future. Modelling bitvectors will be simpler, although maintaining full automation may require some additional work.

3.2 Operational Semantics and Type Safety

Values and state model. Our formalisation embeds integer and boolean values shallowly as their Isabelle counterparts; an Isabelle carrier type for all *abstract values* (those of uninterpreted types) is a parameter of our formalisation. Each uninterpreted type is (indirectly) associated with a *non-empty* subset of abstract values via a surjective *type interpretation* map $\mathcal{T}$ from abstract values to (single) types; particular interpretations of uninterpreted types can be obtained via different choices of type interpretation $\mathcal{T}$.

One can understand Boogie programs in terms of the set of possible *traces* through each procedure body. Traces are (as usual) composed of sequences of steps according to the semantics of basic commands and paths through the CFG; these can be finite or infinite (representing a non-terminating execution). A trace may halt in three different cases: (1) the exit block of the procedure is reached in a state satisfying the procedure's postcondition (a *complete* trace)[2], (2) an

[1] Source-level procedure specifications also include *modifies clauses*, declaring a set of global variables the procedure may modify. As we tackle Boogie programs after procedure calls have been desugared, there are no modifies clauses in our formalisation.

[2] The case of the postcondition *not* holding is subsumed under point (2), since Boogie checks postconditions by generating extra **assert** statements.

`assert` `A;` command is reached in a state not satisfying assertion `A` (a *failing* trace), or (3) an **`assume`** `A;` command is reached in a state not satisfying `A` (a trace which *goes to magic*). Our formalisation correspondingly includes three kinds of Boogie program states: a distinguished *failure state* F, a distinguished *magic state* M, and *normal states* $\mathsf{N}((os, gs, ls))$. A normal state is a triple of partial mappings from variables to values for the old global state (for the evaluation of old-expressions), the (current) global state, and the local state, respectively.

Expression evaluation. An expression e evaluates to value v if the judgement $\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle e, \mathsf{N}(ns)\rangle \Downarrow v$ holds in the context $(\mathcal{T}, \Lambda, \Gamma, \Omega)$. Here, $\mathcal{T}$ is a *type interpretation* (as above), Λ is a *variable context*: a pair (G, L) of type declarations for the global (G) and local (L) data. Γ is a *function interpretation*, which maps each function name to a semantic function mapping a list of types and a list of values to a return value. The type substitution Ω maps type variables to types.

The rules defining this judgement can be found in App. A.2. For example, the following rule expresses when a universal type quantification evaluates to **`true`**:

$$\frac{\forall \tau.\ \mathit{closed}(\tau) \Longrightarrow \mathcal{T}, \Lambda, \Gamma, \Omega(t \mapsto \tau) \vdash \langle e, ns\rangle \Downarrow \mathtt{\mathbf{true}}}{\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle \forall_{ty}\, t.\ e, ns\rangle \Downarrow \mathtt{\mathbf{true}}}$$

The premise requires one to show that the expression e reduces to **`true`** for every possible type τ that is *closed* (i.e. does not contain any type variables). In general, expression evaluation is possible only for well-typed expressions; we also formalise Boogie's type system and (for the first time) prove its type safety.

Command and CFG reduction. The judgement $\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle c, s\rangle \rightarrow s'$ defines when a command c reduces in state s to state s'; the rules are in App. A.3. This reduction is lifted to lists of commands cs to model the semantics of a single trace through a CFG block (the judgement $\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle cs, s\rangle\ [\rightarrow]\ s'$). The operational semantics of CFGs is modelled by the judgement $\mathcal{T}, \Lambda, \Gamma, \Omega, G \vdash \delta \rightarrow_{\mathsf{CFG}} \delta'$, expressing that the CFG configuration δ reduces to configuration δ' in the CFG G. A CFG configuration is either *active* or *final*. An active configuration is given by a tuple $(\mathsf{inl}(b_n), s)$, where b_n is the block identifier indicating the current position of the execution and s is the current state. A final configuration consists of a tuple $(\mathsf{inr}(()), s)$ for state s (and unit value $()$) and is reached at the end of a block that has either no successors, or is in a magic or failure state.

3.3 Correctness

A procedure is *correct* if it has *no failing traces*. This is a *partial correctness* semantics; a procedure body whose traces never leave a loop is trivially correct provided that no intermediate **`assert`** commands fail. Procedure correctness relies on CFG correctness. A CFG G is correct w.r.t. a postcondition Q and a context $(\mathcal{T}, \Lambda, \Gamma, \Omega)$ in an initial normal state $\mathsf{N}(ns)$ if the following holds for all configurations (r, s'):

$$\mathcal{T}, \Lambda, \Gamma, \Omega, G \vdash (\mathsf{inl}(\mathsf{entry}(G)), \mathsf{N}(ns)) \rightarrow^{*}_{\mathsf{CFG}} (r, s') \Longrightarrow$$
$$[s' \neq \mathsf{F} \land (r = \mathsf{inr}(()) \Rightarrow \forall ns'.s' = \mathsf{N}(ns') \Longrightarrow \mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle Q, \mathsf{N}(ns')\rangle \Downarrow \mathtt{\mathbf{true}})]$$

```
assume i != 0
j := 0
while(i != 0)
inv j >= 0 ∧ (i = 0 ⇒ j > 0)
{
  if(i < 5) {
    j := j+1
  }
  i := i-1
}
assert j > 0
```

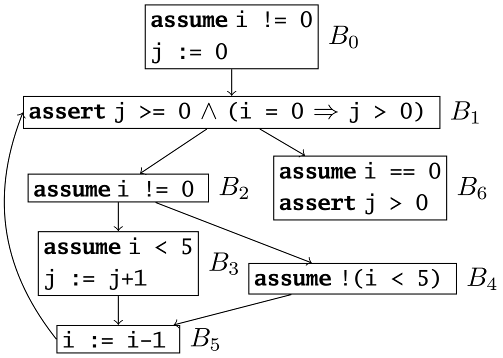


Fig. 2. Running example in source code and CFG representation, respectively.

where entry(G) is the entry block of G and $\rightarrow^*_{\mathsf{CFG}}$ is the reflexive-transitive closure of the CFG reduction. The postcondition is needed only if a final configuration is reached in a normal state, while failing states must be unreachable. Whenever we omit Q, we implicitly mean the postcondition to be simply **true**. In our tool, we consider only empty initial mappings Ω, since we do not support procedure type parameters (lifting our work to this feature will be straightforward).

For a procedure p to be correct w.r.t. a particular context, its body CFG must be correct w.r.t. the same context and p's postcondition, *for all* initial normal states $\mathsf{N}(ns)$ that satisfy p's precondition and which respect the context. For ns to *respect* a context, it must be well-typed and must satisfy the axioms when restricted to its constants. We say that p is *correct*, if it is correct w.r.t. *all well-formed contexts*, which, among other things, must have a well-typed function interpretation (see App. A.4).

Running example. We will use the simple CFG of Fig. 2 as a running example, intended as body of a procedure with trivial (**true**) pre- and post-conditions. The code includes a simple loop with a declared loop invariant, which functions as a classical Floyd/Hoare-style inductive invariant, and for the moment can be considered as an implicit **assert** statement at the loop head. The CFG has infinite traces: those which start from any state in which `i` is negative. Traces starting from a state in which `i` is zero go to magic; they do not reach the loop. The program is correct (has no failing traces): all other initial states will result in traces that satisfy the loop invariant and the **assert** statement. If we removed the initial **assume** statement, however, there *would* be failing traces: the loop invariant check would fail if `i` were initially zero.

4 The CFG-to-DAG Phase

In this section, we present the validation for the CFG-to-DAG phase in the Boogie verifier. This phase is challenging as it changes the CFG structure, inserts

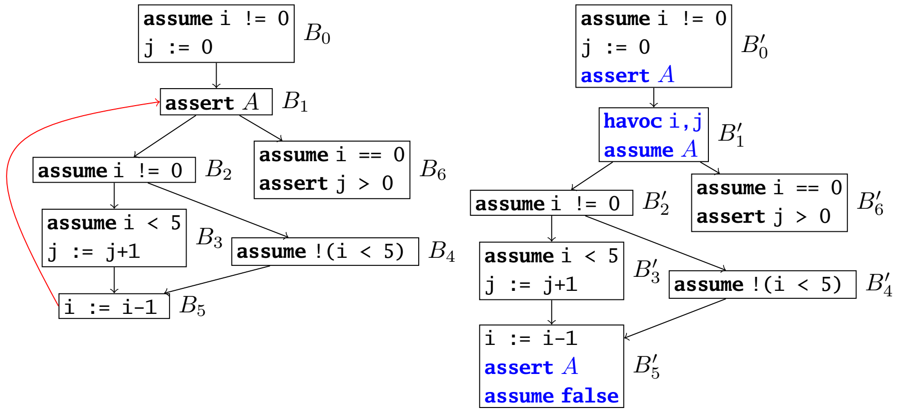


Fig. 3. The CFG-to-DAG phase applied to the running example (source is left, target is right). The back-edge (the red edge in the left CFG) is eliminated. The blue commands are new. A is given by `j >= 0` $\wedge$ `(i = 0` $\Rightarrow$ `j > 0)`.

additional non-deterministic assignments and **assume** statements, and must do so correctly for arbitrary (reducible) nested loop structures.

4.1 CFG-to-DAG Phase Overview

The CFG-to-DAG phase applies to every *loop head* block identified by Boogie's implementation and any *back-edges* from a block reachable from the loop head block back to the loop head (following standard definitions for reducible CFGs [20]). Fig. 3 illustrates the phase's effect on our simple running example. Block B_1 is the (only) loop head here, and the edge from B_5 to it is the only back-edge in this program (completing looping paths via B_2 and B_3 or B_2 and B_4). An **assert** A statement starting a loop head (like B_1) is interpreted as declaring A to be the loop invariant[3]. The CFG-to-DAG phase performs the following steps:

1. Accumulate a set X_H of all (local and global) variables *assigned-to* on *any looping path* from the loop head back to itself. In our example, X_H is $\{i, j\}$.
2. Move the **assert** A statement declaring a loop invariant (if any) from the loop head to the end of *each preceding* block (in our example: B_0 and B_5).
3. Insert **havoc** statements at the start of the loop head block per variable in X_H, followed by a single **assume** A statement (preceding any further statements).
4. For each block with a back-edge to the loop head, delete the back-edge; if this leaves the block with no successors, append **assume false** to its commands.

The havoc-then-assume sequence introduced in step 3 can be understood as generating traces for *arbitrary values of* X_H satisfying the loop invariant A,

[3] In general, multiple assertions at the beginning of a loop head form the invariant. We focus on a single assertion here for simplicity.

effectively over-approximating the set of states reachable at the loop head in the original program. In particular, the remnants of any originally looping path (e.g. B'_1,B'_2,B'_3,B'_5) enforce that any non-failing trace starting from any such state must (due to the **assert** added to block B'_5 in step 2) result in a state which re-establishes the loop invariant. Such paths exist only to enforce this inductive step (analogously to the premise of a Hoare logic while rule); so long as the **assert** succeeds, we can discard these traces via step 4.

While we illustrate this step on a simple CFG, in general a loop head may have multiple back-edges, looping structures may nest, and edges may exit multiple loops. For the above translation to be correct, the CFG must be reducible and loop heads and corresponding back-edges identified accurately, which is complex in general. Importantly (but perhaps surprisingly), our work makes this phase of Boogie certifying *without* explicitly verifying (or even defining) these notions.

4.2 CFG-to-DAG Certification: Local Block Lemmas

We define first our local block lemmas for this phase. Recall that these prove that if executing the statements of a target block yields no failing executions, the same holds for the corresponding source block; this result is trivial for source blocks other than loop heads and their immediate predecessors, since these are unchanged in this phase. To enable eventual composition of our block lemmas, we need to also reflect the role of the **assume** and **assert** statements employed in this phase. The formal statement of our local block lemmas is as follows[4]:

Theorem 1 (CFG-to-DAG Local Block Lemma). *Let cs_S and cs_T be corresponding source and target blocks, respectively, for the CFG-to-DAG transformation. If cs_S is a loop head, let X_H be as defined in CFG-to-DAG step 1 (and empty otherwise) and let A_{pre} be its loop invariant (or* **true** *otherwise). If cs_S is a* predecessor *of a loop head, let A_{post} be the loop invariant of its successor (and* **true** *otherwise). Then,* if*:*

1. $\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle cs_S, \mathsf{N}(ns_1)\rangle\ [\rightarrow]\ s'_1$
2. $\forall s'_2.\ \mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle cs_T, \mathsf{N}(ns_2)\rangle\ [\rightarrow]\ s'_2 \Longrightarrow s'_2 \neq \mathsf{F}$
3. *A_{pre} is satisfied in ns_1, and ns_2 differs from ns_1 only on variables in X_H and variables not defined in Λ*

then*: $s'_1 \neq \mathsf{F}$ and if s'_1 is a normal state, then (1) A_{post} is satisfied in s'_1, and (2) there is a target execution in cs_T from $\mathsf{N}(ns_2)$ that reaches a normal state that does not differ from s'_1 on any variables other than those not defined in Λ.*

The gist of this lemma is to capture *locally* the ideas behind the four steps of the phase. For example, consequence (1) reflects that *after* the transformation, any blocks that *were previously* predecessors of a loop head (B'_0 and B'_5 in our running example) will have an `assert` statement checking for the corresponding invariant (and so if the target program has no failing traces, in each trace this invariant will be true at that point).

[4] We omit some details regarding well-typedness, handled fully in our formalisation.

4.3 CFG-to-DAG Certification: Global Block Theorems

We lift our certification to *all* traces through the source and target CFGs; the statement of the corresponding global block theorems is similar to that of local block theorems lifted to CFG executions, and for space reasons we do not present it here, but it is included in our Isabelle formalisation. In particular, we prove for each block (working in reverse topological order through the target CFG blocks) that if executions starting in the target CFG block never fail, neither do any executions starting from the corresponding source CFG block, and looping paths modify at most the variables havoced according to step 3 of the phase.

The major challenge in these proofs is reasoning about looping paths in the source CFG, since these revisit blocks. To solve this challenge, we perform inductive arguments per loop head in terms of the number of steps remaining in the trace in question[5]. Our global block theorem for a block B then carries as an assumption an induction hypothesis for each loop that contains B. Proving a global block theorem for the origin of a back-edge is taken care of by applying the corresponding induction hypothesis.

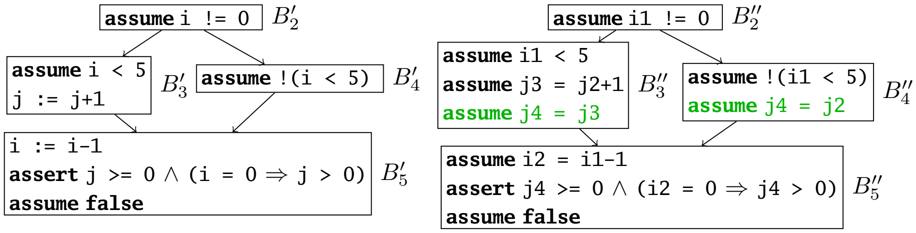
This proof strategy works only if we have obtained the induction hypothesis for the loop head *before* we use the global block theorem of the origin of a back-edge (otherwise we cannot discharge the block theorem's hypothesis). In other words, our proof implicitly shows the necessary requirement that loop heads (as identified by Boogie) dominate all back-edges reaching them *without us formalising any notion of domination, CFG reducibility, or any other advanced graph-theoretic concept*. This shows a major benefit of our validation approach over a once-and-for-all verification of Boogie itself: our proofs indirectly check that the identification of loop heads and back-edges guarantees the necessary *semantic properties* without being concerned with *how* Boogie's implementation computes this information.

Our approach applies equally to nested loops and more-generally to reducible CFG structures; *all* corresponding induction hypotheses are carried through from the visited loop-heads. The requirement that no more than the havoced variables X_H are modified in the source program is easily handled by showing that variables modified in an inner loop are a subset of those in outer loops. As for all of our results, our global block lemmas are proven automatically in Isabelle per Boogie procedure, providing per-run certificates for this phase.

5 The Passification Phase

In this section, we describe the validation of the passification phase in the Boogie verifier. Unlike the previous phase, passification makes no changes to the CFG structure, but makes substantial changes to the program states (via SSA-like renamings), substantially increases non-determinism, and employs **`assume`** statements to re-tame the sets of possible traces.

[5] This may seem insufficient since traces can be infinite, but importantly a *failing* trace is always finite, and our theorems need only eliminate the chance of failing traces.



```
assume i != 0

assume i < 5
j := j+1

assume !(i < 5)

i := i-1
assert j >= 0 ∧ (i = 0 ⇒ j > 0)
assume false
```

```
assume i1 != 0

assume i1 < 5
assume j3 = j2+1
assume j4 = j3

assume !(i1 < 5)
assume j4 = j2

assume i2 = i1-1
assert j4 >= 0 ∧ (i2 = 0 ⇒ j4 > 0)
assume false
```

Fig. 4. The passification phase applied to the branch in the running example with the result on the right. The green commands are the synchronisation commands. At the uppermost blocks shown here, the current versions of `i` and `j` are `i1` and `j2`, respectively. The full CFGs are shown in Fig. 13 and Fig. 14 in App. B.

5.1 Passification Phase Overview

The main goal of passification is to eliminate assignments such that a more efficient VC can be ultimately generated [5, 17, 28]. In the Boogie verifier, this is implemented as a single transformation phase that can be thought of as two independent steps. Firstly, the source CFG is transformed into *static single assignment* (SSA) form, introducing *versions* (fresh variables) for each original program variable such that each version is assigned at most once in any program trace. In a second step, variable assignments are *completely eliminated*: each assignment command $x := e$ is replaced by **assume** $x = e$. Havoc statements are simply removed; their effect is implicit in the fact that a new variable version is used (via the SSA step) *after* such a statement.

Fig. 4 shows the effect of this phase on four blocks of our running example (the full figure of the target CFG is shown in Fig. 14 in App. B). The commands highlighted in green are inserted just before a join in the CFG structure to introduce a consistent variable version (here, `j4`) for use in the subsequent block. It is convenient to speak of target variables in terms of their source program counterparts: we say e.g. that `j` *has version* 4 on entry to block B'_5.

Compared to traces through the source program, the space of variable values in a trace through the target program is initially much larger; each version may, on entry to the CFG, have an arbitrary value. For example, `j4` may have any value on entry to B''_2; traces in which its value does not correspond to the constraint of the **assume** statements in B''_3 or B''_4 will go to magic and not reach B''_5. Importantly, however, not *all* traces go to magic; enough are preserved to simulate the executions of the original program: each **assume** statement constrains the value of exactly one variable version, and the same version is never constrained more than once. Capturing this delicate argument formally is the main challenge in certifying this step.

As extra parts of the passification phase, the Boogie verifier performs constant propagation and desugars old-expressions (using variable versions appropriate to the entry point of the CFG). We omit their descriptions here for brevity, but our implementation certifies them.

5.2 Passification Certification: Local Block Lemmas

To validate the passification phase, it is sufficient to show that each source execution is simulated by a corresponding target execution, made precise by constructing a relation between the states in these executions. Such *forward simulation* arguments are standard for proving correctness of compilers for deterministic languages. However, the situation here is more complex due to the fact that the target CFG has a much wider space of traces: the values of each versioned variable in the target program are initially unconstrained, meaning traces exist for all of their combinations. On the other hand, many of these traces do not survive the **`assume`** statements encountered in the target program. Picking the correct *single* trace or state to simulate a particular source execution would require knowledge of all variable assignments that are *going* to happen, which is not possible due to non-determinism and would preclude the block-modular proof strategies that our validation approach employs.

Instead, we generalise this idea to relating each single source state s with a *set* T of corresponding target program states. We define variable relations $\mathcal{V}_R$ at each point in a trace, making explicit the mappings used in the SSA step between source program variables and their corresponding versions. For example, on entry to block B'_2 in the source version of our running example (correspondingly B''_2 in the target), the $\mathcal{V}_R$ relation relates `i` to `i1` and `j` to `j2`. All states $t \in T$ must precisely agree with s w.r.t. $\mathcal{V}_R$ (e.g., $s(\texttt{i}) = t(\texttt{i1})$, $s(\texttt{j}) = t(\texttt{j2})$). On the other hand, our sets of states T are defined to be completely unconstrained (besides typing) for *future* variable versions. For example, for every $t \in T$ at the same point in our example, there will be states in T assigning each possible value (of the same type) to `i2` (and otherwise agreeing with t).

More precisely, for a set of variables X, we say that a set of states T *constrains at most* X if, for every $t \in T$, $z \notin X$, and value v of z's type, we have $t[z \mapsto v] \in T$. In other words, the set T is closed under arbitrary changes to values of all variables *not* in X. We construct our sets T such that they constrain at most *current and past versions* of program variables. It is this fact that enables us to handle subsequent **`assume`** statements in the target program and, in particular, to show that the set of possible traces in the target program never becomes empty while there are possible traces in the source program. For example, when relating the source command `j := j+1` in B'_3 with the target command **`assume`** `j3 = j2 + 1` in block B''_3, we use the fact that our set of states does not constrain `j3` to prove that, although many traces go to magic at this point, for a non-empty set of states $T' \subseteq T$ (those in which `j3` has the "right" value equal to `j2 + 1`), execution continues in the target.

We now make these notions more precise by showing the definition of our local block lemmas for the passification phase.

Theorem 2 (Passification Local Block Lemma). *Let B be a source block with commands cs, whose corresponding target block has commands cs'; let $\mathcal{V}_R$ and $\mathcal{V}'_R$ be the variable relations at the beginning and end of B, respectively. Let X be a set of variable versions, and $\mathsf{N}(ns)$ be a normal state. Let T be a non-empty set of normal states such that $\mathsf{N}(ns)$ agrees with T according to $\mathcal{V}_R$, and T*

constrains at most X. Furthermore, let Y be the variable versions corresponding to the targets of assignment and havoc statements in cs. If both

1. $A, \Lambda_1, \Gamma, \Omega \vdash \langle cs, \mathsf{N}(ns)\rangle \; [\rightarrow] \; s'$
2. $X \cap Y = \emptyset$

then there exists a non-empty set of normal states $T' \subseteq T$ s.t. T' depends only on $X \uplus Y$ and for each normal state $t' \in T'$, there exists a state t'^{*} s.t.

1. $A, \Lambda_2, \Gamma, \Omega \vdash \langle cs_2, t'\rangle \; [\rightarrow] \; t'^{*} \wedge (s' = \mathsf{F} \Longrightarrow t'^{*} = \mathsf{F})$
2. *If s' is a normal state, then s' and t' are related w.r.t. $\mathcal{V}_R'$ (and $t'^{*} = t'$).*

This lemma captures our generalised notion of forward simulation appropriately. The first conclusion expresses that the target does not get stuck and that failures are preserved, while the second shows that if execution neither fails nor stops then the resulting states are related. Note that premise 2 is essential in the proof to guarantee that the **assume** statements introduced by passification do not eliminate the chance to simulate source executions; the condition expresses that the variable versions newly constrained do not intersect with those previously constrained. To prove these lemmas over the commands in a single block, we are forced to check that the same version is not constrained twice.

5.3 Passification Certification: Global Block Theorems

As for all phases, we lift our local block lemmas to theorems certifying all executions *starting* from a particular block, and thus, ultimately, to entire CFGs. For the passification phase, most of the conceptual challenges are analogous to those of the local block lemmas; we similarly employ $\mathcal{V}_R$ relations between source variables and their corresponding target versions. To connect with our local block lemmas (and build up our global block theorems, which we do backwards through the CFG structure), we repeatedly require the key property that the set of variable versions constrained in our executions so far is disjoint from those which may be constrained by a subsequent **assume** statement (*cf.* premise 2 of our local block lemma above). Concretely tracking and checking disjointness of these concrete sets of variables is simple, but turns out to get expensive in Isabelle when the sets are large.

We circumvent this issue with our own *global versioning scheme* (as opposed to the versions used by Boogie, which are *independent* for different source variables): according to the CFG structure, we assign a *global* version number $\mathsf{ver}_{\mathcal{G}}(x)$ to each variable x in the target program such that, if x is constrained in a target block B' and y is constrained in another target block B'' reachable from B', then $\mathsf{ver}_{\mathcal{G}}(x) < \mathsf{ver}_{\mathcal{G}}(y)$. Such a consistent global versioning always exists in the target programs generated by Boogie because the only variables not constrained exactly once *in the program* are those used to synchronise executions (i.e., `j4` in Fig. 4), which always appear right before branches are merged. We can now encode our disjointness properties (which imply this fact) much more cheaply: we simply compare the *maximal* global version of all already-constrained variables with the *minimal* global version of those (potentially) to be constrained. Since we represent variables as integers in the mechanisation, we directly use our global version *as* the variable name for the target program; there is no need for an extra lookup table. Note that (readability aside) it makes no difference which variables names are used in intermediate CFGs; we ultimately care only about validating the original CFG.

6 The VC Phase

In this section, we present the validation of the VC phase in the Boogie verifier. This phase has two main aspects: (1) it encodes and desugars all aspects of the Boogie type system, employing additional uninterpreted functions and axioms to express its properties [31]; program expression elements such as Boogie functions are analogously desugared in terms of these additional uninterpreted functions, creating a non-trivial logical gap between expressions as represented in the VC and those from the input program. (2) It performs an efficient (block-by-block) calculation of a weakest precondition for the (acyclic, passified) CFG, resulting in a formula characterising its verification requirements, subject to background axioms and other hypotheses.

6.1 VC Structure

The generated VC has the following overall structure (represented as a shallow embedding in our certificates)[6]:

$$\forall \ \underbrace{\mathit{VC\ quantifiers}}_{\substack{\text{type encoding parameters,}\\ \text{functions, variable values}}} \ . \ \ (\ \underbrace{\mathit{VC\ assumptions}}_{\substack{\text{type encoding,}\\ \text{func./var./prog. axioms}}} \implies \mathit{CFG\ WP})$$

The VC quantifies over parameters required for the type encoding, as well as VC counterparts representing the variable values and functions in the Boogie program. The VC is an implication: the premise contains assumptions that axiomatise the functions generated in this phase to desugar the type system, including axioms expressing the typing of variables and functions, as well as assumptions directly relating to axioms explicitly declared in the Boogie program. The conclusion of the implication is an optimised version of the weakest (liberal) precondition (WP) of the CFG[7].

[6] Note that top-level quantification over functions is implicit in the (first-order) SMT problem generated by Boogie; we quantify explicitly in our Isabelle representation.

[7] One difference in our version of the Boogie verifier is that we switched off the generation of extra variables introduced to report error traces [30]; these are redundant for programs that do not fail and further complicate the VC structure.

6.2 Boogie's Logical Encoding of the Boogie Type System

We first briefly explain Boogie's logical encoding of its own type system. Values and types are represented at the VC level by two uninterpreted carrier sorts V and T. An uninterpreted function *typ* from V to T maps each value to the representation of its type. Boogie type constructors are each modelled with an (injective) uninterpreted function C with return sort T and taking arguments (per type constructor parameter) of sort T. For example, a type constructor $\mathit{List}(t)$ is represented by a VC function from T to T. Inverse projection functions are also generated (C_i^{-1} for each type argument at position i), e.g. mapping the representation of a type $\mathit{List}(t)$ to the representation of type t.

This encoding is then used throughout the Boogie program to map all typed Boogie expressions to untyped VC expressions with types as explicit values. This can have a non-trivial effect on the corresponding program elements. For example, a polymorphic Boogie function declared as: `function foo<t>(x:List t): t` would, in our semantics for Boogie, be a partial function f of type $ty_{closed} \to val \rightharpoonup val$, where $f(\tau, v)$ is defined only if v has type $\mathit{List}(\tau)$. By contrast, the corresponding VC-level function h_{vc} is *total* of type $val \to val$; it does not take a type as input (even though this type defines the return type). This modelling suffices because *after this desugaring*, the type parameter is technically redundant: one can recover the return type from the argument value: $\mathit{List}_1^{-1}(\mathit{typ}(v))$.

6.3 Working from VC Validity

Our certificates assume that the generated VC is valid (recall that certifying the validity-checking of the VC by an SMT solver is an orthogonal concern). However, connecting VC validity back to block (and command)-level properties about the specific program requires a number of technical steps.

Firstly, we need to construct Isabelle-level semantic values (e.g. functions) to *instantiate* the top-level quantifiers (e.g. over functions) in the VC. We instantiate the carrier sort for values with the corresponding type *val* denoting Boogie values in our formalisation; the carrier sort for *types* is instantiated to be all *closed* Boogie types ty_{closed}. Constructing explicit models for the functions used to model Boogie's type system (satisfying e.g. suitable inverse properties for the projection functions) is straightforward. For the VC-level variable values, we can directly instantiate the values in the initial Boogie program state.

VC-level functions representing those declared in the Boogie program are instantiated as (total) functions which, *for input values of appropriate type*, are defined simply to return the same values as the corresponding function in our model. However, perhaps surprisingly, Boogie's VC embedding of functions logically requires properties of these functions even in other cases. For example, for the `foo` function above, *some* value of the type $\mathit{List}_1^{-1}(\mathit{typ}(v))$ must be returned even for arguments which are not lists! We define the function to return some such value, which is possible since in well-formed contexts, every closed type has at least one value in our model.

Secondly, we need to prove the hypotheses of the VC's implication; in particular that all axioms (both those generated by the type system encoding and those coming from the program itself) are satisfied. The former are standard and simple to prove (given the work above), while the latter largely follow from the assumption on *executions* that each declared axiom must be satisfied in the initial state restricted to the constants. The only remaining challenge is to relate VC expressions with the evaluation of corresponding Boogie expressions; an issue which also arises (and is explained) below.

6.4 Certifying the VC Phase

Boogie's weakest precondition calculation is made size-efficient by the usage of explicit named constants for the weakest preconditions $wp(B, \texttt{true})$ for each block B, which is defined in terms of the named constants for its successor blocks. For example, in Fig. 4, $wp(B_2'', \texttt{true})$ is given by $i_1^{vc} \neq 0 \implies wp(B_3'', \texttt{true}) \wedge wp(B_4'', \texttt{true})$. Here i_1^{vc} is the value that we instantiated for the variable `i1`.

We exploit this modular construction of the generated weakest precondition for the local and global block theorems. We prove for each block B with commands cs the following local block lemma:

Theorem 3 (VC Phase Local Block Lemma).
If $A, \Lambda, \Gamma, \Omega \vdash \langle cs, \mathsf{N}(ns)\rangle \; [\rightarrow] \; s'$ *and* $wp(B, \texttt{true})$ *holds, then* $s' \neq \mathsf{F}$ *and if* s' *is a normal state, then* $\forall B_{suc} \in successors(B).\ wp(B_{suc}, \texttt{true})$.

Once one has proved this lemma for all blocks in the CFG, combining them to obtain the corresponding global block theorems (via our usual reverse walk of the CFG) is straightforward. The main challenge is in decomposing the proof for the local block lemma itself for a block B, for which we outline our approach next.

By this phase, the first command in B must be either an **`assume`** e or an **`assert`** e command. In the former case, we rewrite $wp(B, \texttt{true})$ into the form $e^{vc} \implies H$, where e^{vc} is the VC counterpart of e and where H corresponds to the weakest precondition of the remaining commands. This rewriting may involve undoing certain optimisations Boogie's implementation performed on the formula structure. Next, we need to prove that e evaluates to e^{vc} (see below). Hence, if e evaluates to **`true`** (the execution does not go to magic) then H' must be true, and we can continue inductively. The argument for **`assert`** e is similar but rewriting the VC to $e^{vc} \wedge H$ (i.e., e^{vc} and H must both hold); if e evaluates to e^{vc}, we know that the execution does not fail.

Proving that e evaluates to e^{vc} arises in both cases and also in our previous discharging of VC hypotheses. Note that e^{vc} is not a Boogie expression, but a shallowly embedded formula that includes the instantiations of quantified variables we constructed above. Showing this property works largely on syntax-driven rules that relate a Boogie expression with its VC counterpart, except for extra work due to mismatching function signatures (*cf.* Sec. 6.2) and optimisations that Boogie made either to the formula structure or via the type system encoding. We handle some of these cases by making Isabelle prove that we can rewrite the formula back into the unoptimised standard form we require for our syntax-driven rules and in other cases we use Isabelle to prove the goal directly.

This concludes our discussion of the certification of Boogie's three key phases. Combining the three certificates yields an end-to-end proof that the validity of the generated verification conditions implies the correctness of the input program, that is, that the given verification run is sound.

7 Implementation and Evaluation

In this section, we evaluate our certifying version of the Boogie verifier, which produces Isabelle certificates proving the correctness of Boogie's pipeline for programs it verifies.

Implementation. We have implemented our validation tool as a new C# module compiled with Boogie. We instrumented Boogie's codebase to call out to our module logging various information that we use to validate the key phases, and extended parts of the codebase to extract information more easily. Moreover, we disabled counter-example related VC features and the generation of VC axioms for any built-in types that we do not support. We added or changed only 143 lines of code across 6 files in the existing Boogie implementation.

Given an input file verified by Boogie, our work produces an Isabelle certificate per procedure p that certifies the correctness of its CFG as represented internally in Boogie. In addition to the three key phases we describe in detail, our implementation also handles several smaller transformations made by Boogie, such as constant propagation. Our tool currently supports the default options of Boogie (only) and does not support e.g. advanced source-level *attributes* (usable e.g. to selectively force procedures to be inlined).

Experimental Evaluation. We evaluated our work in two ways. Firstly, to evaluate the applicability of our certificate generation, we automatically collected all input files with at least one procedure from Boogie's own test suite [1] which verify successfully and which either use no unsupported features or are easily desugared (by hand) into versions without them. This includes programs with procedure calls since Boogie simply desugars these in an early stage. For programs employing unsupported attributes, we checked whether the program still verifies *without* attributes, and if so we also kept these. In total, this yields 95 programs from Boogie's test suite. Secondly, we collected a corpus of ten Boogie programs which verify interesting algorithms with non-trivial specifications: three from Boogie's test suite and seven from the literature [11, 25]. Where needed we manually desugared usages of Boogie maps (which our work does not yet support) using type declarations, functions, and axioms.

Of the 95 programs from Boogie's test suite, we successfully generate certificates in 89 cases (93%). The remaining 6 cases involve special cases that we do not handle yet. For 4 of them, extending our work is straightforward: one special

Name	LOC	#P	Time [s]	Size
TuringFactorial	29	1	17.1	1994
Find	27	2	39.7	2168
DivMod	69	2	34.7	4839
Summax [25]	23	1	16.9	1962
MaxOfArray [11]	22	1	17.1	1949
SumOfArray [11]	22	1	17.6	1539
Plateau [11]	50	1	20.8	2024
WelfareCrook [11]	52	1	38.4	2545
ArrayPartitioning [11]	56	2	38.0	3606
DutchFlag [11]	76	2	66.4	4124

Table 1. Selection of algorithmic examples with the lines of code (LOC), the number of procedures (#P), the time it takes for Isabelle to check the certficate in seconds (the average of 5 runs on a Lenovo T480 with 32 GB, i7-8550U 1.8 GhZ, Ubuntu 18.04 on the Windows Subsystem for Linux), and the certificate size expressed as the number of non-empty lines of Isabelle.

case is that Boogie's passification transforms a boolean assignment `x := e` to **`assume`** `xi` ⇔ `e'` instead of **`assume`** `xi = e'`. The other case includes a block with an edge directly to itself; this unusual case trips up our current implementation, which will be easily amended. The remaining two fail, because of our incomplete handling of function calls in the VC phase when combined with coercions between VC integers or booleans and their Boogie counterparts. Handling this is more challenging, but is not a fundamental issue.

For the corpus of 10 examples, Tab. 1 shows the generated certificate size and the time for Isabelle to check their validity[8]. The ratio of certificate size to code size ranges from 40 to 89; this rather large ratio emphasises the substantial work in precisely and formally validating the substantial work which Boogie's implementation performs. The validation of certificates takes usually under one second per line of code. While these times are not short, they are acceptable since certificate generation needs to run only for a final (verified) version of the program in question.

8 Related Work

Several works explore the validation of program verifiers. Garchery et al. [19] validate VC rewritings in the Why3 VC generator [15]. Unlike our work, they do not connect VCs with programs and do not handle the erasure of polymorphic types. Strub et al. [35] validate part of a previous version of the F* verifier [36] by generating a certificate for the F* type checker itself, which type checks programs by generating VCs. Like us, they assume the validity of the generated VC itself, but they do not consider program-to-program transformations such as

[8] The time to generate the certificate is not included, but is negligible for these examples.

ours. Another approach to validate verification results is taken by Aguirre [2] who shows how one can map proofs of the VC back to correctness of an F* program, which could be used in conjunction with the proof-producing capability of modern SMT solvers [6]; they prove a once-and-for-all result, but the approach could be directly lifted to validation. However, the work has not been implemented, and makes various assumptions about the VC proof that are not guaranteed by SMT solvers such as the proof being constructive and being in a normal form.

There is some work on proving VC generator implementations correct once and for all, although none of the proven tools are used in practice. Homeier and Martin [22] prove a VC generator correct in HOL for an executable programming language and a simpler VC generation technique than Boogie's. Herms et al. [21] prove a VC generator inspired by Why3 correct in Coq. However, some more-challenging aspects of Why3's VC transformation and polymorphic type system are not handled. Vogels et al. [41] prove a toolchain for a Boogie-like language correct in Coq, including passification and VC phases. However, the language is quite limited: without unstructured control flow, loops (i.e. no need for a CFG-to-DAG phase), functions, or polymorphism (i.e., no type encoding).

In the related context of compiler correctness, many validation techniques express a per-run validator in Coq, prove it correct once-and-for-all [7, 37, 39], and then extract executable code (the extraction must be trusted). One such work related to our certification of the passification phase is the validation of the SSA phase in CompCert [7], dealing also with versioned variables in the target (but not with **assume** statements that prune executions). In contrast to our work, they require an explicit notion of CFG domination and they do not use a global versioning scheme to efficiently check that two parts of the CFG constrain disjoint versions. Our versioning idea is similar to a technique used for the validation of a dominator relation in a CFG [8], which assigns intervals to basic blocks (as opposed to assigning versions to variables) to efficiently determine whether a block dominates another one. The validation of the Cogent compiler [34] follows a similar approach to ours in that it directly generates proofs in Isabelle.

9 Conclusion

We have presented and implemented a novel verifier validation approach, and applied it successfully to three key phases of the Boogie verifier, providing formal underpinnings for both the language and its verifier for the first time. Our work demonstrates that it is feasible to provide strong formal guarantees regarding the verification results of practical VC generators written in modern mainstream programming languages. In the future, we plan to investigate the extension and application of our overall validation approach to verification tools which map verification problems concerning other languages and logics into intermediate verification languages such as Boogie.

References

1. Boogie verifier repository. https://github.com/boogie-org/boogie
2. Aguirre, A.: Towards a provably correct encoding from F* to SMT. Inria Internship Report (Aug 2016), http://prosecco.gforge.inria.fr/personal/hritcu/students/alejandro/report.pdf
3. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging Rust types for modular specification and verification. In: Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). vol. 3, pp. 147:1–147:30. ACM (2019)
4. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: The Spec# experience. Communications of the ACM **54**(6), 81–91 (June 2011)
5. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: Workshop on Program Analysis for Software Tools and Engineering (PASTE). p. 82–87. PASTE '05 (2005). https://doi.org/10.1145/1108792.1108813
6. Barrett, C., de Moura, L., Fontaine, P.: Proofs in satisfiability modulo theories. In: Delahaye, D., Woltzenlogel Paleo, B. (eds.) All about Proofs, Proofs for All, Mathematical Logic and Foundations, vol. 55, pp. 23–44. College Publications (2015)
7. Barthe, G., Demange, D., Pichardie, D.: Formal verification of an SSA-based middle-end for compcert. Transactions on Programming Languages and Systems (TOPLAS **36**(1) (2014)
8. Blazy, S., Demange, D., Pichardie, D.: Validating dominator trees for a fast, verified dominance test. In: Urban, C., Zhang, X. (eds.) Interactive Theorem Proving (ITP). pp. 84–99 (2015)
9. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors tool set: Verification of parallel and concurrent software. In: Polikarpova, N., Schneider, S. (eds.) Integrated Formal Methods (IFM). Lecture Notes in Computer Science, vol. 10510, pp. 102–110. Springer (2007)
10. Böhme, S., Weber, T.: Fast LCF-style proof reconstruction for Z3. In: Kaufmann, M., Paulson, L.C. (eds.) Interactive Theorem Proving (ITP). Lecture Notes in Computer Science, vol. 6172, pp. 179–194. Springer (2010)
11. Chen, Y., Furia, C.A.: Triggerless happy – intermediate verification with a first-order prover. In: Polikarpova, N., Schneider, S. (eds.) Proceedings of the 13th International Conference on integrated Formal Methods (iFM). Lecture Notes in Computer Science, vol. 10510, pp. 295–311. Springer (September 2017)
12. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics. pp. 23–42. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
13. Coq Development Team, T.: The Coq Reference Manual, version 8.10 (2019), available electronically at http://coq.inria.fr/documentation
14. Ekici, B., Mebsout, A., Tinelli, C., Keller, C., Katz, G., Reynolds, A., Barrett, C.W.: Smtcoq: A plug-in for integrating SMT solvers into coq. In: Majumdar, R., Kuncak, V. (eds.) Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 10427, pp. 126–133. Springer (2017)
15. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems. pp. 125–128. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)

16. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 4590, pp. 173–177. Springer (2007)
17. Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: Generating compact verification conditions. In: Principles of Programming Languages (POPL). p. 193–205 (2001)
18. Fleury, M., Schurr, H.: Reconstructing veriT proofs in Isabelle/HOL. In: Reis, G., Barbosa, H. (eds.) Sixth Workshop on Proof eXchange for Theorem Proving (PxTP). EPTCS, vol. 301, pp. 36–50 (2019)
19. Garchery, Q., Keller, C., Marché, C., Paskevich, A.: Des transformations logiques passent leur certificat. In: Journées Francophones des Langages Applicatifs (JFLA) (2020)
20. Hecht, M.S., Ullman, J.D.: Flow graph reducibility. SIAM J. Comput. **1**(2), 188–202 (1972)
21. Herms, P., Marché, C., Monate, B.: A certified multi-prover verification condition generator. In: Verified Software: Theories, Tools, Experiments VSTTE (2012)
22. Homeier, P.V., Martin, D.F.: A mechanically verified verification condition generator. The Computer Journal **38**(2), 131–141 (1995)
23. Isabelle Development Team, T.: The Isabelle Documentation, version June 2019 (2019), available electronically at https://isabelle.in.tum.de/documentation.html
24. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c: A software analysis perspective. Formal Aspects of Computing **27**(3), 573–609 (2015)
25. Klebanov, V., Müller, P., Shankar, N., Leavens, G.T., Wüstholz, V., Alkassar, E., Arthan, R., Bronish, D., Chapman, R., Cohen, E., Hillebrand, M., Jacobs, B., Leino, K.R.M., Monahan, R., Piessens, F., Polikarpova, N., Ridge, T., Smans, J., Tobies, S., Tuerk, T., Ulbrich, M., Weiß, B.: The 1st verified software competition: Experience report. In: Butler, M., Schulte, W. (eds.) FM 2011: Formal Methods. pp. 154–168. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
26. Lal, A., Qadeer, S., Lahiri, S.K.: A solver for reachability modulo theories. In: Madhusudan, P., Seshia, S.A. (eds.) Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 7358, pp. 427–443. Springer (2012)
27. Leino, K.R.M.: This is Boogie 2 (June 2008), https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/
28. Leino, K.R.M.: Efficient weakest preconditions. Inf. Process. Lett. **93**(6), 281–288 (2005)
29. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning (LPAR). Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010)
30. Leino, K.R.M., Millstein, T.D., Saxe, J.B.: Generating error traces from verification-condition counterexamples. Science of Computer Programming **55**(1-3), 209–226 (2005)
31. Leino, K.R.M., Rümmer, P.: A polymorphic intermediate verification language: Design and logical encoding. In: Esparza, J., Majumdar, R. (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 6015, pp. 312–327. Springer (2010)
32. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: Principles of Programming Languages POPL. pp. 42–54 (2006)

33. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) Verification, Model Checking, and Abstract Interpretation (VMCAI). Lecture Notes in Computer Science, vol. 9583, pp. 41–62. Springer (2016)
34. Rizkallah, C., Lim, J., Nagashima, Y., Sewell, T., Chen, Z., O'Connor, L., Murray, T., Keller, G., Klein, G.: A framework for the automatic formal verification of refinement from Cogent to C. In: Blanchette, J.C., Merz, S. (eds.) Interactive Theorem Proving (ITP). pp. 323–340. Springer (2016)
35. Strub, P.Y., Swamy, N., Fournet, C., Chen, J.: Self-certification: Bootstrapping certified typecheckers in F* with Coq. In: Principles of Programming Languages (POPL). p. 571–584 (2012)
36. Swamy, N., Hritcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., Zinzindohoué, J.K., Zanella-Béguelin, S.: Dependent types and multi-monadic effects in F*. In: Principles of Programming Languages (POPL). pp. 256–270 (2016)
37. Tristan, J.B., Leroy, X.: Formal verification of translation validators: A case study on instruction scheduling optimizations. In: Necula, G.C., Wadler, P. (eds.) Principles of Programming Languages (POPL). pp. 17–27. ACM (2008)
38. Tristan, J.B., Leroy, X.: Verified validation of lazy code motion. In: Hind, M., Diwan, A. (eds.) Programming Language Design and Implementation (PLDI). pp. 316–326. ACM (2009)
39. Tristan, J.B., Leroy, X.: A simple, verified validator for software pipelining. In: Principles of Programming Languages (POPL). pp. 83–92 (2010)
40. Vogels, F., Jacobs, B., Piessens, F.: A machine checked soundness proof for an intermediate verification language. In: Nielsen, M., Kucera, A., Miltersen, P.B., Palamidessi, C., Tuma, P., Valencia, F.D. (eds.) Theory and Practice of Computer Science (SOFSEM). Lecture Notes in Computer Science, vol. 5404, pp. 570–581. Springer (2009)
41. Vogels, F., Jacobs, B., Piessens, F.: A machine-checked soundness proof for an efficient verification condition generator. In: Shin, S.Y., Ossowski, S., Schumacher, M., Palakal, M.J., Hung, C. (eds.) Symposium on Applied Computing (SAC). pp. 2517–2522. ACM (2010)

$$\begin{aligned}
\mathit{bop} &::= \ = \mid \neq \mid + \mid - \mid * \mid \leq \mid < \mid \geq \mid \wedge \mid \vee \mid \longrightarrow \mid \longleftrightarrow \qquad \mathit{uop} ::= \ - \mid \neg \\
\tau &::= t \mid \mathit{Int} \mid \mathit{Bool} \mid C(\vec{\tau}) \\
e &::= x \mid \texttt{\textbf{false}} \mid \texttt{\textbf{true}} \mid i \mid e_1 \ \mathit{bop} \ e_2 \mid \mathit{uop}(e) \mid f[\vec{\tau}](\vec{e}) \mid \texttt{\textbf{old}}(e) \mid \\
&\qquad \forall x : \tau.\ e \mid \exists x : \tau.\ e \mid \forall_{ty}\, t.\ e \mid \exists_{ty}\, \tau.\ e \\
s &::= \texttt{\textbf{assume}}\ e \mid \texttt{\textbf{assert}}\ e \mid x := e \mid \texttt{\textbf{havoc}}\ x
\end{aligned}$$

Fig. 5. The syntax of the Boogie subset that we formalise, where τ, e, and s, denote the types, expressions, and basic commands respectively; control-flow is handled via CFGs over the basic commands. *bop* and *uop* denote binary and unary operations, respectively. We assume that procedure calls have been desugared into basic commands.

A A Formal Semantics for Boogie

A.1 The Boogie Language: Syntax

The types, expressions and basic commands in our Boogie subset are shown in Fig. 5. We support the primitive types *Int* and *Bool*; other types (obtained via declared type constructors) are *uninterpreted types*; the sets of values such types may denote are constrained only via Boogie axioms and **`assume`** commands.

Expressions include variables, boolean/integeral literals, unary/binary expressions. We also supports function calls $f[\vec{\tau}](\vec{e})$. The arguments $\vec{\tau}$ to a function call $f[\vec{\tau}](\vec{e})$ instantiate any *type* parameters and are inferred by the type-checker; in our formalization type parameters are always explicit. The remaining expressoins are old expressions, value quantification ($\forall x : \tau.\ e / \exists x : \tau.\ e$), and type quantification ($\forall_{ty}\, t.\ e / \exists_{ty}\, t.\ e$).

The commands are given by assumptions, assertions, assignments and havoc commands. Sequential composition is represented by basic blocks that contain a list of commands.

Boogie source programs contain richer expressions and commands that can be desugared straightforwardly into our subset, such as havocs of multiple variables and combined type/value quantification with multiple binders. Some, such as procedure calls are already desugared by Boogie in pre-processing phases.

A.2 Expression evaluation.

The rules for expression evaluation are given in Fig. 6 (basic expressions), Fig. 7 (quantified expressions), and Fig. 8 (lists of expressions). The rule for variable lookup is defined in terms of the function $\mathit{lookup}((G, L), \mathit{gs}, \mathit{ls}, x)$, which returns $\mathit{ls}(x)$ if x is a local variable (i.e., x is recorded in the local variable declarations L) and $\mathit{gs}(x)$ otherwise. This models the fact that local variables shadow global variables.

In the rule for literals l_e and l_v denote literal expressions and the corresponding literal values respectively. The rules for value quantification are defined in terms of $\mathit{typ}_{\mathcal{T}}(v)$, which maps a value v to its type w.r.t. the type interpretation A.

$$\frac{\mathit{lookup}(\varLambda, gs, ls, x) = v}{\mathcal{T}, \varLambda, \varGamma, \varOmega \vdash \langle x, \mathsf{N}((os, gs, ls))\rangle \Downarrow v} \qquad \frac{}{\mathcal{T}, \varLambda, \varGamma, \varOmega \vdash \langle l_e, \mathsf{N}(ns)\rangle \Downarrow l_v}$$

$$\frac{\begin{array}{c} \mathcal{T}, \varLambda, \varGamma, \varOmega \vdash \langle e_1, \mathsf{N}(ns)\rangle \Downarrow v_1 \\ \mathcal{T}, \varLambda, \varGamma, \varOmega \vdash \langle e_2, \mathsf{N}(ns)\rangle \Downarrow v_2 \\ v_1\ \overline{bop}\ v_2 = v \end{array}}{\mathcal{T}, \varLambda, \varGamma, \varOmega \vdash \langle e_1\ bop\ e_2, \mathsf{N}(ns)\rangle \Downarrow v} \qquad \frac{\mathcal{T}, \varLambda, \varGamma, \varOmega \vdash \langle e, \mathsf{N}(ns)\rangle \Downarrow v' \quad \overline{uop}(v') = v}{\mathcal{T}, \varLambda, \varGamma, \varOmega \vdash \langle uop(e), \mathsf{N}(ns)\rangle \Downarrow v}$$

$$\frac{\begin{array}{c} A, \varLambda, \varGamma, \varOmega \vdash \langle \vec{e}, \mathsf{N}(ns)\rangle\ [\Downarrow]\ \vec{v'} \\ \varGamma(f) = \overline{f} \quad \overline{f}(\varOmega(\vec{\tau}), \vec{v'}) = v \end{array}}{\mathcal{T}, \varLambda, \varGamma, \varOmega \vdash \langle f[\vec{\tau}](\vec{e}), \mathsf{N}(ns)\rangle \Downarrow v} \qquad \frac{\mathcal{T}, \varLambda, \varGamma, \varOmega \vdash \langle e, \mathsf{N}((os, os, ls))\rangle \Downarrow v}{\mathcal{T}, \varLambda, \varGamma, \varOmega \vdash \langle \mathtt{\textbf{old}}(e), \mathsf{N}((os, gs, ls))\rangle \Downarrow v}$$

Fig. 6. Expression reduction for basic expressions

A.3 Command and CFG reduction.

The rules for the reduction of commands and lists of commands is given in Fig. 9 and Fig. 10. Assignment reduces only if the value to be assigned has the right type, i.e., assignment preserves well-typed states. This restriction is not required for well-typed programs, but it nevertheless makes some reasoning easier. The rules for assignment and havoc rely on $\mathit{lookup}_T(\varLambda, x)$ that maps x to its type w.r.t. the variable context $\varLambda$ (if x is defined), and on $\mathit{update}(\varLambda, ns, x, v)$, which returns the state ns where x is updated to v (ensuring that the local state is updated if x is local and the global state otherwise).

The rules for the CFG reduction are given in Fig. 11. $\mathit{cmds}(G, b)$ gives the list of commands for block b in CFG G.

A.4 Procedure Correctness

As already discussed in Sec. 3.3, a procedure is correct w.r.t. a context if it is correct w.r.t. to all *well-formed* contexts relative to the program. A context $(\mathcal{T}, \varLambda, \varGamma, \varOmega)$ is well-formed relative to a Boogie program if the following holds:

- The type interpretation $\mathcal{T}$ maps some abstract value to every closed type obtained via a type constructor (i.e., closed types are inhabited by non-empty sets of values). A type is closed if it does not contain type variables.
- $\varGamma$ interprets each declared function f consistently with its signature.

B The Phases For the Running Example

For our running example in Fig. 2, the full CFG is shown in Fig. 12. The full CFG after the CFG-to-DAG phase is shown in Fig. 13. Finally, the full CFG after the passification phase is shown in Fig. 14. In practice, Boogie applies a constant

Value quantification

$$\frac{\forall w.\ \mathit{typ}_{\mathcal{T}}(w) = \mathit{\Omega}(\tau) \implies \mathcal{T}, (G, L, \mathit{\Gamma}, \mathit{\Omega} \vdash \langle e, \mathsf{N}((os, gs, ls(x \mapsto w)))\rangle \Downarrow \texttt{\textbf{true}}}{\mathcal{T}, (G, L), \mathit{\Gamma}, \mathit{\Omega} \vdash \langle \forall x : \tau.\ e, \mathsf{N}((os, gs, ls))\rangle \Downarrow \texttt{\textbf{true}}}$$

$$\frac{\mathit{typ}_{\mathcal{T}}(w) = \mathit{\Omega}(\tau) \quad \mathcal{T}, (G, L), \mathit{\Gamma}, \mathit{\Omega} \vdash \langle e, \mathsf{N}((os, gs, ls(x \mapsto w)))\rangle \Downarrow \texttt{\textbf{false}}}{\mathcal{T}, (G, L), \mathit{\Gamma}, \mathit{\Omega} \vdash \langle \forall x : \tau.\ e, \mathsf{N}((os, gs, ls))\rangle \Downarrow \texttt{\textbf{false}}}$$

$$\frac{\mathit{typ}_{\mathcal{T}}(w) = \mathit{\Omega}(\tau) \quad \mathcal{T}, (G, L, \mathit{\Gamma}, \mathit{\Omega} \vdash \langle e, \mathsf{N}((os, gs, ls(x \mapsto w)))\rangle \Downarrow \texttt{\textbf{true}}}{\mathcal{T}, (G, L), \mathit{\Gamma}, \mathit{\Omega} \vdash \langle \exists x : \tau.\ e, \mathsf{N}((os, gs, ls))\rangle \Downarrow \texttt{\textbf{true}}}$$

$$\frac{\forall w.\ \mathit{typ}_{\mathcal{T}}(w) = \mathit{\Omega}(\tau) \implies \mathcal{T}, (G, L, \mathit{\Gamma}, \mathit{\Omega} \vdash \langle e, \mathsf{N}((os, gs, ls(x \mapsto w)))\rangle \Downarrow \texttt{\textbf{false}}}{\mathcal{T}, (G, L), \mathit{\Gamma}, \mathit{\Omega} \vdash \langle \exists x : \tau.\ e, \mathsf{N}((os, gs, ls))\rangle \Downarrow \texttt{\textbf{false}}}$$

Type quantification

$$\frac{\forall \tau.\ \mathit{closed}(\tau) \implies \mathcal{T}, \mathit{\Lambda}, \mathit{\Gamma}, \mathit{\Omega}(t \mapsto \tau) \vdash \langle e, \mathsf{N}(ns)\rangle \Downarrow \texttt{\textbf{true}}}{\mathcal{T}, \mathit{\Lambda}, \mathit{\Gamma}, \mathit{\Omega} \vdash \langle \forall_{ty}\, t.\ e, \mathsf{N}(ns)\rangle \Downarrow \texttt{\textbf{true}}}$$

$$\frac{\mathit{closed}(\tau) \quad \mathcal{T}, \mathit{\Lambda}, \mathit{\Gamma}, \mathit{\Omega}(t \mapsto \tau) \vdash \langle e, \mathsf{N}(ns)\rangle \Downarrow \texttt{\textbf{false}}}{\mathcal{T}, \mathit{\Lambda}, \mathit{\Gamma}, \mathit{\Omega} \vdash \langle \forall_{ty}\, t.\ e, \mathsf{N}(ns)\rangle \Downarrow \texttt{\textbf{false}}}$$

$$\frac{\mathit{closed}(\tau) \quad \mathcal{T}, \mathit{\Lambda}, \mathit{\Gamma}, \mathit{\Omega}(t \mapsto \tau) \vdash \langle e, \mathsf{N}(ns)\rangle \Downarrow \texttt{\textbf{true}}}{\mathcal{T}, \mathit{\Lambda}, \mathit{\Gamma}, \mathit{\Omega} \vdash \langle \exists_{ty}\, t.\ e, \mathsf{N}(ns)\rangle \Downarrow \texttt{\textbf{true}}}$$

$$\frac{\forall \tau.\ \mathit{closed}(\tau) \implies \mathcal{T}, \mathit{\Lambda}, \mathit{\Gamma}, \mathit{\Omega}(t \mapsto \tau) \vdash \langle e, \mathsf{N}(ns)\rangle \Downarrow \texttt{\textbf{false}}}{\mathcal{T}, \mathit{\Lambda}, \mathit{\Gamma}, \mathit{\Omega} \vdash \langle \exists_{ty}\, t.\ e, ns\rangle \Downarrow \texttt{\textbf{false}}}$$

Fig. 7. Expression reduction for quantifiers.

$$\frac{}{A, \mathit{\Lambda}, \mathit{\Gamma}, \mathit{\Omega} \vdash \langle \mathsf{nil}, \mathsf{N}(ns)\rangle\ [\Downarrow]\ \mathsf{nil}} \qquad \frac{\begin{array}{c} A, \mathit{\Lambda}, \mathit{\Gamma}, \mathit{\Omega} \vdash \langle e, \mathsf{N}(ns)\rangle \Downarrow v \\ A, \mathit{\Lambda}, \mathit{\Gamma}, \mathit{\Omega} \vdash \langle es, \mathsf{N}(ns)\rangle\ [\Downarrow]\ vs \end{array}}{A, \mathit{\Lambda}, \mathit{\Gamma}, \mathit{\Omega} \vdash \langle (e : es), \mathsf{N}(ns)\rangle\ [\Downarrow]\ (v : vs)}$$

Fig. 8. Expression reduction for lists of expressions

$$\frac{\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle e, \mathsf{N}(ns)\rangle \Downarrow \texttt{\textbf{true}}}{\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle \texttt{\textbf{assert}}\ e, \mathsf{N}(ns)\rangle \to \mathsf{N}(ns)} \quad \frac{\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle e, \mathsf{N}(ns)\rangle \Downarrow \texttt{\textbf{false}}}{\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle \texttt{\textbf{assert}}\ e, \mathsf{N}(ns)\rangle \to \mathsf{F}}$$

$$\frac{\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle e, \mathsf{N}(ns)\rangle \Downarrow \texttt{\textbf{true}}}{\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle \texttt{\textbf{assume}}\ e, \mathsf{N}(ns)\rangle \to \mathsf{N}(ns)} \quad \frac{\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle e, \mathsf{N}(ns)\rangle \Downarrow \texttt{\textbf{false}}}{\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle \texttt{\textbf{assume}}\ e, \mathsf{N}(ns)\rangle \to \mathsf{M}}$$

$$\frac{\begin{array}{c} \mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle e, \mathsf{N}(ns)\rangle \Downarrow v \\ \mathit{lookup}_T(\Lambda, x) = \tau \quad \mathit{typ}_{\mathcal{T}}(v) = \Omega(\tau) \\ ns' = \mathit{update}(\Lambda, ns, x, v) \end{array}}{\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle x := e, \mathsf{N}(ns)\rangle \to \mathsf{N}(ns')} \quad \frac{\begin{array}{c} \mathit{lookup}_T(\Lambda, x) = \tau \quad \mathit{typ}_{\mathcal{T}}(v) = \Omega(\tau) \\ ns' = \mathit{update}(\Lambda, ns, x, v) \end{array}}{\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle \textbf{havoc}\ x, \mathsf{N}(ns)\rangle \to \mathsf{N}(ns')}$$

$$\frac{}{\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle c, \mathsf{M}\rangle \to \mathsf{M}} \qquad \frac{}{\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle c, \mathsf{F}\rangle \to \mathsf{F}}$$

Fig. 9. Command reduction

$$\frac{}{\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle \mathsf{nil}, s\rangle\ [\to]\ s} \quad \frac{\begin{array}{c} \mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle c, s\rangle \to s'' \\ \mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle cs, s''\rangle\ [\to]\ s' \end{array}}{\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle (c : cs), \mathsf{N}(ns)\rangle\ [\to]\ s'}$$

Fig. 10. Reduction for lists of commands

$$\frac{\begin{array}{c} \mathit{cmds}(G, b) = cs \quad b' \in \mathit{successors}(G, b) \\ \mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle cs, \mathsf{N}(ns)\rangle\ [\to]\ \mathsf{N}(ns') \end{array}}{\mathcal{T}, \Lambda, \Gamma, \Omega, G \vdash (\mathsf{inl}(b), \mathsf{N}(ns)) \to_{\mathsf{CFG}} (\mathsf{inl}(b'), \mathsf{N}(ns'))}$$

$$\frac{\begin{array}{c} \mathit{cmds}(G, b) = cs \quad \mathit{successors}(G, b) = \emptyset \\ \mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle cs, \mathsf{N}(ns)\rangle\ [\to]\ \mathsf{N}(ns') \end{array}}{\mathcal{T}, \Lambda, \Gamma, \Omega, G \vdash (\mathsf{inl}(b), \mathsf{N}(ns)) \to_{\mathsf{CFG}} (\mathsf{inr}(()), \mathsf{N}(ns'))}$$

$$\frac{\begin{array}{c} \mathit{cmds}(G, b) = cs \\ \mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle cs, \mathsf{N}(ns)\rangle\ [\to]\ \mathsf{M} \end{array}}{\mathcal{T}, \Lambda, \Gamma, \Omega, G \vdash (\mathsf{inl}(()), \mathsf{N}(ns)) \to_{\mathsf{CFG}} (\mathsf{inr}(b'), \mathsf{M})}$$

$$\frac{\begin{array}{c} \mathit{cmds}(G, b) = cs \\ \mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle cs, \mathsf{N}(ns)\rangle\ [\to]\ \mathsf{F} \end{array}}{\mathcal{T}, \Lambda, \Gamma, \Omega, G \vdash (\mathsf{inl}(()), \mathsf{N}(ns)) \to_{\mathsf{CFG}} (\mathsf{inr}(b'), \mathsf{F})}$$

Fig. 11. CFG reduction

propagation transformation as part of the passifcation phase. Moreover, multiple empty blocks are added as well during the three phases. We ignore both these points here for the sake of presentation, but we handle them in our validation tool.

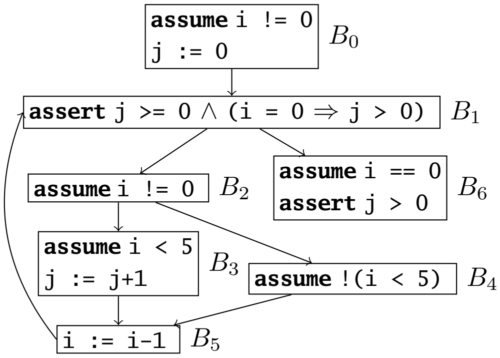


Fig. 12. CFG representation of running example

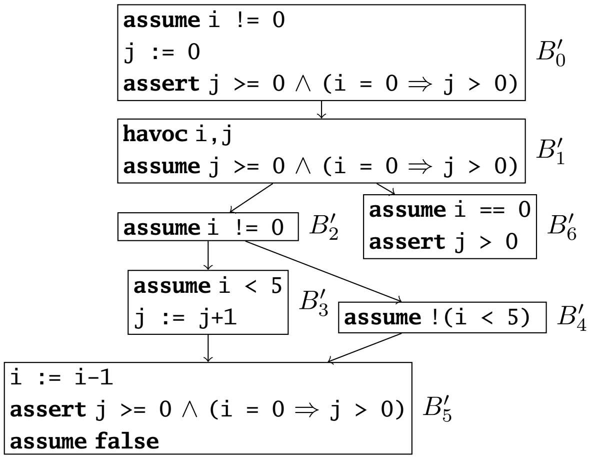
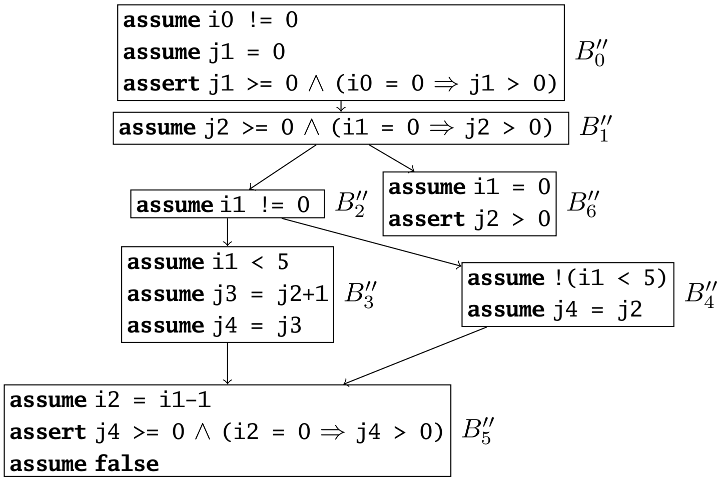


Fig. 13. CFG representation of running example after CFG-to-DAG phase



```
assume i0 != 0
assume j1 = 0
assert j1 >= 0 ∧ (i0 = 0 ⇒ j1 > 0)
```

```
assume j2 >= 0 ∧ (i1 = 0 ⇒ j2 > 0)
```

```
assume i1 != 0
```

```
assume i1 = 0
assert j2 > 0
```

```
assume i1 < 5
assume j3 = j2+1
assume j4 = j3
```

```
assume !(i1 < 5)
assume j4 = j2
```

```
assume i2 = i1-1
assert j4 >= 0 ∧ (i2 = 0 ⇒ j4 > 0)
assume false
```

Fig. 14. CFG representation of running example after passification phase