Extending a Validator for a Verification Condition Generator

Bachelor's Project Description

Aleksandar Hubanov
Supervised by Gaurav Parthasarathy and Prof. Peter Müller

February 2022

Introduction and existing work

A program verifier's goal is to prove that the implementation of a given input program is correct with respect to its specification. In order for the verifier's results to be meaningful, it must be ensured that the underlying logic the verifier uses is sound and that the verifier's implementation contains no bugs. While it is common for the soundness of the logic to be formally proved, there are rarely any formal guarantees regarding the implementation of program verifiers.

Parthasarathy et al. [3] address this fundamental issue for the widely-used Boogie verifier [2]. The Boogie verifier is a verification condition generator that takes a Boogie program as input and produces a first-order formula called a verification condition (VC) for each procedure in that program. Boogie uses an external SMT solver to check if the generated VCs are valid. If the SMT solver reports that the VC for a procedure is valid, then Boogie reports that the procedure is correct. In [3], through a formalization of a subset of the Boogie language in the interactive theorem prover Isabelle and a light-weight instrumentation of the existing Boogie verifier, a novel method has been devised to produce a formal proof on every successful run of the Boogie verifier, which proves that if the generated VC is valid, then its corresponding procedure is indeed correct. I will refer to this proof-generating instrumentation of Boogie as 'the validator' for the rest of this document.

Boogie applies a sequence of transformations to the input program before generating the VC. Overall, there are four major phases. First, Boogie transforms the input program represented as an abstract syntax tree (AST) into a control-flow graph (CFG). Then, Boogie eliminates loops in the CFG, transforming it into a directed acyclic graph (DAG). Next, Boogie *passifies* the DAG by versioning variables and replacing assignments by corresponding assume statements. Finally, Boogie generates a VC from the passified DAG.

The validator employs a modular approach to certify the correctness of a significant part of this process. It generates formal proofs in Isabelle for the correctness of each of the last three phases, which it then connects together to form an overall proof that a valid VC implies a correct procedure in CFG form.

While the current version of the validator captures an important part of Boogie, there are still parts of the verifier's implementation for which no correctness certificates are generated and parts for which the existing proof generation techniques have limitations. The goal of this project is to address some of these limitations by extending and modifying the current validator.

Open challenges

In this section some of the more significant open challenges in the existing validator are concretely outlined.

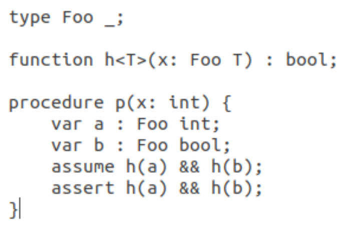
Validation of the AST-to-CFG transformation and the subsequent optimizations

In the first phase of the pipeline, the source program, given in an AST form, is transformed into a CFG. Then, before proceeding with the CFG-to-DAG phase, some basic transformations to the CFG are applied, most notably dead variable elimination, desugaring of procedure calls, block coalescing and pruning of unreachable blocks. As of yet, the validator does not produce certificates for this part of the verification engine.

Validation of Boogie's monomorphization feature

In the VC generation phase, Boogie uses a *monomorphization* technique in certain cases to deal with polymorphic functions and type constructors. Monomorphization is a transformation of the Boogie program, where a polymorphic function with a generic type parameter is replaced by multiple monomorphic functions - one for each relevant type instantiation of the parameter. Additionally, type constructors with one or more arguments are replaced by type constructors without arguments. Monomorphization circumvents the need to represent Boogie types explicitly as terms in the VC, which is what Boogie does for more complex examples where monomorphization is challenging. Support for validating monomorphization remains to be added.

Figure 1 illustrates a simple example of monomorphization, where Figure 1 (a) shows the original program and Figure 1 (b) shows the monomorphized program. The local variables 'a' and 'b' in the original procedure 'p' are of types 'Foo int' and 'Foo bool', respectively. The unary type constructor 'Foo' in (a) is replaced by the nullary type constructors 'FooInt' and 'FooBool' in (b) and the polymorphic function 'h' in (a) is replaced by the monomorphic functions

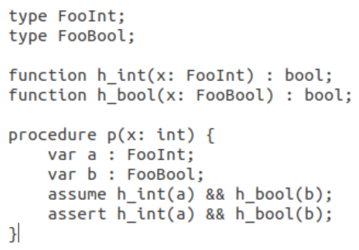


```
type Foo _;

function h<T>(x: Foo T) : bool;

procedure p(x: int) {
    var a : Foo int;
    var b : Foo bool;
    assume h(a) && h(b);
    assert h(a) && h(b);
}
```

(a) Original Program



```
type FooInt;
type FooBool;

function h_int(x: FooInt) : bool;
function h_bool(x: FooBool) : bool;

procedure p(x: int) {
    var a : FooInt;
    var b : FooBool;
    assume h_int(a) && h_bool(b);
    assert h_int(a) && h_bool(b);
}
```

(b) Monomorphized Program

Figure 1: Monomorphization

'h_int' and 'h_bool' in (b).

Improved proof generation for the VC phase

The existing work uses a coarse-grained approach that relies on Isabelle's automation to certify the translation of Boogie function calls in the VC phase. While this technique is successful in many cases, there are some instances where the technique fails even though the translation is correct.

One class of instances for which the technique does not work is due to type coercions that Boogie introduces as part of the translation of function calls. The reason Boogie introduces type coercions is because for built-in types such as the integers the VC contains two representations of the same type.

Core Goals

AST-to-CFG phase

The main goal of the project is to extend the validator to handle the AST-to-CFG transformation phase and the subsequent basic program optimizations before the CFG-to-DAG phase. It will be realized through the following steps:

- Understand Boogie's implementation of the translation from AST to CFG.
- Decide which elements of the Boogie AST should be supported by the validator. If-statements and while-loops are to be supported but it remains to be decided if more complex statements such as goto-statements and break-statements will also be supported.
- Formalize the AST and a corresponding semantics in Isabelle building on the already existing CFG semantics.

- Design and implement a validation approach for the AST-to-CFG transformation ignoring optimizations on the CFG. There are at least two possible ways one could approach this:
    1. As is done for the other phases, one could generate an Isabelle proof for each successful run of the verifier.
    2. Alternatively, one could implement a validator algorithm in Isabelle, which takes as input the AST and the CFG and returns true only if the correctness of the CFG implies the correctness of the AST. If this approach is chosen, the algorithm can be extracted to an efficient programming language, potentially making the validation faster. This would mean, however, that the extraction will have to be trusted.
- Extend the validation approach to handle dead variable elimination, block coalescing and pruning of unreachable blocks.

VC generation phase

The second goal of the project is to tackle one of the open challenges in the VC phase. To what extent this second goal will be explored will depend on the complexity of the AST-to-CFG validation. The goal includes the following:

- Extend the validator to produce correctness certificates for monomorphization.
- Understand why the validation of function calls in the VC phase fails in some cases and design and implement an approach to optimize the current validation of function calls.

Evaluation

Evaluate the quality of all newly implemented functionalities.

Extension Goals

- Extend the validator with a semantic model for currently unsupported features of Boogie such as bit vectors.
- Validate that the VC generated by Boogie is indeed valid using existing work on validating results produced by SMT solvers in Isabelle [1].

References

[1] Sascha Böhme and Tjark Weber. "Fast LCF-Style Proof Reconstruction for Z3". In: *Interactive Theorem Proving (ITP)*. 2010.

[2] K. Rustan M. Leino. “This is Boogie 2”. Available from `http://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf`.

[3] Gaurav Parthasarathy, Peter Müller, and Alexander J. Summers. “Formally Validating a Practical Verification Condition Generator”. In: *Computer Aided Verification (CAV)*. Ed. by Alexandra Silva and K. Rustan M. Leino. Cham: Springer International Publishing, 2021, pp. 704–727.