ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Formally Validating the AST-to-CFG Phase of the Boogie Program Verifier

Bachelor's Thesis

Aleksandar Hubanov

August 23, 2022

Advisors: Gaurav Parthasarathy, Prof. Dr. Peter Müller

Department of Computer Science, ETH Zürich

Abstract

The outputs of a program verifier are trustworthy only if the implementation of the program verifier is correct. Unfortunately, program verifier implementations are rarely verified. One exception is the Boogie program verifier. Existing work provides formal guarantees for parts of its implementation by generating per-run validation certificates in Isabelle via an instrumentation of the Boogie verifier. However, the Boogie verifier implementation contains a transformation from an abstract syntax tree (AST) program representation to a control flow graph (CFG) representation for which the prior work does not provide any formal guarantees in its generated certificate.

The work presented in this report extends the prior work to automatically produce Isabelle certificates that validate the AST-to-CFG transformation. We evaluate our work on the benchmarks used to evaluate the prior Boogie certificate generation work along with additional examples.

Contents

Chapter 1

Introduction

Program verifiers are tools concerned with proving the correctness of a given program with respect to a formal specification. Unfortunately, formal guarantees regarding the implementations of program verifiers are often not supplied. As a result, large parts of the verifier implementation (consisting of many lines of code) are often part of the *trusted code base*. That is, one must trust the verifier implementation itself to be correct in order to deduce that the verifier's produced verification result is correct. One exception is the Boogie program verifier, which verifies programs written in the Boogie intermediate verification language [1]. Parthasarathy et al. [2] instrument the Boogie verifier by adding a validation engine, which can be used to remove a large fraction of Boogie's implementation from the trusted code base. The validation engine employs a formalization of a subset of the Boogie language to automatically produce per-run formal proofs in the interactive theorem prover Isabelle [3], which serve as certificates for the correctness of Boogie's outputs.

The Boogie verifier verifies an input Boogie program via a series of program-to-program transformations before finally producing a mathematical formula called a *verification condition* (VC). Figure 1.1 illustrates this. In the figure, G_1 denotes the *abstract syntax tree* (AST) of the program, which Boogie builds after it parses the source code of the program. Boogie then transforms the abstract syntax tree into a *control-flow graph* (CFG), denoted by G_2 in the figure. We refer to this transformation as the *AST-to-CFG phase* of Boogie. After this, Boogie applies basic optimizations to the CFG to create what we informally call the "optimized CFG", denoted by G_3 in the figure. From the optimized CFG, Boogie produces the verification condition during the *CFG-to-VC phase*. Both the optimizations and the CFG-to-VC phase consist of multiple "smaller" phases. After having created the VC, Boogie queries an external SMT solver to determine if the VC is *valid* and, if it is, Boogie outputs that the input program is correct. (Correctness of a program includes that no assertion in the program can ever fail.)

A validation certificate that captures the correctness of the Boogie implementation from the AST to the VC for a single run of Boogie requires showing that the validity of the VC implies that the AST of the input program is correct: $\mathsf{valid}(\text{VC}) \models \mathsf{correct}(G_1)$. Proving that the verification condition is indeed valid if the SMT solver says so is an orthogonal concern that relates to the correctness of the SMT solver.

The validation engine for Boogie provided in [2] produces per-run certificates for the CFG-to-VC phase of Boogie. Referring to figure 1.1, such a certificate shows that $\mathsf{valid}(\mathrm{VC}) \models \mathsf{correct}(G_3)$. In order to achieve the desired certificate, one needs to additionally prove $\mathsf{correct}(G_3) \models \mathsf{correct}(G_2)$ (validation of optimizations), and $\mathsf{correct}(G_2) \models \mathsf{correct}(G_1)$ (validation of the AST-to-CFG phase). The goal of the work presented in this report is to extend the validation engine to produce certificates that validate the AST-to-CFG phase of Boogie.

$$\begin{tikzcd}
\boxed{G_1} \arrow[r, "\text{AST-to-CFG}"] & \boxed{G_2} \arrow[r, "\text{Optimizations}"] \arrow[l, dashed, bend left, "\substack{\text{Goal of this work:}\\ \mathsf{correct}(G_2) \models \mathsf{correct}(G_1)}"] & \boxed{G_3} \arrow[r, "\text{CFG-to-VC}"] & \boxed{\mathrm{VC}} \arrow[l, dashed, bend left, "\substack{\text{Prior work [2]:}\\ \mathsf{valid}(\mathrm{VC}) \models \mathsf{correct}(G_3)}"]
\end{tikzcd}$$

Figure 1.1: Outline of the program-to-program transformations Boogie applies in order to generate a verification condition (VC) from an abstract syntax tree (AST).

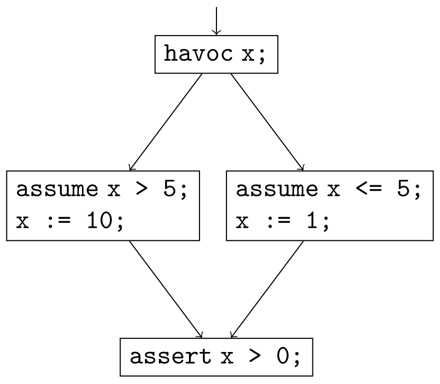
1.1 The AST-to-CFG Phase and the Optimizations

In the AST-to-CFG phase, Boogie transforms a program represented as an astract syntax tree (AST) into a control-flow graph (CFG). The AST is a direct representation of the source code via an inductively defined datatype. In the examples given in this report, we will occasionally omit showing the AST form explicitly and instead directly refer to the source code form. The CFG in Boogie is a graph that illustrates all possible execution paths through the program. It eliminates branching constructs such as if-clauses, while-loops and breaks. Other control-flow statements, such as jumps and return-statements, are treated as transitions between nodes in the graph and unconditional halts, respectively.

Figure 1.2 shows two representations of the same Boogie program, where the source code is shown on the left and the CFG form is shown on the right. The goal of the AST-to-CFG phase is to transform an AST representation of the source code into the CFG on the right. Figure 1.3 shows the same for a program containing a while-loop. We will use both of these as running examples throughout this document.

After the AST-to-CFG phase has been completed, Boogie optimizes the CFG. It does this by applying optimizations such as pruning of unreachable blocks, elimination of dead variables and block coalescing (combining two blocks of the CFG into one if the second block is the only successor of the first and if the second block has no further predecessors).

```
havoc x;
if(x > 5)
{
  x := 10;
} else {
  x := 1;
}
assert x > 0;
```



```
havoc x;
assume x > 5;
x := 10;
assume x <= 5;
x := 1;
assert x > 0;
```

Figure 1.2: Example program containing an if-statement in the form of its source code on the left and the same program in its CFG form on the right.

1.2 Approach Towards Validation of the Phase

The work presented in this report uses the formalization provided in [2] as a foundation and follows the same approach towards validation. The formalization given there is in terms of the syntax and semantics of Boogie's CFG.

We devise and implement a formalization of Boogie's internal AST representation and a corresponding semantics in Isabelle building on the existing CFG formalization and semantics. We do include goto- and break-statements in the formalization but we do not support proof generation for programs containing either of those yet. Then, we employ this formalization in deriving general lemmas in Isabelle, which are used in producing the concrete per-run certificates for the phase. Lastly, we extend the already existing instrumentation of Boogie to realize the automation of the proof generation.

1.3 Outline

In Chapter 2, we clarify how Boogie builds the AST from the source code and we present our formalization of the syntax and semantics of the AST. In Chapter 3, we break down and explain the proof generation approach we use and we show our theorems that realize it. We also describe some key implementation details of our work. In Chapter 4, we clarify how we integrate our work into the Boogie verifier modules as an extension of the previous work on its validation [2] and we present the qualitative and quantitative evaluation of our work. That is, we discuss the example Boogie programs we applied our work on and we show a performance comparison in terms of time between the generation of a certificate for both the AST-to-CFG phase and the CFG-to-VC phase versus the generation of a certificate just for the CFG-to-VC phase. In Chapter 5, we draw conclusions and outline possibilities for future work on the project. For

```
i := 0;
assume n > 0;
while(i < n)
  invariant i <= n;
{
  i := i + 1;
}
assert i >= n;
```

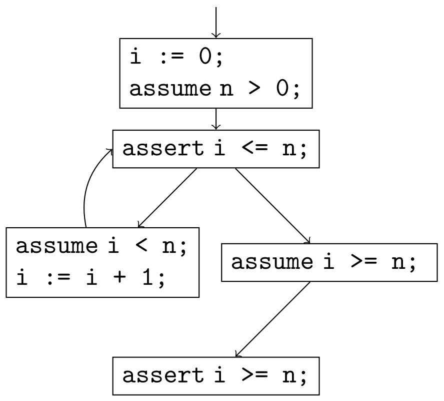


Figure 1.3: Example program containing a while-loop in the form of its source code on the left and the same program in its CFG form on the right.

reference, we also provide an appendix to this document, where we give a brief overview the Boogie language and some of its elements.

Chapter 2

Formalization of Boogie's Abstract Syntax Tree

In this chapter we discuss Boogie's abstract syntax tree (AST) and we present our formalization for it. We outline how it is built and, in subsection 2.1, we give a detailed explanation of its structure, content and characteristics. Following this, in subsection 2.2 we provide a formal syntax and semantics of the AST. Such a formalization is required in order to achieve the goal of constructing a correctness proof for the AST-to-CFG phase of Boogie.

In Boogie, a program comprises a set of procedures. Each procedure has a procedure body and a specification. After a user runs Boogie with some input program, Boogie converts it into a stream of parsing tokens. After this, Boogie's parser module iterates through this stream and, for the body of each procedure in the program, it builds what we refer to as the internal abstract syntax tree of the procedure. For the rest of this document, we always speak of an AST for a single procedure body.

2.1 Boogie's Internal Abstract Syntax Tree

Boogie builds the internal AST of a procedure body by dividing and thereby delineating the source code of the procedure into parts which are to be executed consecutively. Each part has at most two elements:

1. A sequence of *simple commands*. These are program statements in the Boogie language, which are not control-flow. Examples include `assume`- and `assert`-statements, assignments and `havoc`-statements. Refer to appendix A for more details on the Boogie language.

2. A single program construct that indicates a branch or a stopping point in the code - an if-statement, a while-loop, a break-statement, a goto-statement or a return-statement.

Each part is encoded into a special construct called an *AST block*. Therefore, *the internal AST of a procedure is an ordered list of AST blocks.*

Figures 2.1 and 2.2 outline the separation of the source code into parts to be encoded into AST blocks for our two example programs. In figure 2.1 the

```
--- start of AST block 0 ---

havoc x;
if(x > 5)
{
  --- start of AST block 1, nested in AST block 0 ---

  x := 10;

  --- end of AST block 1, nested in AST block 0 ---

} else {

  --- start of AST block 2, nested in AST block 0 ---

  x := 1;

  --- end of AST block 2, nested in AST block 0 ---
}

--- end of AST block 0 ---

--- start of AST block 3 ---

assert x > 0;

--- end of AST block 3 ---
```

Figure 2.1: Separation of source code into parts delineating the AST blocks for a program with an if-statement.

presence of the if-statement causes Boogie to split the procedure right after it, thus creating two parts and, accordingly, two AST blocks. AST block 0 encodes the if-statement in itself. The content of each branch of the if-statement is also encoded in AST blocks as illustrated. These AST blocks are recursively contained or *nested* in AST block 0. Similarly, in figure 2.2, due to the presence of the while-loop, Boogie partitions the procedure into two parts encoded into two AST blocks, where the first one recursively contains another AST block which encodes the content of the loop body.

Our Isabelle formalization (described in Section 2.2) of a Boogie AST closely follows Boogie's internal AST. That's why we now give a more detailed explanation of an AST block as represented in Boogie's implementation.

An AST block consists of an optional name, a list of simple commands, an optional *structured command* and an optional *transfer command*:

1. **Name**: If a label exists as a program statement in the procedure, upon

```
--- start of AST block 0 ---

i := 0;
assume n > 0;
while(i < n)
  inv i <= n;
{
  --- start of AST block 1, nested in AST block 0 ---

  i := i + 1;

  --- end of AST block 1, nested in AST block 0 ---
}

--- end of AST block 0 ---

--- start of AST block 2 ---

assert i >= n;

--- end of AST block 2 ---
```

Figure 2.2: Separation of source code into parts delineating the AST blocks for a program with a while-loop.

encounter by the parser, its corresponding token immediately triggers the creation of a new AST block and the label serves as a name for that AST block. Otherwise, the AST block has no name.

2. **Simple Commands**: A possibly-empty list of simple commands.

3. **Structured Command**: This is an encoding of an if-statement, an encoding of a while-loop or an encoding of a break statement. We refer to these encodings as an *If-Command*, a *While-Command* and a *Break-Command*, respectively. If necessary, an If-Command and a While-Command may themselves contain lists of AST blocks, so as to accurately reflect the program construct they represent.

   - **If-Command**: contains a *guard* condition (an expression), a possibly-empty list of AST blocks corresponding to the branch satisfying the guard, and possibly-empty optional lists of AST blocks corresponding to the else-if and else- branches.
   - **While-Command**: contains a *guard* condition and a possibly-empty list of AST blocks corresponding to the body of the loop.

4. **Transfer Command**: This is an encoding of a return-statement or an encoding of a goto-statement.

An AST block has a couple of notable characteristics. Firstly, it can never contain a structured command and a transfer command at the same time. Secondly, if an AST block contains a structured command *str* (or a transfer command *tr*), then the simple commands it contains (if any) always precede *str* (or *tr*) in the code. Any simple commands, which come after *str* (or *tr*) are delegated to a following AST block. Summarized, this means that an AST block is either 1.) a contiguous sequence of simple commands, 2.) a contiguous sequence of simple commands followed by a structured command or 3.) a contiguous sequence of simple commands followed by a transfer command.

Figures 2.3 and 2.4 illustrate the concrete AST representations of our example procedures. Both procedures are broken down into AST blocks without names, since there are no explicit label statements in either of them. In AST block 0 in figure 2.3 the list of simple commands contains only `havoc x` as this is the only simple command before the if-statement. In accordance to the description in the previous paragraph, the simple command `assert x > 0` cannot be in AST block 0 and is therefore delegated to AST block 3 which contains only `assert x > 0` and nothing else. The structured command in AST block 0 is an If-Command, since it encodes an if-statement and there is no transfer command as there is no return- or goto-statement in the procedure. Similar reasoning applies for figure 2.4 where the structured command in AST block 0 is a While-Command instead of an If-Command.

```
--- start of AST block 0 ---

havoc x;
if(x > 5)
{
  --- start of AST block 1 ---

  x := 10;

  --- end of AST block 1 ---

} else {

  --- start of AST block 2 ---

  x := 1;

  --- end of AST block 2 ---
}

--- end of AST block 0 ---

--- start of AST block 3 ---

assert x > 0;

--- end of AST block 3 ---
```

AST block 0

```
Name: -
Simple Commands: [ havoc x; ]
Structured Command:
  If-Command {
    guard: x > 5;,
    then-branch: AST block 1,
    else-branch: AST block 2 }
Transfer Command: -
```

AST block 3

```
Name: -
Simple Commands: [ assert x > 0; ]
Structured Command: -
Transfer Command: -
```

Figure 2.3: Example program with an if-statement in the form of its (annotated) source code on the left and its concrete internal AST representation on the right.

```
--- start of AST block 0 ---

i := 0;
assume n > 0;
while(i < n)
  inv i <= n;
{
  --- start of AST block 1 ---

  i := i + 1;

  --- end of AST block 1 ---
}

--- end of AST block 0 ---

--- start of AST block 2 ---

assert i >= n;

--- end of AST block 2 ---
```

AST block 0

```
Name: -
Simple Commands: [ i := 0; assume n > 0; ]
Structured Command:
  While-Command {
    guard: i < n,
    invs: [ i <= n ]
    loop body: AST Block 1,
Transfer Command: -
```

AST block 2

```
Name: -
Simple Commands: [ assert i >= n; ]
Structured Command: -
Transfer Command: -
```

Figure 2.4: Example program with a while-loop in the form of its (annotated) source code on the left and its concrete internal AST representation on the right.

2.2 Formalization in Isabelle

We formally model the behaviour of the AST by formalizing the syntax and semantics of the AST blocks in Isabelle. We chose this approach, so as to reflect Boogie's representation as closely as possible. This makes the connection between our formalization and Boogie's implementation clearer. Another approach would have been to formalize a more generic notion of an AST that could have possibly reflected the original source code more closely.

2.2.1 Formalization of the AST Syntax

Figure 2.5 shows the formal syntax of a structured command, a transfer command, an AST block and an AST as datatypes in Isabelle. The *simplecmd* type is part of the existing formalization of the Boogie's control flow graph [2] and it is congruent with the notion of a simple command outlined in the previous section. Similarly, the *expr* type is congruent with the notion of an expression.

A while-loop in our formalization is a **WhileWrapper** term applied to a **While** term as illustrated in figure 2.6. The *None* keyword in the figure denotes non-existence of the name, structured command and transfer command of the AST block. (We elaborate on *Some* and *None* below.) We employ the

$$
\begin{aligned}
\mathit{str} ::= {} & \textbf{If}\ \mathit{expr}\ [\mathit{astblock}]\ [\mathit{astblock}] \mid \textbf{While}\ \mathit{expr}\ [\mathit{expr}]\ [\mathit{astblock}] \mid \\
& \textbf{Break}\ \mathit{nat} \mid \textbf{WhileWrapper}\ \mathit{str} \\
\mathit{tr} ::= {} & \textbf{Return} \mid \textbf{Goto}\ s \\
\mathit{astblock} ::= {} & \\
& \textbf{ASTBlock}\ \ \textbf{option}\!<s>\ \ [\mathit{simplecmd}]\ \ \textbf{option}\!<\mathit{str}>\ \ \textbf{option}\!<\mathit{tr}> \\
\mathit{ast} ::= {} & [\mathit{astblock}]
\end{aligned}
$$

Figure 2.5: Formalization of the AST syntax in Isabelle

WhileWrapper term even though the **While** term already formalizes the syntax of a while-loop fully. The **WhileWrapper** term is used in the inference rules of the semantics to identify the first time a while-loop is encountered in a trace. After this first instance, the **WhileWrapper** is removed. This special treatment of the first encounter is useful for formalizing the semantics of break statements. We go into more detail about how the **WhileWrapper** is useful in section 2.2.2.

```
while(i < n)
  inv i <= n;
{
  i := i + 1;
}
```

$$
\boxed{\begin{array}{l}
\textbf{WhileWrapper} \\
\quad (\textbf{While}\ \ (\mathsf{i} < \mathsf{n})\ \ [\mathsf{i} \leq \mathsf{n}] \\
\qquad (\textbf{ASTBlock}\ \ \mathit{None}\ \ [\mathsf{i} := \mathsf{i} + 1]\ \ \mathit{None}\ \ \mathit{None}))
\end{array}}
$$

Figure 2.6: Formalization of a while-loop in Isabelle

The type *nat* in figure 2.5 represents natural numbers. A **Break** type constructor with a *nat* argument formalizes the notion of a *numbered* break statement. Numbered break statements are a simplified version of *labeled* break statements in Boogie where a label associated with some outer enclosing loop l_1 is provided. The loop associated with the label is the loop that is to be jumped out of. So, instead of specifying exactly *which* loop is to be broken out of, in our model, the number in the numbered break indicates *how many* loops to jump out of until one has broken out of the loop associated with l_1.

In figure 2.5, the types **option**$< s >$, **option**$< str >$ and **option**$< tr >$ capture the possibility that an AST block may or may not contain a name, a structured command or a transfer command, respectively. *Some* and *None* are the type constructors that comprise this polymorphic **option**<> type. *None* is nullary, *Some* encapsulates a data type.

Additionally, there are two discrepancies between our formalization and the syntax of Boogie's AST. Our formalization models *else*-branches of a given if-statement explicitly, even if the if-statement has no such branch in Boogie. For this reason we always add an empty *else*-branch to if-statements, which have none. We elaborate on this in section 3.3. The second discrepancy concerns *else-if*-branches. Our formalization does not handle these currently.

2.2.2 Formalization of the AST Semantics

In this section, we present our formal semantics for Boogie's AST via a small-step semantics describing the possible executions through the AST of a procedure in Boogie. We model the semantics via *continuations* as is done by Appel and Blazy [4] and Leroy [5].

In our formalization we reuse the formalized notion of a *Boogie program state* from [2]. (Sometimes we say just "program state".) As defined there, a program state could be a *failure state* F, reached when an assertion or a loop invariant is not satisfied, a *magic state* M, reached when an `assume`-statement is not satisfied and a *normal state* $\mathsf{N}(ns)$ in all other cases, where ns is a partial mapping from variables to values.

We employ the notion of a Boogie program state to introduce and formalize the notion of an *AST configuration.* This is a triplet of an AST Block, a continuation and a state as illustrated in figure 2.7. A *trace* through the AST in our formalization is a sequence of AST configurations.

$$\mathit{ASTConfig} ::= \mathit{(astblock,\ cont,\ state)}$$

Figure 2.7: AST Configuration

Our continuations are similar to the ones used by Appel and Blazy [4], but adjusted for the case of Boogie. Figure 2.8 illustrates our definition of a continuation.

$$\mathit{cont} ::= \mathbf{KStop} \mid \mathbf{KSeq}\ \mathit{astblock}\ \mathit{cont} \mid \mathbf{KEndBlock}\ \mathit{cont}$$

Figure 2.8: Continuations

A continuation indicates how a trace through the AST blocks needs to continue once the current AST block has been processed. A **KStop** continuation term means that the trace must stop (i.e., the current AST block is the last one to be executed). A **KEndBlock** term denotes the exit of a loop. This means that the current AST block is part of the body of a loop and, once it and all of its nested AST blocks have been executed, all subsequent AST blocks in the AST are not part of the loop body. A **KSeq** term means that there is an AST block following the current one and then, there is some continuation after that.

A *final* AST configuration represents that the procedure has finished executing. An AST configuration is final if it consists of an AST block with an empty list of simple commands, no structured command and no transfer command, and a **KStop** continuation.

A small-step judgement of the form
$\mathcal{T}, \Lambda, \Gamma, \Omega, S \vdash \langle(b0, k0, s0)\rangle \rightarrow_{\mathsf{AST}} (b1, k1, s1)$ denotes a transition from the AST configuration $(b0, k0, s0)$ to the AST configuration $(b1, k1, s1)$ in the

context of $\mathcal{T}, \Lambda, \Gamma, \Omega, S$, where $\mathcal{T}, \Lambda, \Gamma$ and Ω are reused from [2]. $\mathcal{T}$ is a type interpretation map from abstract values to types. This is needed to accommodate uninterpreted types. Λ is a variable context, which is necessary to distinguish global from local variables. Γ is a function interpretation map, which is needed to interpret Boogie's functions. Ω is a type substitution map, which is needed to handle type parameters in procedures. S denotes the AST (i.e., the list of AST blocks). The judgement $\mathcal{T}, \Lambda, \Gamma, \Omega, T \vdash \langle(\mathit{b0}, \mathit{k0}, \mathit{s0})\rangle \rightarrow^*_{\mathsf{AST}} (\mathit{b1}, \mathit{k1}, \mathit{s1})$ denotes a sequence of such transitions (i.e., transitive closure). Figures 2.9, 2.10 and 2.11 illustrate our inference rules defining the AST configuration transition relation.

The inference rules use judgements of the form $\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle \mathit{cs}, \mathit{s0}\rangle\ [\rightarrow]\ \mathit{s1}$, denoting a reduction of a list of simple commands *cs* from state *s0* to *s1*, and judgements of the form $\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle \mathit{expr}, \mathsf{N}(\mathit{ns})\rangle \Downarrow \mathit{val}$, denoting evaluation of an expression in the normal state $\mathsf{N}(\mathit{ns})$, as well as the predicate **expr_all_sat** that takes as arguments a context and a list of expressions and returns true if all of the expressions in the list evaluate to true in the context. Both of the judgements and the predicate were defined as part of the previous Boogie proof generation work [2]. We also employ the auxiliary functions **is_final**, **list_to_cont** and **find_label** in the formalization. **is_final** returns true if an AST configuration is final. **list_to_cont** takes a list of AST blocks and a continuation k as arguments and converts the list into a corresponding **KSeq** continuation, which it prepends to k. **find_label** returns the point in the AST where a given label resides (we explain **find_label** in more detail below). Finally, we make use of Isabelle's concatenation operator @ and Isabelle's *cons* operator :: in the definitions of some of our rules. A term of the form *xs*@*ys* denotes that the list *ys* is appended to the list *xs*. A term of the form $x :: \mathit{xs}$ denotes a list, where x is the first element and *xs* is the tail of the list (i.e., a sublist consisting of all of the other elements in the list in order).

We build our formalization in such a way so as to first have a transition via a single step from a configuration C_1 to a configuration C_2, where C_1 and C_2 contain the same AST block and continuation with the distinction that, in C_2, the simple commands of the AST block have been reduced/processed. Afterwards we define rules that deal with the structured command or transfer command, which may both alter the continuation since both of these commands may contain breaks or gotos. If one ever reaches an AST block that contains no simple commands, no structured command and no transfer command, then one directly inspects the continuation in order to transition to the next AST configuration. In the following paragraphs we provide more details for some of the more notable rules.

The second of the simple rules in figure 2.9 formalizes the notion of an *incomplete* trace through the AST. This is a trace which models an execution of the procedure, which ends prematurely because either an `assume`-statement or an `assert`-statement fails. The rule states that one should immediately stop execution if a Magic or a Failure state has been reached. Precisely, it expresses that if the current AST configuration is not final, the current state is one of Failure or Magic and the current AST block has already had its simple commands reduced, then one should disregard the continuation, the structured and transfer commands and immediately transition into a final configuration, effectively ending the trace.

Regarding branching constructs, the Boogie language allows for if-clauses and while-loops to have no guard condition. In that case, it is decided non-deterministically whether the branch is taken or not. We reflect this in our rules for these constructs. For example, the first rule for if-statements in figure 2.10 formalizes taking the then-branch by having as a premise that, if there is a guard condition, the guard condition should hold true. If there isn't a guard condition, the premise is trivially satisfied and the rule is still applicable.

Let us now consider the rules regarding while-loops in figure 2.10. Firstly, we define a rule for "unwrapping" a **While** term. Recall our use of the **WhileWrapper** term from figure 2.6. When a loop is encountered for the first time in an AST configuration, it is in a "wrapped" state. That is, it has the **WhileWrapper** term applied to it. On encounter, the **WhileWrapper** is removed and instead, a **KEndBlock** term is applied to the existing continuation. Subsequently, on every following iteration through the loop, the "unwrapped" variant of the loop (i.e a **While** term encoding the loop) is added to the continuation so that the loop can be executed again. Should there happen to be a (potentially labelled) break statement in the body of the loop, then one must go through the continuation and look for **KEndBlock** terms. After one has gone through as many **KEndBlock** terms as the natural number parameter of the break indicates (recall breaks are "numbered" in our model), one transitions to a configuration with an *empty* AST block and a configuration equivalent to the rest of the continuation after the latest **KEndBlock** term that was discovered. An empty AST block is an AST block with no name, empty list of simple commands, no structured command and no transfer command.

After that, we define a rule for the case where an invariant of a loop does not hold true in some state s, in which the loop is reached. In order to formalize the failure of an arbitrary invariant, we do the following: we represent the list of invariants *invs* as a concatenation of three list - *invs1*, $[I]$ and *invs2*, where $[I]$ is a singleton list. We then require as a premise of the rule that all invariants in *invs1* hold true in s and the invariant I fails in s. Then, if the guard expression of the loop also holds true in s, the trace transitions into an ending state and stops execution. This formulation of the rule captures the possibility of failure of any arbitrary invariant in the list *invs* and ensures that the first invariant that fails will immediately end the trace.

Lastly, let us discuss the rule we define for goto-statements. If the trace is in an AST configuration, where the AST block has a goto statement as its transfer command, then the **find_label** function with the label of the goto as its argument is employed. Similar **find_label** function can be found in [5]. The **find_label** function starts iterating through the AST of the procedure from the beginning. Once it finds an AST block with a name that is equivalent to the label, it records the AST block and it computes the corresponding continuation of this AST block. The AST trace transitions to a configuration consisting of the newly found AST block, the computed continuation and the same state it was already in. Figures 2.12 and 2.13 illustrate our precise definition of the **find_label** function in Isabelle.

Simple rules

$$\frac{\begin{array}{c}\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle cs, \mathsf{N}(ns)\rangle \; [\rightarrow] \; s1 \quad cs \neq [\,] \\ b0 = \textbf{ASTBlock}\; name\; cs\; str\; tr \\ b1 = \textbf{ASTBlock}\; name\; [\,]\; str\; tr\end{array}}{\mathcal{T}, \Lambda, \Gamma, \Omega, S \vdash \langle (b0, k0, \mathsf{N}(ns))\rangle \;\; \rightarrow_{\mathsf{AST}} \;\; (b1, k0, s1)}$$

$$\frac{\begin{array}{c}(s1 = \mathsf{M}) \vee (s1 = \mathsf{F}) \quad \neg\textbf{is\_final}\,(b0,\; k0,\; s1) \\ b0 = \textbf{ASTBlock}\; name\; [\,]\; str\; tr \\ b1 = \textbf{ASTBlock}\; name\; [\,]\; None\; None\end{array}}{\mathcal{T}, \Lambda, \Gamma, \Omega, S \vdash \langle (b0, k0, s1)\rangle \;\; \rightarrow_{\mathsf{AST}} \;\; (b1, \textbf{KStop}, s1)}$$

$$\frac{\begin{array}{c}b0 = \textbf{ASTBlock}\; name\; [\,]\; None\; None \\ k0 = \textbf{KSeq}\; b1\; k1\end{array}}{\mathcal{T}, \Lambda, \Gamma, \Omega, S \vdash \langle (b0, k0, \mathsf{N}(ns))\rangle \;\; \rightarrow_{\mathsf{AST}} \;\; (b1, k1, \mathsf{N}(ns))}$$

$$\frac{\begin{array}{c}b0 = \textbf{ASTBlock}\; name\; [\,]\; None\; None \\ k0 = \textbf{KEndBlock}\; k1\end{array}}{\mathcal{T}, \Lambda, \Gamma, \Omega, S \vdash \langle (b0, k0, \mathsf{N}(ns))\rangle \;\; \rightarrow_{\mathsf{AST}} \;\; (b0, k1, \mathsf{N}(ns))}$$

Rule for Return-statements

$$\frac{\begin{array}{c}b0 = \textbf{ASTBlock}\; name\; [\,]\; None\; (Some\; (\textbf{Return})) \\ b1 = \textbf{ASTBlock}\; name\; [\,]\; None\; None\end{array}}{\mathcal{T}, \Lambda, \Gamma, \Omega, S \vdash \langle (b0, k0, \mathsf{N}(ns))\rangle \;\; \rightarrow_{\mathsf{AST}} \;\; (b1, \textbf{KStop}, \mathsf{N}(ns))}$$

Figure 2.9: simple inference rules defining transitions between AST configurations which reduce the simple commands, end the trace in the case of a execution with failing `assume`- and `assert`-commands or manipulate the remaining continuation. The symbol [ ] denotes the empty list.

Rules for If-Statements

$$\frac{\begin{array}{c}\forall b.\ \mathit{guard} = (\mathit{Some}\ b) \Rightarrow \mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle b, \mathit{ns}\rangle \Downarrow \textbf{True}\\ \mathit{b0} = \textbf{ASTBlock}\ \mathit{name}\ [\,]\ (\mathit{Some}\ (\textbf{If}\ \mathit{guard}\ (\mathit{then\_hd} :: \mathit{then\_blks})\ \mathit{else\_blks}))\ \mathit{None}\end{array}}{\mathcal{T}, \Lambda, \Gamma, \Omega, S \vdash \langle(\mathit{b0}, \mathit{k0}, \mathsf{N}(\mathit{ns}))\rangle \ \rightarrow_{\mathsf{AST}}\ (\mathit{then\_hd}, \textbf{list\_to\_cont}\ \mathit{then\_blks}\ \mathit{k0}, \mathsf{N}(\mathit{ns}))}$$

$$\frac{\begin{array}{c}\forall b.\ \mathit{guard} = (\mathit{Some}\ b) \Rightarrow \mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle b, \mathit{ns}\rangle \Downarrow \textbf{False}\\ \mathit{b0} = \textbf{ASTBlock}\ \mathit{name}\ [\,]\ (\mathit{Some}\ (\textbf{If}\ \mathit{guard}\ \mathit{then\_blks}\ (\mathit{else\_hd} :: \mathit{else\_blks})))\ \mathit{None}\end{array}}{\mathcal{T}, \Lambda, \Gamma, \Omega, S \vdash \langle(\mathit{b0}, \mathit{k0}, \mathsf{N}(\mathit{ns}))\rangle \ \rightarrow_{\mathsf{AST}}\ (\mathit{else\_hd}, \textbf{list\_to\_cont}\ \mathit{else\_blks}\ \mathit{k0}, \mathsf{N}(\mathit{ns}))}$$

Rules for While-Loops

$$\frac{\begin{array}{c}\mathit{b0} = \textbf{ASTBlock}\ \mathit{name}\ [\,]\ (\mathit{Some}\ (\textbf{WhileWrapper}\ \mathit{whilecmd}))\ \mathit{None}\\ \mathit{b1} = \textbf{ASTBlock}\ \mathit{name}\ [\,]\ (\mathit{Some}\ \mathit{whilecmd})\ \mathit{None}\end{array}}{\mathcal{T}, \Lambda, \Gamma, \Omega, S \vdash \langle(\mathit{b0}, \mathit{k0}, \mathsf{N}(\mathit{ns}))\rangle \ \rightarrow_{\mathsf{AST}}\ (\mathit{b1}, \textbf{KEndBlock}\ \mathit{k0}, \mathsf{N}(\mathit{ns}))}$$

$$\frac{\begin{array}{c}\forall b.\ \mathit{guard} = (\mathit{Some}\ b) \Rightarrow \mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle b, \mathit{ns}\rangle \Downarrow \textbf{True}\\ \mathit{invs} = \mathit{invs1}@[I]@\mathit{invs2} \quad \textbf{expr\_all\_sat}\ \mathcal{T}\ \Lambda\ \Gamma\ \Omega\ \mathit{ns}\ \mathit{invs1}\\ \mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle I, \mathit{ns}\rangle \Downarrow \textbf{False}\\ \mathit{b0} = \textbf{ASTBlock}\ \mathit{name}\ [\,]\ (\mathit{Some}\ (\textbf{While}\ \mathit{guard}\ \mathit{invs}\ (\mathit{body\_hd} :: \mathit{body\_blks})))\ \mathit{None}\\ \mathit{b1} = \textbf{ASTBlock}\ \mathit{name}\ [\,]\ \mathit{None}\ \mathit{None}\end{array}}{\mathcal{T}, \Lambda, \Gamma, \Omega, S \vdash \langle(\mathit{b0}, \mathit{k0}, \mathsf{N}(\mathit{ns}))\rangle \ \rightarrow_{\mathsf{AST}}\ (\mathit{b1}, \textbf{KStop}, \mathsf{F})}$$

$$\frac{\begin{array}{c}\forall b.\ \mathit{guard} = (\mathit{Some}\ b) \Rightarrow \mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle b, \mathit{ns}\rangle \Downarrow \textbf{True}\\ \textbf{expr\_all\_sat}\ \mathcal{T}\ \Lambda\ \Gamma\ \Omega\ \mathit{ns}\ \mathit{invs}\\ \mathit{b0} = \textbf{ASTBlock}\ \mathit{name}\ [\,]\ (\mathit{Some}\ (\textbf{While}\ \mathit{guard}\ \mathit{invs}\ (\mathit{body\_hd} :: \mathit{body\_blks})))\ \mathit{None}\end{array}}{\mathcal{T}, \Lambda, \Gamma, \Omega, S \vdash \langle(\mathit{b0}, \mathit{k0}, \mathsf{N}(\mathit{ns}))\rangle \ \rightarrow_{\mathsf{AST}}\ (\mathit{body\_hd}, \textbf{list\_to\_cont}\ (\mathit{body\_blks}@\mathit{b0})\ \mathit{k0}, \mathsf{N}(\mathit{ns}))}$$

$$\frac{\begin{array}{c}\forall b.\ \mathit{guard} = (\mathit{Some}\ b) \Rightarrow \mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle b, \mathit{ns}\rangle \Downarrow \textbf{False}\\ \textbf{expr\_all\_sat}\ \mathcal{T}\ \Lambda\ \Gamma\ \Omega\ \mathit{ns}\ \mathit{invs}\\ \mathit{b0} = \textbf{ASTBlock}\ \mathit{name}\ [\,]\ (\mathit{Some}\ (\textbf{While}\ \mathit{guard}\ \mathit{invs}\ \mathit{bodyastblocks}))\ \mathit{None}\\ \mathit{b1} = \textbf{ASTBlock}\ \mathit{name}\ [\,]\ \mathit{None}\ \mathit{None}\end{array}}{\mathcal{T}, \Lambda, \Gamma, \Omega, S \vdash \langle(\mathit{b0}, \mathit{k0}, \mathsf{N}(\mathit{ns}))\rangle \ \rightarrow_{\mathsf{AST}}\ (\mathit{b1}, \mathit{k0}, \mathsf{N}(\mathit{ns}))}$$

Figure 2.10: inference rules defining transitions between AST configurations regarding return-statements, if-statements and while-loops. The symbol [ ] denotes the empty list.

Rules for Breaks

$$\frac{\begin{array}{c} b0 = \textbf{ASTBlock}\ \mathit{name}\ [\,]\ (\mathit{Some}\ (\textbf{Break}\ 0))\ \mathit{None} \\ b1 = \textbf{ASTBlock}\ \mathit{name}\ [\,]\ \mathit{None}\ \mathit{None} \end{array}}{\mathcal{T}, \Lambda, \Gamma, \Omega, S \vdash \langle (b0, \textbf{KEndBlock}\ k0, \mathsf{N}(ns)) \rangle\ \rightarrow_{\mathsf{AST}}\ (b1, k0, \mathsf{N}(ns))}$$

$$\frac{b0 = \textbf{ASTBlock}\ \mathit{name}\ [\,]\ (\mathit{Some}\ (\textbf{Break}\ n))\ \mathit{None}}{\mathcal{T}, \Lambda, \Gamma, \Omega, S \vdash \langle (b0, \textbf{KSeq}\ b\ k0, \mathsf{N}(ns)) \rangle\ \rightarrow_{\mathsf{AST}}\ (b0, k0, \mathsf{N}(ns))}$$

$$\frac{\begin{array}{c} b0 = \textbf{ASTBlock}\ \mathit{name}\ [\,]\ (\mathit{Some}\ (\textbf{Break}\ n+1))\ \mathit{None} \\ b1 = \textbf{ASTBlock}\ \mathit{name}\ [\,]\ (\mathit{Some}\ (\textbf{Break}\ n))\ \mathit{None} \end{array}}{\mathcal{T}, \Lambda, \Gamma, \Omega, S \vdash \langle (b0, \textbf{KEndBlock}\ k0, \mathsf{N}(ns)) \rangle\ \rightarrow_{\mathsf{AST}}\ (b1, k0, \mathsf{N}(ns))}$$

Rule for Gotos

$$\frac{\begin{array}{c} b0 = \textbf{ASTBlock}\ \mathit{name}\ [\,]\ \mathit{None}\ (\mathit{Some}\ (\textbf{Goto}\ \mathit{label})) \\ (\textbf{find\_label}\ \mathit{label}\ S\ \textbf{KStop}) = (\mathit{Some}\ (\mathit{found\_astblock},\ \mathit{found\_cont})) \end{array}}{\mathcal{T}, \Lambda, \Gamma, \Omega, S \vdash \langle (b0, k0, \mathsf{N}(ns)) \rangle\ \rightarrow_{\mathsf{AST}}\ (\mathit{found\_astblock},\ \mathit{found\_cont},\ \mathsf{N}(ns))}$$

Figure 2.11: inference rules defining transitions between AST configurations regarding break-statements and goto-statements. The symbol [ ] denotes the empty list.

$$\textbf{find\_label}\,\mathit{lbl},\ [\,],\ k = \mathit{None}$$

$$\textbf{find\_label}\,\mathit{lbl},\ (\textbf{ASTBlock}\ n\ \mathit{cs}\ \mathit{None}\ \mathit{None}) :: [\,],\ k = \begin{cases} \mathit{Some}\ ((\textbf{ASTBlock}\ n\ \mathit{cs}\ \mathit{None}\ \mathit{None}),\ k) & \text{if } \mathit{lbl} = n \\ \mathit{None} & \text{otherwise} \end{cases}$$

$$\textbf{find\_label}\,\mathit{lbl},\ (\textbf{ASTBlock}\ n\ \mathit{cs}\ \mathit{None}\ \mathit{None}) :: \mathit{blks},\ k = \begin{cases} \mathit{Some}\ ((\textbf{ASTBlock}\ n\ \mathit{cs}\ \mathit{None}\ \mathit{None}),\ \textbf{list\_to\_cont}\ \mathit{blks}\ k) & \text{if } \mathit{lbl} = n \\ \textbf{find\_label}\ \mathit{lbl}\ \mathit{blks}\ k & \text{otherwise} \end{cases}$$

$$\textbf{find\_label}\,\mathit{lbl},\ (\textbf{ASTBlock}\ n\ \mathit{cs}\ (\mathit{Some}\ (\textbf{If}\ g\ \mathit{thn\_blks}\ \mathit{else\_blks}))\ \mathit{None}) :: \mathit{blks},\ k = \begin{cases} \mathit{Some}\ ((\textbf{ASTBlock}\ n\ \mathit{cs} & \\ \qquad (\mathit{Some}\ (\textbf{If}\ g\ \mathit{thn\_blks}\ \mathit{else\_blks}))\ \mathit{None}), & \\ \qquad\ \ \textbf{list\_to\_cont}\ \mathit{blks}\ k) & \text{if } \mathit{lbl} = n \\ \textbf{find\_label}\ \mathit{lbl}\ (\mathit{thn\_blks}@\mathit{blks})\ k & \text{if } (\mathit{lbl} \neq n\ \wedge \\ & \quad \textbf{find\_label}\ \mathit{lbl}\ \mathit{then\_blks}\ k \neq \mathit{None}) \\ \textbf{find\_label}\ \mathit{lbl}\ (\mathit{else\_blks}@\mathit{blks})\ k & \text{otherwise} \end{cases}$$

$$\textbf{find\_label}\,\mathit{lbl},\ (\textbf{ASTBlock}\ n\ \mathit{cs}\ (\mathit{Some}\ (\textbf{While}\ g\ \mathit{invs}\ \mathit{body\_blks}))\ \mathit{None}) :: \mathit{blks},\ k = \begin{cases} \mathit{Some}\ ((\textbf{ASTBlock}\ n\ \mathit{cs} & \\ \qquad (\mathit{Some}\ (\textbf{While}\ g\ \mathit{invs}\ \mathit{body\_blks}))\ \mathit{None}), & \\ \qquad\ \ \textbf{list\_to\_cont}\ \mathit{blks}\ k) & \text{if } \mathit{lbl} = n \\ \textbf{find\_label}\ \mathit{lbl}\ \mathit{body\_blks} & \\ \qquad (\textbf{list\_to\_cont}(\mathit{blks}@ & \\ \qquad\quad (\textbf{ASTBlock}\ \mathit{None}\ [\,] & \\ \qquad\qquad (\mathit{Some}\ (\textbf{While}\ g\ \mathit{invs}\ \mathit{body\_blks})) & \\ \qquad\qquad\quad \mathit{None}))\ k) & \text{if } (\mathit{lbl} \neq n\ \wedge \\ & \quad \textbf{find\_label}\ \mathit{lbl}\ \mathit{body\_blks}\ \mathit{cont} \neq \mathit{None}) \\ \textbf{find\_label}\ \mathit{lbl}\ \mathit{blks}\ k & \text{otherwise} \end{cases}$$

Figure 2.12: Isabelle definition of the **find_label** function, part 1. Meaning of symbols: *lbl* - label, [ ] - empty list, *k* - continuation, *n* - name, *cs* - simple commands list, *blks* - AST blocks, *thn_blks* - AST blocks in then-branch, *else_blks* - AST blocks in else-branch, *body_blks* - AST blocks in a loop body, *g* - guard condition, *invs* - list of loop invariants.

$$\textbf{find\_label}\ \mathit{lbl},\ (\textbf{ASTBlock}\ n\ cs\ (\mathit{Some}\ (\textbf{Break}\ \mathit{nat}))\ \mathit{None}) :: \mathit{blks},\ k =$$
$$\begin{cases} \mathit{Some}\ ((\textbf{ASTBlock}\ n\ cs\ (\mathit{Some}(\textbf{Break}\ \mathit{nat}))\ \mathit{None}), \textbf{list\_to\_cont}\ \mathit{blks}\ k) & \text{if } \mathit{lbl} = n \\ \textbf{find\_label}\ \mathit{lbl}\ \mathit{blks}\ k & \text{otherwise} \end{cases}$$

$$\textbf{find\_label}\ \mathit{lbl},\ (\textbf{ASTBlock}\ n\ cs\ (\mathit{Some}\ (\textbf{WhileWrapper}\ \mathit{loop}))\ \mathit{None}) :: \mathit{blks},\ k =$$
$$\textbf{find\_label}\ \mathit{lbl},\ (\textbf{ASTBlock}\ n\ cs\ (\mathit{Some}\ (\mathit{loop})\ \mathit{None}) :: \mathit{blks},\ k$$

$$\textbf{find\_label}\ \mathit{lbl},\ (\textbf{ASTBlock}\ n\ cs\ \mathit{None}\ (\mathit{Some}\ \mathit{tr})) :: \mathit{blks},\ k =$$
$$\begin{cases} \mathit{Some}\ ((\textbf{ASTBlock}\ n\ cs\ \mathit{None}\ (\mathit{Some}\ \mathit{tr})),\ \textbf{list\_to\_cont}\ \mathit{blks}\ k) & \text{if } \mathit{lbl} = n \\ \textbf{find\_label}\ \mathit{lbl}\ \mathit{blks}\ k & \text{otherwise} \end{cases}$$

$$\textbf{find\_label}\ \mathit{lbl},\ (\textbf{ASTBlock}\ n\ cs\ (\mathit{Some}\ \mathit{str})\ (\mathit{Some}\ \mathit{tr})) :: \mathit{blks},\ k = \mathit{None}$$

Figure 2.13: Isabelle definition of the **find_label** function, part 2. Meaning of symbols: *lbl* - label, n - name, *cs* - simple commands list, *nat* - natural number, k - continuation, *blks* - AST blocks.

Chapter 3

Proof Generation

In this chapter we present how we realize the generation of per-run certificates for the AST-to-CFG phase of Boogie. To begin with, we make clear what is the guarantee which a certificate provides, we describe our proof generation approach and we show an example. Next, we present and explain the formal theorems we formulate in their general forms, which when applied on a run of Boogie, constitute the overall certificate for the AST-to-CFG phase. Lastly, we discuss in more detail some important features of our implementation of the proof generation.

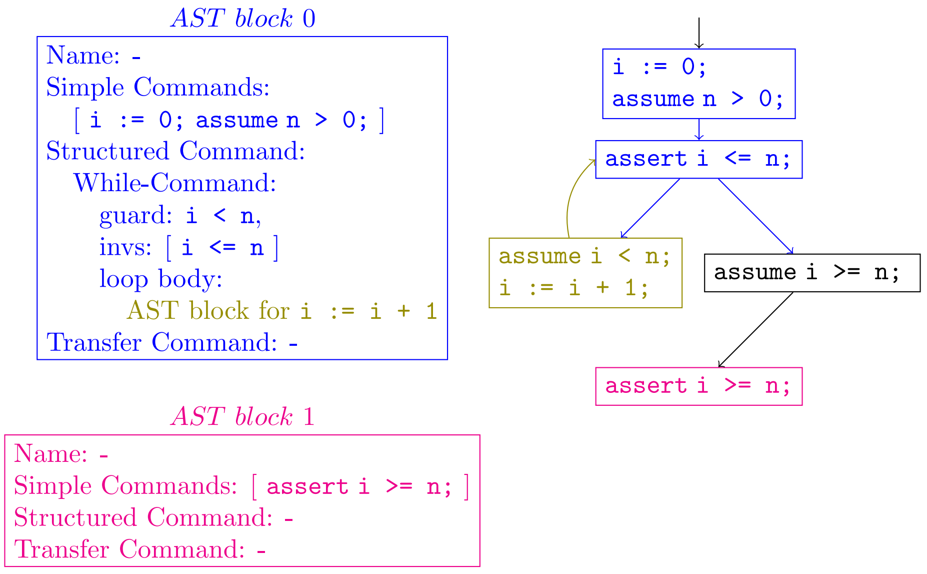
Let us now clarify the meaning of a certificate for the AST-to-CFG phase. Each procedure in Boogie has some formal specification which consists of preconditions, postconditions, loop invariants and assertions in the code. We say that the AST of a procedure (that is, the AST representation of the procedure body) is correct if no executions through the AST are *failing*. In other words, there must be no executions through the AST, which result in a failure of any postcondition, loop invariant or assertion.

Let us now define what a failing execution means formally. Recall from chapter 2 that an execution through the AST is expressed in our formalization via a sequence of AST configurations we call an AST trace. Recall also that an AST configuration is a triplet of an AST block, a continuation and a Boogie program state, and an AST configuration is final if it consists of an AST block with an empty list of simple commands, no structured command and no transfer command, and a **KStop** continuation. Within this context, a failing execution is an AST trace, which at some point transitions into an AST configuration that is not *valid*. An AST configuration is valid if:

1. Its state is not a failure state.
2. If it is final and its state is some normal state $\mathsf{N}(ns)$, then all postconditions in the specification of the procedure are true in $\mathsf{N}(ns)$.

With these definitions of AST correctness and a failing execution in mind, we say that the AST-to-CFG phase for a given procedure in Boogie is correct if it holds that, *if the CFG of a procedure body obtained via the AST-to-CFG phase from a corresponding AST S of the procedure body has no failing executions, then the AST S also has no failing executions.* A certificate for the AST-to-CFG phase for a procedure is a formal proof of this claim. The strategy we adopt towards constructing such a proof is to show that for every execution in the AST there is a corresponding execution in the CFG. So, if the corresponding execution in the CFG does not fail, then neither does the execution in the AST.

We now describe in more detail our approach towards generating a certificate. Recall that Boogie's AST is an ordered list of AST blocks, where an AST block could potentially contain other nested AST blocks in its structured command. In this setting, the concept of corresponding executions through the AST and CFG manifests itself in the idea of corresponding AST and CFG blocks. Figure 3.1 gives some high-level intuition about this. It shows the AST representation of our example program containing a while-loop on the left and its unoptimized CFG representation on the right. In the figure, we illustrate with colours which AST blocks informally coincide with which CFG blocks. AST block 0, coloured in blue, is translated into the blue-coloured CFG blocks on the right. Similarly, AST block 1 is translated into the magenta-coloured CFG block on the right, and the olive-coloured AST block nested in AST block 0 is translated into the olive-coloured CFG block on the right. For this particular example, Boogie also creates the black-coloured CFG block on the right, which does not have a matching AST block. Boogie always produces such CFG blocks at the end of while-loops, which serve as an ending point of the loop. These consist of a single `assume`-command, reflecting the assumption that the guard of the loop does not hold true and the execution should continue after the loop. Because of their simple structure, the lack of a matching AST block for these CFG blocks does not pose a significant challenge for generating AST-to-CFG certificates.


```
AST block 0
Name: -
Simple Commands:
  [ i := 0; assume n > 0; ]
Structured Command:
  While-Command:
    guard: i < n,
    invs: [ i <= n ]
    loop body:
      AST block for i := i + 1
Transfer Command: -

AST block 1
Name: -
Simple Commands: [ assert i >= n; ]
Structured Command: -
Transfer Command: -
```
```
i := 0;
assume n > 0;

assert i <= n;

assume i < n;
i := i + 1;

assume i >= n;

assert i >= n;
```

Figure 3.1: The AST and the CFG representation of a procedure with a while-loop

We utilize this notion of corresponding AST and CFG blocks to devise lemmas regarding the correctness of each AST block (in the list or nested). For each pairing of an AST block and a CFG block we produce a *local* and a *global* lemma:

- **Local lemma**: Assuming that no executions through a given CFG block are failing, then executing just the simple commands of a corresponding AST block starting from a given program state will result in a final program state, which is not a failure state F, and is the same as the one that would be the result of executing the simple commands of the CFG block. A local lemma is not generated if the AST block contains no simple commands.
- **Global lemma**: Assuming that no executions through the CFG starting at a given CFG block from some program state s and extending to the end of the CFG are failing executions, then, executing a given arbitrary AST trace t starting from an AST configuration t_{start} from the same program state s will result in a valid AST configuration. There are no restrictions on the trace t. It may execute more than the simple commands in the AST block in t_{start}. In particular, t could be a trace which extends to the end of the AST.

In order for a global lemma for a trace starting at some AST configuration to be proved, one needs to first prove global lemmas for traces starting at each successive AST configuration in the trace. Therefore, the overall proof is constructed bottom-up, from the end of the AST to the beginning. For instance, the first global lemma that is constructed involves the final AST block at the end of the AST and its corresponding CFG block. This first global lemma expresses a result about the final AST configuration transition in the AST trace (if an AST trace reaches this final AST block). Then, one by one, the proof generates global lemmas for AST blocks appearing earlier in the program until it finally generates a global lemma for the initial AST block at the beginning of the procedure. Such a global lemma covers an execution of the procedure in its entirety.

For example, in figure 3.1, global lemmas will be generated for each of the three colour pairings. A global lemma L_0 will first be generated for the magenta-coloured pairing. This pairing consists of blocks, which are final blocks in their respective representations of the procedure. Hence, L_0 can be proved without requiring global lemmas for block pairings regarding successor blocks, since there are no such pairings. Afterwards, global lemmas L_1 and L_2 will be generated for the olive-coloured pairing and for the blue-coloured pairing, respectively. The proof of L_2 will depend on both L_0 and L_1, since those are both global lemmas that concern pairings of AST and CFG blocks appearing after the blocks in the blue-coloured pairing. At this point, one complication regarding the blocks in the olive-coloured pairing arises. Since these olive-coloured blocks represent a body of a loop, where the loop as a whole is represented by the blocks in the blue-coloured pairing, the proof of L_1 will depend on L_2. Said in simpler terms, the proofs of L_1 and L_2 depend on each other because the blocks in blue are both predecessors and successors to the blocks in olive. This introduces a circular dependency. We resolve this issue by generating another lemma L_2^{loop}, which concerns only the parts of the blocks in the blue-coloured pairing regarding the loop, and we apply an inductive argument in the proof of L_2^{loop} on the number of steps left in the execution at the corresponding point in the procedure. L_1 then carries as an assumption an induction hypothesis. The proof of L_2 then only depends on L_2^{loop}. We explain this in more detail in sections 3.2.1 and 3.2.2.

In the following sections we present and explain the precise formulations of a local and a global lemma.

3.1 Local Lemma

The local lemma can be formally expressed as follows:

Theorem 1 (AST-to-CFG Local Lemma) *Let B be an AST block with a non-empty set of simple commands cs_0, and let its corresponding CFG block have the set of simple commands cs_1. Then,* if*:*

1. $\mathcal{T}, \Lambda, \Gamma, \Omega, T \vdash \langle (B0, k0, \mathsf{N}(ns)) \rangle \rightarrow_{AST} (B1, k1, s1)$
2. $\forall s_2.\, \mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle cs_1, \mathsf{N}(ns) \rangle\ [\rightarrow]\ s_2 \Longrightarrow s_2 \neq \mathsf{F}$

then: $s_1 \neq \mathsf{F}$ and if s_1 is some normal state $\mathsf{N}(ns')$, then it holds true that $\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle cs_1, \mathsf{N}(ns) \rangle\ [\rightarrow]\ \mathsf{N}(ns')$.

In premise 1 the symbol $\rightarrow_{AST}$ denotes a single-step transition between AST configurations and in premise 2 the symbol $[\rightarrow]$ denotes the reduction of a list of simple commands. The conclusion states that the single-step transition in the AST trace will not result in a failing state and if it results in a normal state, then it will be the same normal state that is reached after executing the simple commands in the CFG block from the same starting state. Note that in the conclusion of the lemma one does not need to formally reason about the possibility for an execution to enter a magic state. If s_1 is a magic state, then the conclusion trivially holds. This is because, if s_1 is a magic state, this means that there is a command `assume A` in cs_0 that is reached in the AST trace such that `A` does not hold. In such a case, any command that appears afterwards in the trace is ignored (intuitively, since a preceding assumption does not hold) and so the corresponding AST trace can never fail.

Syntactic relation between blocks. In order to prove a local lemma for a particular pairing of an AST and a CFG block, one needs to make concrete the abstract idea of the correspondence between the two blocks. In our implementation we do this by using an auxiliary *syntactic relation* between AST and CFG blocks. We say that an AST block and a CFG block are syntactically related if one of the following is true:

1. The simple commands in the AST block are exactly the same as the simple commands in the CFG block.

2. The simple commands in the CFG block include all of the commands in the AST block but there is an additional `assume`-command in the beginning of the CFG block. The reason for this is because, in creating the CFG blocks from the AST, Boogie removes branching constructs and inserts an `assume`-command in each branch indicating the condition that is assumed to have been satisfied in order for the branch to be taken. Refer back to figure 1.2, where this is shown for our example procedure that contains an if-statement.

3. The AST block has an empty list of simple commands and a While-Command as its structured command. The simple commands in the CFG block are `assert`-statements for the invariants in the While-Command.

We show that an AST block and its corresponding CFG block are syntactically related and use this fact to then derive the local lemma. Specifically, when generating a local lemma, we do so according to one of 4 different local lemma templates we define in Isabelle. There is one template that reflects case 1 above, one template that reflects case 3 and two templates which reflect case 2. Of these two, one carries the assumption that a guard has been satisfied and the other carries the assumption that a guard has failed.

Each local lemma for corresponding blocks serves as a building block for a global lemma for the same blocks.

3.2 Global Lemma

The global lemma can be formally expressed as follows:

Theorem 2 (AST-to-CFG Global Lemma) *Let B be an AST block with a set of simple commands, and let C be a corresponding CFG block. Then,* if*:*

1. $\mathcal{T}, \Lambda, \Gamma, \Omega, T \vdash \langle(B, \mathit{cont0}, \mathsf{N}(ns))\rangle \rightarrow^*_{\mathsf{AST}} (\mathit{EndB}, \mathit{EndCont}, \mathit{EndState})$
2. $\forall m_1 s_1.$
   $\mathcal{T}, \Lambda, \Gamma, \Omega, G \vdash (C, \mathsf{N}(ns)) \rightarrow^*_{\mathsf{CFG}} (m_1, s_1) \Longrightarrow$
   $((s_1 \neq \mathsf{F}) \wedge$
   $(\textbf{is\_final}\ (m_1, s_1) \Longrightarrow (\forall ns'.\ s_1 = \mathsf{N}(ns') \longrightarrow$
   $(\textbf{expr\_all\_sat}\ \mathcal{T}\ \Lambda\ \Gamma\ \Omega\ ns'\ \mathit{postconditions}))))$

then: $(\mathit{EndB}, \mathit{EndCont}, \mathit{EndState})$ *is a valid AST configuration.*

Point 1 states that there is some arbitrary AST trace from the configuration $(B, \mathit{cont0}, \mathsf{N}(ns))$ until some ending configuration $(\mathit{EndB}, \mathit{EndCont}, \mathit{EndState})$. Point 2 captures the assumption that the CFG has no failing executions.

A global lemma shows that executing the AST from the point in the program corresponding to $(B, \mathit{cont0}, \mathsf{N}(ns))$ cannot result in a failing state. This validates the correctness of the part of the procedure from that point onwards. Hence, if $(B, \mathit{cont0}, \mathsf{N}(ns))$ represents a starting configuration for the entire AST, then its global lemma validates correctness of the AST-to-CFG phase.

If B has a non-empty set of simple commands, we first prove a local lemma for B and C in order to prove the global lemma. Recall that the local lemma provides the guarantee that, assuming that executing the simple commands in C does not result in a failing state, one can also execute the simple commands in B without failing.

Additionally, in order to prove the global lemma, we first prove the global lemmas relevant for successive AST configurations. That is, our proof strategy for the global lemma is to first show that the given AST trace in premise 1 can be split into a first step and remaining steps, where the first step leads to a non-failing state. To obtain the conclusion we apply an already proved successor global lemma for the AST trace for the remaining steps. The AST trace for the remaining steps proves premise 1 of the successor global lemma. Premise 2 of the successor global lemma can be derived from premise 2 of the original lemma.

We derive the global lemma for a given pairing of an AST and a CFG block using one of 5 different formulations (i.e., templates in Isabelle) depending on the structure of the AST block or its position in the AST. We formulate a template for:

- An AST block with no structured command and no transfer command.
- A final AST block - An AST block that either has an encoding of a return-statement as its transfer command, or it is an AST block to be executed last in the AST.
- An AST block with an If-Command as its structured command.
- An AST block with a While-Command as its structured command.
- A special, artificially created AST block modelling a loop. We elaborate on this in the next subsection.

3.2.1 Loop Head AST Blocks

In this section we explain in detail the way we model AST blocks corresponding to parts of the source code with while-loops as it is different than our modelling for other blocks. Afterward, we clarify how we generate the corresponding local and global lemmas for loops.

Recall from figure 2.6 that in our Isabelle formalization of Boogie's syntax, a while-loop is encoded with a **While** term that we require to be subject to a **WhileWrapper** term application. Then, to reflect the fact that in Boogie's internal AST a loop (i.e., its encoding via a While-Command) must always be part of an AST block, we require in our formalization a **WhileWrapper** term to always be subject to an **ASTBlock** term application.

Rules for While-Loops

$$\frac{\begin{array}{c} \mathit{b0} = \textbf{ASTBlock}\ \mathit{name}\ [\,]\ (\mathit{Some}\ (\textbf{WhileWrapper}\ \mathit{whilecmd}))\ \mathit{None} \\ \mathit{b1} = \textbf{ASTBlock}\ \mathit{name}\ [\,]\ (\mathit{Some}\ \mathit{whilecmd})\ \mathit{None} \end{array}}{\mathcal{T}, \Lambda, \Gamma, \Omega, S \vdash \langle(\mathit{b0}, \mathit{k0}, \mathsf{N}(\mathit{ns}))\rangle \ \rightarrow_{\mathsf{AST}}\ (\mathit{b1}, \textbf{KEndBlock}\ \mathit{k0}, \mathsf{N}(\mathit{ns}))}$$

$$\frac{\begin{array}{c} \forall b.\ \mathit{guard} = (\mathit{Some}\ b) \Rightarrow \mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle b, \mathit{ns}\rangle \Downarrow \textbf{True} \\ \mathit{invs} = \mathit{invs1}@[I]@\mathit{invs2} \quad \textbf{expr\_all\_sat}\ \mathcal{T}\ \Lambda\ \Gamma\ \Omega\ \mathit{ns}\ \mathit{invs1} \\ \mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle I, \mathit{ns}\rangle \Downarrow \textbf{False} \\ \mathit{b0} = \textbf{ASTBlock}\ \mathit{name}\ [\,]\ (\mathit{Some}\ (\textbf{While}\ \mathit{guard}\ \mathit{invs}\ (\mathit{body\_hd} :: \mathit{body\_blks})))\ \mathit{None} \\ \mathit{b1} = \textbf{ASTBlock}\ \mathit{name}\ [\,]\ \mathit{None}\ \mathit{None} \end{array}}{\mathcal{T}, \Lambda, \Gamma, \Omega, S \vdash \langle(\mathit{b0}, \mathit{k0}, \mathsf{N}(\mathit{ns}))\rangle \ \rightarrow_{\mathsf{AST}}\ (\mathit{b1}, \textbf{KStop}, \mathsf{F})}$$

$$\frac{\begin{array}{c} \forall b.\ \mathit{guard} = (\mathit{Some}\ b) \Rightarrow \mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle b, \mathit{ns}\rangle \Downarrow \textbf{True} \\ \textbf{expr\_all\_sat}\ \mathcal{T}\ \Lambda\ \Gamma\ \Omega\ \mathit{ns}\ \mathit{invs} \\ \mathit{b0} = \textbf{ASTBlock}\ \mathit{name}\ [\,]\ (\mathit{Some}\ (\textbf{While}\ \mathit{guard}\ \mathit{invs}\ (\mathit{body\_hd} :: \mathit{body\_blks})))\ \mathit{None} \end{array}}{\mathcal{T}, \Lambda, \Gamma, \Omega, S \vdash \langle(\mathit{b0}, \mathit{k0}, \mathsf{N}(\mathit{ns}))\rangle \ \rightarrow_{\mathsf{AST}}\ (\mathit{body\_hd}, \textbf{list\_to\_cont}\ (\mathit{body\_blks}@\mathit{b0})\ \mathit{k0}, \mathsf{N}(\mathit{ns}))}$$

$$\frac{\begin{array}{c} \forall b.\ \mathit{guard} = (\mathit{Some}\ b) \Rightarrow \mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle b, \mathit{ns}\rangle \Downarrow \textbf{False} \\ \textbf{expr\_all\_sat}\ \mathcal{T}\ \Lambda\ \Gamma\ \Omega\ \mathit{ns}\ \mathit{invs} \\ \mathit{b0} = \textbf{ASTBlock}\ \mathit{name}\ [\,]\ (\mathit{Some}\ (\textbf{While}\ \mathit{guard}\ \mathit{invs}\ \mathit{bodyastblocks}))\ \mathit{None} \\ \mathit{b1} = \textbf{ASTBlock}\ \mathit{name}\ [\,]\ \mathit{None}\ \mathit{None} \end{array}}{\mathcal{T}, \Lambda, \Gamma, \Omega, S \vdash \langle(\mathit{b0}, \mathit{k0}, \mathsf{N}(\mathit{ns}))\rangle \ \rightarrow_{\mathsf{AST}}\ (\mathit{b1}, \mathit{k0}, \mathsf{N}(\mathit{ns}))}$$

Figure 3.2: inference rules defining transitions between AST configurations regarding while-loops.

Recall also our operational semantics rules concerning these constructs (illustrated in figure 3.2). As can be seen in the first rule in the figure, when a configuration with an AST block with no simple commands and a **WhileWrapper** is encountered, the **WhileWrapper** is removed. We call the resulting AST block with an “unwrapped” **While** term and without simple commands a *Loop Head AST block*. The trace continues by checking if the guard condition of the **While** term holds in the current program state and, if it does, the trace needs to model one or more iterations through the body of the loop. As indicated by the third rule, it does this by adding the Loop Head AST block to the continuation of the target AST configuration of the transition this rule models. The addition ensures that the Loop Head AST block will be encountered again later in the trace, thereby simulating iteration through the loop.

We now need to reconcile this formalization of loop iteration with the generation of global lemmas. We do this in the following way:

1. For each AST block in Boogie that contains a While-Command, we distinguish between two corresponding AST blocks - a *primer* AST block that reflects it closely by encoding its simple commands and its While-Command and "wrapping" it, and one Loop Head AST block with no simple commands and "unwrapped" **While** term. In a sense, the Loop Head AST block is a snapshot of the primer a few steps ahead in its processing.

2. We generate global lemmas for both of these AST blocks. The global lemma for the Loop Head AST block provides the validation guarantee for the loop execution, whereas the global lemma for the primer only shows that one can execute its simple commands without failing and then one can "unwrap" it.

Note that the global lemma for the primer is necessary. One cannot rely only on a local lemma for the primer because AST blocks in the AST which, for example, precede the primer are dependent on the primer AST block, meaning that, in proving their global lemmas, one requires the global lemma of the primer and not the global lemma of the Loop Head AST block. In contrast to all other AST blocks, a Loop Head AST block does not appear explicitly in the AST. It is neither a member of the list of AST blocks, nor is it nested in any other AST block. It also can never appear in a starting configuration of a full trace through the procedure - even in the continuation. It is simply a convenient stepping stone for constructing the global lemma of its primer.

One could also consider separating the primer and the Loop Head AST block entirely and then incorporating the Loop Head AST block into the AST as a direct successor of the primer. On the one hand, doing so would simplify the programming logic in Isabelle because it would make one of the templates for a global lemma redundant. On the other hand, it would introduce a larger gap between the AST formalization and Boogie's internal AST representation.

3.2.2 Generating Global Lemmas for Loop Head AST Blocks

Generating global lemmas for Loop Head AST blocks requires inductive reasoning, since one cannot know a priori how many iterations of the loop should be executed. Hence, in order to prove these global lemmas, we prove an adjusted global lemma that makes explicit the number of transitions j taken in the given input AST trace starting from a given starting configuration A_s until some ending configuration A_f and shows under further assumptions that A_f is valid. We prove this adjusted global lemma by induction on j. The induction hypothesis gives us that for any AST trace starting from the same Loop Head AST block transitioning to A_f in some number of steps j' with $j' < j$, we know that A_f is valid. That is, we assume that if we encounter the Loop Head AST block in the same configuration again, we assume that its execution will not lead to a failing state.

Global lemmas for any AST block B_{body} that is part of the body of the loop, i.e., is nested in the Loop Head AST block, need to carry the induction hypothesis as an assumption as well, because any trace that enters a configuration with B_{body} as its AST block, must have already encountered the Loop Head AST block in a previous configuration and it will encounter it again. Therefore, in proving the global lemma for B_{body}, one uses the induction hypothesis for the Loop Head AST block to discharge correctness for the part of the trace after that second encounter.

Importantly, assuming there are no breaks and gotos in a procedure, the only exit point of a loop is when the end of the loop body is reached. In particular, if an execution of the procedure is at a point inside of a loop that is nested n levels deep, then it is certain that it will exit each of the n loops one by one. As a result, to prove the global lemma for a block nested inside a loop one must only know about the induction hypothesis of that loop.

Said in terms of AST blocks, this means that a global lemma for an AST block needs to carry only the induction hypothesis of its immediate enclosing Loop Head AST block. It does not need to carry hypotheses for other outer Loop Head AST blocks it may be nested in.

3.3 Addition of Empty AST Blocks

A secondary important point regards the presence of *empty* AST blocks in the AST. We say an AST block is empty if it does not have a name, a structured command or a transfer command and its list of simple commands has no elements.

The only way such an AST block can be included in the AST during the process of creating the AST is if the procedure itself is empty (i.e., contains no program statements). However, there are cases where either the Boogie verifier adds empty AST blocks itself or cases where we add empty AST blocks to make the proof generation easier. In total, there are three cases where such empty AST block additions occur. The necessity for these additions in two of the cases in which it occurs (end of procedure, lack of else-branches), stems from our formalization and proof generation approach, while the third (target for breaks) was hardcoded in the verifier prior to this work:

- End of procedure: If the procedure ends with a loop and therefore its AST ends with an AST block that contains a While-Command, we add an empty AST block after it. This is because our proof generation approach requires an AST block that serves as a clear ending point of the procedure.
- Lack of else-branches: If an if-statement in the procedure has no else-branch, then Boogie leaves the else-branch in its If-Command encoding uninitialized. However, our Isabelle formalization of an If-Command requires AST blocks

for both the then- and else-branches, hence we change the If-Command encoding by initializing the else-branch to an empty AST block.

- Target for breaks: If the body of a loop ends with another loop, then an empty AST block is inserted at the end of the list of AST blocks corresponding to the body, so that if there happened to be a break statement in the body of the inner loop, there would be a clear target for it in the AST.

Only the last one of these cases (targets for breaks) has implications for the resulting unoptimized CFG after the AST-to-CFG phase. In this last case, the added empty AST block translates into an empty CFG block, which is optimized away after the AST-to-CFG phase.

3.4 Continuation Lemmas

As explained in previous sections, we define the terms **KEndBlock** and **KSeq** as part of the continuations. **KSeq** denotes a sequence of AST blocks that are to be executed after the AST block in the current configuration has been completed. **KEndBlock** marks a point in a sequence of continuations that separates the AST blocks that are to be executed as part of some corresponding loop body and the AST blocks that come after the loop.

The continuations, which these terms define, naturally engender rules in our operational semantics for transitions that cause a change in the continuations themselves. For example, the following rule defines a simple advancement in the sequence of AST blocks comprising the continuation:

$$\frac{\begin{array}{c} \mathit{b0} = \textbf{ASTBlock}\ \mathit{name}\ [\,]\ \mathit{None}\ \mathit{None} \\ \mathit{cont0} = \textbf{KSeq}\ \mathit{b1}\ \ \mathit{cont1} \end{array}}{\mathcal{T}, \Lambda, \Gamma, \Omega, T \vdash \langle(\mathit{b0}, \mathit{cont0}, \mathsf{N}(\mathit{ns}))\rangle\ \rightarrow_{\mathsf{AST}}\ (\mathit{b1}, \mathit{cont1}, \mathsf{N}(\mathit{ns}))}$$

Here is another example, where **KEndBlock** is simply skipped:

$$\frac{\begin{array}{c} \mathit{b0} = \textbf{ASTBlock}\ \mathit{name}\ [\,]\ \mathit{None}\ \mathit{None} \\ \mathit{cont0} = \textbf{KEndBlock}\ \mathit{cont1} \end{array}}{\mathcal{T}, \Lambda, \Gamma, \Omega, T \vdash \langle(\mathit{b0}, \mathit{cont0}, \mathsf{N}(\mathit{ns}))\rangle\ \rightarrow_{\mathsf{AST}}\ (\mathit{b0}, \mathit{cont1}, \mathsf{N}(\mathit{ns}))}$$

Such rules provide utility but they do not advance the execution of the trace through the AST. Nevertheless, given our modular proof generation approach, one needs to be able to generate global lemmas for traces starting in AST configurations, where only such rules can be applied. We refer to these lemmas as *continuation lemmas*.

Essentially, they show that if one could validate correctness for some trace starting in configuration A_s and ending in configuration A_f, then one could also validate correctness for a trace starting in configuration A'_s, which goes through A_s at some later point and ends again A_f but the only transitions that could be applied from A'_s to A_s are defined by such utility rules.

Chapter 4

Implementation and Evaluation

In this chapter we discuss the implementation of our proof generation as well as a qualitative and a quantitative evaluation of our work.

Implementation. We implement the proof generation for the AST-to-CFG phase of Boogie as an extension of the existing validation engine [2], which is implemented as a C# module compiled with Boogie. We instrument Boogie's core module in order to collect information about the AST, which allows us to generate the certificates for the phase. We add 2 new small files and a little more than 100 lines to Boogie's existing codebase. Certificates are generated automatically on each run of the verifier.

In terms of expressions and simple commands, the certificates, which the extended validation engine produces, support the same subset of the Boogie language as the original validation engine (that is, maps and bitvectors are not included) and in terms of control-flow program statements, our work covers if-statements, while-loops and return-statements. Break-statements and goto-statements are not supported yet.

Currently, our work successfully generates certificates for the AST-to-CFG phase for our supported Boogie language subset, in cases where the optimizations, which Boogie applies on the CFG after the AST-to-CFG phase, have no effect. Referring to figure 1.1, this means that G_2 and G_3 denote exactly the same program representation. In such cases, the certificate for the AST-to-CFG phase is connected with the certificate for the CFG-to-VC phase to produce a certificate for Boogie's pipeline from the AST to the VC. If the optimizations cause the CFG to change, the certificate generation for the AST-to-CFG phase fails, while the certificate generation for the CFG-to-VC phase remains unaffected. This failure is due to the fact we do not currently have a mechanism in our programming logic to concretely distinguish between the CFG before and after the optimizations. Adding such a mechanism is straightforward and part of future work. Once this mechanism is in place, then the generation of the AST-to-CFG certificates will no longer fail, i.e., the validation engine will be able to produce AST-to-CFG certificates even for programs in our supported Boogie subset, for which the optimizations do change the CFG. In such cases, one would not be able to directly link the AST-to-CFG to the CFG-to-VC certificate. To connect the two certificates, one would additionally need to produce an intermediate certificate that validates correctness of the optimizations. This is also part of future work.

Qualitative evaluation. For the evaluation of our work we use the same benchmark programs on which the original validation engine [2] was evaluated. These consist of 100 benchmark programs collected from Boogie's testsuite and a group of 10 external benchmark programs. Additionally, we applied our work on 36 simple Boogie programs we wrote, each consisting of a single procedure.

Of the 100 testsuite benchmark programs, for 78 of them we successfully generate certificates for the full pipeline of Boogie. Of the remaining 22 programs, 16 contain breaks or gotos, which causes the AST-to-CFG certificate to fail because our work does not support these features yet. One program contains an `assert false` statement. Such an assertion renders unreachable any following program statements and CFG blocks. One of the optimizations Boogie applies to the CFG after the AST-to-CFG phase augments the CFG to reflect this by transforming the `assert false` into a `return` and removing edges outgoing from the CFG block, which contains the assertion. This optimization causes the generation of the AST-to-CFG certificate to fail. As mentioned earlier, this will not be the case once programming logic to distinguish the unoptimized CFG from the optimized CFG has been added. For the last 5 programs in the testsuite, the generation of the CFG-to-VC certificate fails.

We also succesfully generate a certificate for 8 of the 10 external benchmark programs and for the 36 simple example programs. The certificates for the remaining 2 external benchmark programs fail due to breaks or gotos.

Quantitative evaluation. For the 8 external benchmark programs that are in our supported subset, we evaluate the certificates in terms of the time it takes for their validity to be checked in Isabelle. Table 4.1 shows the time it takes for Isabelle to check both the certificates for the AST-to-CFG and CFG-to-VC phases and the time it takes for Isabelle to check the certificate covering only the CFG-to-VC phase. For these programs, the table shows that coverage of the AST-to-CFG phase adds an average slow down of 6.55 sec.

Program	***#P***	***#LOC***	***T1***	***T2***
DivMod	2	69	27.89 sec	20.50 sec
Summax	1	23	21.16 sec	16.73 sec
MaxOfArray	1	22	23.94 sec	17.95 sec
SumOfArray	1	22	18.32 sec	16.21 sec
Plateau	1	50	23.91 sec	21.01 sec
WelfareCrook	1	52	45.09 sec	35.72 sec
ArrayPartitioning	2	57	32.38 sec	28.69 sec
DutchFlag	2	76	65.87 sec	49.33 sec

Table 4.1: Times needed for Isabelle to check validity of certificates per program. ***#P*** denotes the number of procedures in a program. ***#LOC*** denotes the number of lines of code in a program. ***T1*** denotes the time in seconds it takes for Isabelle to check a certificate covering the AST-to-CFG phase and the CFG-to-VC phase. ***T2*** denotes the time in seconds it takes for Isabelle to check a certificate covering only the CFG-to-VC phase. These times were produced on an HP Pavilion 15, i5-8250U 1.60GHz, Ubuntu 18.04.

Chapter 5

Conclusion

The main goal of this thesis was to extend the validation engine provided by Parthasarathy et.al. [2] with support for formal validation of the AST-to-CFG phase of the Boogie program verifier. To this end, we devised and mechanized in Isabelle a formalization of the syntax and semantics of Boogie's abstract syntax tree representation. We did so by building upon the existing mechanization in the validation engine, of Boogie's control-flow graph. We employed our AST formalization to realize the generation of per-run certificates for the AST-to-CFG phase of Boogie, applicable to Boogie programs that utilize a subset of the Boogie language. We extended the already existing instrumentation of the Boogie verifier to automate the generation of the certificates and we evaluated our work by applying it on Boogie's testsuite as well as examples from the literature and additional examples.

5.1 Future Work

Generating overall certificates covering Boogie's AST-to-VC pipeline

In order to achieve the overall goal of producing validation certificates for the correctness of Boogie's implementation from the AST to the VC, one needs to validate the sequence of basic optimizations Boogie applies to the CFG after the AST-to-CFG phase. This includes validating correctness for pruning of unreachable blocks, elimination of dead variables and coalescing of CFG blocks.

Generating certificates for programs containing breaks and gotos

Currently, even though breaks and gotos are included in our formalization of the AST syntax and semantics, the certificate generation for the AST-to-CFG phase fails if the input program contains breaks or gotos. This is because the mechanization of our proof generation approach in Isabelle does not handle them. A natural continuation of our work would be to extend the mechanization with support for these statements.

Bibliography

[1] K. R. M. Leino, “This is boogie 2.” June 2008.

[2] G. Parthasarathy, P. Müller, and A. J. Summers, “Formally validating a practical verification condition generator,” in *Computer Aided Verification (CAV)* (A. Silva and K. R. M. Leino, eds.), vol. 12760 of *LNCS*, pp. 704–727, Springer International Publishing, 2021.

[3] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*, vol. 2283. Springer Science & Business Media, 2002.

[4] A. W. Appel and S. Blazy, “Separation logic for small-step cminor,” in *Theorem Proving in Higher Order Logics (TPHOLs)* (K. Schneider and J. Brandt, eds.), (Berlin, Heidelberg), pp. 5–21, Springer Berlin Heidelberg, 2007.

[5] X. Leroy, “A formally verified compiler back-end,” *Journal of Automed Reasoning (JAR)*, vol. 43, no. 4, pp. 363–446, 2009.

[6] A. J. Summers, “The boogie intermediate verification language.” URL: https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Courses/SS2017/Program%20Verification/08-Boogie.pdf, 2017.

Appendix A

The Boogie Language

The Boogie language, as presented in [1] and [6], is an intermediate verification language. It is imperative and procedural. It supports built-in types for integers (`int`), booleans (`bool`), real numbers (`real`) as well as for maps and bitvectors. The language allows for declarations of custom types, global variables and procedures as well as axioms and functions. We do not discuss axioms and functions further. One could also declare local variables at the beginning of a procedure body.

The signature for a procedure in Boogie is illustrated in figure A.1. In the figure, `<T, ...>` denotes (optional) type parameters, `(x : T1,...)` denotes value parameters and `(y : T1',...)` denotes values returned to the caller. `requires` and `ensures` are keywords enforcing the specification of the procedure in the form of pre- and post-conditions, respectively.

```
procedure name <T, ...> (x : T1,...) returns (y : T1',...)
   requires ... ;
   ensures ... ;
{
   \\ procedure body
}
```

Figure A.1: Signature for a procedure in Boogie. The figure is adapted from [6].

In terms of constructs in a procedure body, Boogie supports features such as *simple* program statements (assignments, `assert`-statements, `assume`-statements, `havoc`-statements), break-statements, jumps, return-statements, if-statements, while-loops (with optional invariants) and call statements. A detailed description of all of these can be found in the Boogie reference manual [1]. We now give a brief overview of the simple program statements.

Assignments. An assignment in Boogie assigns a value to a variable. For example, the assignment `x := y + z` assigns the sum of the values of the variables `y` and `z` as value to the variable `x`. Boogie also allows for parallel assignments such as `x,y = z,w`, where the value of the variable `z` is assigned to the variable `x` and the value of the variable `w` is assigned to the variable `y`.

Havocs. A `havoc`-statement in Boogie assigns arbitrary values to a set of variables. For example, if the variables `x` and `y` are of type `int`, then the statement `havoc x, y` would assign to them arbitrary integer values.

Assertions. In Boogie, an `assert`-statement at a given program location gives an boolean expression that must evaluate to true at that location during any execution that reaches the location. Informally, it defines a check. For example, `assert x > 0` checks is the value of the variable `x` is bigger than 0.

Assumptions. In Boogie, an `assume`-statement at a given program point gives an expression that is assumed to be true at that point. For example, if there is an `assume x > 0` at some program point p, then the value of the variable `x` is assumed to be bigger than 0 at p. Informally, the goal of assumptions is to restrict the set of execution paths through a program. As explained thoroughly in [1, pg. 27-28], if an execution arrives at an assumption that fails, the path it took to reach the assumption is rendered "infeasible". To visualize this, consider the a `havoc`, followed by an assumption as in `havoc x; assume x > 0`. Here, the integer variable `x` may have been assigned any integer value by the `havoc`-statement but because of the `assume`-statement after it, executions, where the value assigned to `x` are smaller or equal to 0, are disregarded.

The control-flow statements which Boogie supports, namely break-statements, jumps, return-statements, if-statements, while-loops and call statements, are all defined analogously as in other imperative programming languages. One notable point concerns if-statements and while-loops in Boogie. The language allows if-statements and while-loops to have a *wildcard* expression as their condition, which reflects the possibility of a non-deterministic choice. An if-statement with a wildcard means that Boogie chooses non-deterministically, which branch of the if-statement to take. Similarly, a while-loop with a wildcard means that Boogie chooses non-deterministically, whether to execute the body of the loop.

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

FORMALLY VALIDATING THE AST-TO-CFG PHASE OF THE BOOGIE PROGRAM VERIFIER

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Hubanov	Aleksandar

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 23.08.2022	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.