CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Incremental Deductive Verification for a subset of the Boogie language

Master's thesis in Computer Science, algorithms, languages and logic

LEO ANTTILA
MATTIAS ÅKESSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Master's thesis 2017

Incremental Deductive Verification for a subset of the Boogie language

LEO ANTTILA
MATTIAS ÅKESSON

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

Incremental Deductive Verification for a subset of the Boogie language
LEO ANTTILA
MATTIAS ÅKESSON

Supervisor: Carlo A. Furia, Department of Computer Science and Engineering
Examiner: John Hughes, Department of Computer Science and Engineering

Master's Thesis 2017
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2017

Incremental Deductive Verification for a subset of the Boogie language
LEO ANTTILA
MATTIAS ÅKESSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

As computer programs and systems get larger and more complex, the conventional method of ensuring system correctness by feeding it input data and analyzing the output becomes harder and more time consuming, since the space of possible input data becomes nearly infinite. Formal verification is a method of proving the correctness of a software or hardware system in accordance to a formal specification of its intended behavior, proposed as a complement to regular testing to increase the accuracy of detection of bugs in a system. An issue with formal verification today is its scalability. The SMT-problem is known to be at least NP-hard, thus the time required to verify a program increases rapidly as programs get larger.

This thesis presents the design, implementation and evaluation of an incremental deductive verifier for a subset of the intermediate verification language Boogie. The technique is intended to speed up the verification of a program when moving between different iterations by identifying and re-verifying only those parts of the program that are affected by the modifications made since the last verification. This reduces the size of the input to the SMT-solver and can therefore help mitigate the issue of scalability in the verification process. The verifier is evaluated by running it on a set of test programs, each with multiple versions to simulate realistic software development, with incrementality turned on or off. The results show promise for the technique with the majority of tests showing considerable time save with up to 15-49% time saved for most programs with three iterations when solved incrementally compared to non-incrementally, and increased number of iterations generally lead to further time savings.

Keywords: Computer science, formal verification, deductive verification, boogie, incremental verification, verification, sidecar

Acknowledgements

We would like to thank our supervisor Carlo A. Furia for his assistance and guidance during the project. We would also like to thank Antonio Filieri for his assistance and allowing us to make us of the SiDECAR project, which has been key to the success of our project.

Leo Anttila, Mattias Åkesson, Gothenburg, June 2017

Contents

1 Introduction

Formal verification is the process of proving or disproving the correctness of a software or hardware system in accordance to some formal specification of its intended behavior. There are two main approaches to this: model checking and deductive verification. In model checking a model of the system to be tested is produced, for example an automaton. The properties to be verified are specified through logic formulas such as linear temporal logic and with those properties we can perform exhaustive search on the model to look for states and transitions which might break them. In deductive reasoning we instead generate a set of verification conditions, i.e. mathematical formulas, which if proved establish that a component fulfills its specification. This can be done for example by producing logic formulas and feeding them to a satisfiability modulo theory (SMT) solver.

Today formal verification is mainly used in the hardware industry, with the software industry lagging behind, although its use there is growing. It is not hard to imagine why this gap came to exist: a hardware fault is difficult to correct after a component has been produced, thus leading to potential mass recalls, significant economic damage to the producers and inconvenience to its users. Software can in most cases be patched to fix bugs or security issues as they are discovered. Therefore any fault, as long as it does not put people or property in danger, is less severe.

One obstacle to the deductive verification of software is its algorithmic complexity. The SMT-problem is a more general version of the SAT-problem which in turn is known to be NP-complete. Progress in heuristics and an increase in raw computational power has allowed us to solve increasingly larger instances but the execution time can still make it infeasible to use for larger, more complex projects.

The goal of the project is to design and implement *incremental* generation and validation of verification conditions for the verification language and tool Boogie[1]. With the current modular design of Boogie, when a procedure's body is changed even slightly, all the verification conditions determined by the procedure are re-verified [16]. By contrast if only the verification conditions that are affected by the change were regenerated then only those verification conditions would need to be re-verified. Then small changes in a program would no longer yield full re-verifications, making the modularity more fine-grained. This could reduce the average running time of

[1]https://github.com/boogie-org/boogie

verification, thus contributing to the feasibility of deductive verification for software systems.

In this introductory chapter the goal of the project is presented along with context for the work and the limitations imposed to keep the project in scope. Chapter 2 contains a brief description, along with some examples, of the major tools and environments used to develop the verifier. In chapter 3 we present the design and implementation of the verifier, the obstacles encountered during development and the solutions chosen to overcome them. Chapter 4 describes how the verifier was tested, including a brief description of the test cases, and a presentation of the results. In chapter 5 we discuss the results and implementation of the verifier. Finally, chapter 6 closes with a conclusion on the project as a whole, and some ideas for future work within the area.

1.1 Context

The project targets Boogie, which denotes both a language and a tool for verification of that language. The Boogie language is an intermediate language used to generate verification conditions. It was developed to target imperative languages and is therefore especially efficient to that purpose. This means that to efficiently verify any program written in an imperative language it is sufficient to convert its code into Boogie [17], a less extensive undertaking compared to directly generating the verification conditions. Since Boogie can be used for all imperative languages, successfully implementing incremental verification can improve the verification process of all those languages.

Incremental verification can be built on top of incremental parsing, a currently active research field [8]. Incremental parsers only regenerate parts of the syntax tree that are affected by the code that has been changed since the previous parsing. By only regenerating the verification conditions for the same parts of the syntax tree and by reusing the already solved and unaffected verification conditions an incremental verification scheme can be created.

Boogie does not currently support incremental verification; however, it does support modular verification. The effects of a procedure call are limited to what is specified in the caller's specification (pre- and postcondition). Thus, each procedure's body is verified against its specification, but a caller's code does not require re-verification as long as the specification of the procedures it calls do not change [16]. Even with this type of modularity it can take a long time to verify a program when a procedure has been changed [18]. Being able to just re-verify a minimal part of the procedure that has been changed would be a big improvement to the already supported modular verification in Boogie, making verification less time-consuming and thus making developing verified software significantly easier.

There have been some previous implementations of incremental verification. Bianculli et al. made an incremental verifier for a fragment of the C language, KernelC, and the results look promising [8]. In comparison with the state-of-the-art verifier MatchC [23] their verifier outperformed MatchC in almost all cases, hinting toward big potentials for future development in the field.

1.2 Goals and Challenges

The goal of this thesis project is to design, implement, test and evaluate an incremental verifier for a suitable subset of the Boogie language. If a speed-up, with respect to non-incremental verification, is achieved this can stand as a proof of concept to support further development of incremental verification for Boogie. This could contribute to the applicability of deductive verification so that its use can become more widespread in the future.

An appropriate breakdown into sub-goals is:

1. Research and investigate the theory of incremental parsing and incremental verification.
2. Design an incremental verification system for a suitable subset of the Boogie language.
3. Implement the incremental verification technique in a fitting tool.
4. Perform experimental testing and evaluation with regards to the speed of the incremental verifier.

The main challenge of the project, making up its core, is the design of the incremental verifier. This includes determining what method to use for logically structuring program configurations and their contracts, as well as how to perform syntax driven reachability checking. A key aspect of the design is constructing the generation of verification conditions such that it minimizes the amount that are regenerated in case of local changes. To increase efficiency it is also important to take full advantage of the stateful SMT-solver interface used to pass the verification conditions.

1.3 Limitations

The main goal of the project is to investigate the viability and potential gain from incremental verification, and as such only a subset of the Boogie language is targeted. Since Boogie is a well-developed language with some complex structures and features, it would not be possible to implement the entire language within the time frame of this project.

The main features omitted from the Boogie language:

- Advanced types - Only boolean and integer types are supported
- Bitvectors
- Labels and gotos
- Procedures that returns multiple values
- Maps and Arrays
- Existential and universal quantifiers

Additionally, overloading of variables names will not be supported. For instance, if the global variable `x` exists, the variable name `x` can not be re-used as a parameter for a procedure.

Very little typechecking exists in the verifier. Essentially this prototype will assume the user is writing syntactically correct code that just needs to be verified. As a consequence of this, writing syntactically incorrect code can cause crashes, and tracing the fault can be difficult.

2

Tools and Environments

In this chapter an overview of the tools, structures and environments used in the thesis work is presented. In section 2.1 we describe Boogie, the verification environment whose language is targeted by the verifier developed in this project. Section 2.2 describes SiDECAR, a tool developed as a framework for incremental verification and used in this project to create an incremental parser for the language. Finally section 2.3 gives a brief description of SMT-solvers and the specific solver used in this project: Z3.

2.1 Boogie

As discussed in section 1.1, Boogie denotes both a programming language and a tool for verification of that language, developed by the RiSE team at Microsoft Research [2]. Together they are intended to function as a layer on top of which program verifiers for other languages can be developed.

The task of verifying a modern programming language is complex, but by separating the task into two parts the work required can be greatly reduced [17]. The first step is converting the program and its proof obligations into an intermediate language. The second step is to transform the program from the intermediate language into verification conditions and feed those to an SMT-solver. Differences in semantics between different source languages are handled by encoding the behavior with primitive constructs, for example by recording any properties guaranteed during an execution of a program as assumptions.

Boogie serves as both the intermediate language and the tool that transforms the intermediate program representation into verification conditions and checks their validity. As such, only the first step has to be redone for different program languages, saving a lot of work. Translations into the boogie language has been done, at least partly, for a number of languages: Spec# [4], C [9] [22], Dafny [19], Eiffel [24] and Jimple (an intermediate representation of Java) [1].

```
var counter : int;
procedure incrementCounter()
  requires counter >= 0;
  ensures counter == old(counter) + 1;
  modifies counter;
{
  counter := counter + 1;
}
```

Figure 2.1: Simple Boogie program incrementing a counter by one

2.1.1 The Boogie Language

The Boogie language is a procedural intermediate verification language. It allows the formal specification of a program through certain language constructs, a set of verification-specific statements and native support for a typed first-order logic. These are used to generate the verification conditions required to prove or disprove the correctness of the program.

Formal specification of the behavior of a procedure is constructed by associating it to a collection of preconditions, postconditions and mutable variables, called a contract. The conditions are expressed in first-order logic formulas and the modifiable variables are declared by their identifier. The precondition must hold at a procedure call, the postcondition must hold after the called procedure has been executed and only variables declared as modifiable may be changed during its execution. This is an extension of the design by contract method [20], introduced by Bertrand Meyer in the Eiffel programming language 1986, based on earlier work such as the pre/postcondition technique and invariable reasoning originating from work by Tony Hoare [13][14].

Figure 2.1 shows an example of a procedure which increments the value of a counter by one. The counter is defined to be inactive if its value is less than zero, thus the `requires` clause establishes the precondition that the counter must be active for the procedure to be legally run. The `ensures` clause establishes the postcondition that the value of the counter after the execution of the procedure should be equal to the value of the counter at the start of the procedure plus one. Finally the `modifies` clause establishes that the procedure may only modify the value of the `counter` variable.

In comparison to other language constructs, the verification of loops is more complex. Complications arise from the fact there is no easy way to know, except in trivial cases, how many iterations of a loop will be executed. Every iteration changes the program state and consequently the postcondition of the loop. Therefore, the verifier has no choice but to execute the loop until termination, called loop unrolling, a very inefficient and often impossible feat. To solve this Boogie supports the use of loop invariants, a method for reasoning about the behavior of loops, introduced, as previously mentioned, in 1969 by Tony Hoare [14]. A loop invariant is a first-order logic formula which must hold at the start and end of each iteration of the loop,

```
while (x < y) invariant x <= y; {
  x := x + 1;
}
```

Figure 2.2: Simple loop incrementing x until it is equal to y

including when the loop is first reached and when it is terminated. The goal is to construct the invariant such that it, together with the loop exit condition, implies the postcondition of the loop. Then it is sufficient to prove that the invariant holds for the loop and let the effects of the loop be determined by the implied postcondition.

Figure 2.2 shows an example of a simple while-loop that increments the variable x until its value equals the value of y. As long as x is smaller than y when the loop is reached the invariant will hold, since x will at most be equal to y (at loop termination). By combining the negation of the loop condition and the invariant the effects on x can be inferred: $\neg(x < y) \land x <= y \implies x == y$.

Quantifiers raise another complication for the verifier, as most possible instantiations of them will not bring us closer to proving or disproving the formula. Many SMT solvers can infer the possible instantiations by analyzing the body of the quantifier but as of yet this is often crude and inefficient. It is therefore better to use user-defined triggers to more accurately limit the space of possible instantiation. To this end Boogie supports the addition of triggers when coding quantifiers which can be passed directly on to the SMT-solver.

In addition to the above, Boogie supports the following language features:

- Where-clauses, applied to variables, which are first-order logic formulas used to define the allowed scope for the variable.
- Axioms, which are first-order logic formulas that must hold at any point in the program.
- Assert statements, which checks that an input formula holds at the execution of the statement.
- Assume statements, which holds the input formula as true for the rest of that procedure.
- Havoc statements, which assigns a non-deterministic value to one or more variables such that the assignment satisfies all program axioms and where-clauses if possible.

Finally, Boogie allows non-deterministic execution of condition guarded statements (if-statements and while-loops). For if-statements this constitutes randomly choosing which branch to execute, or randomly choosing to enter the if-statement body should no else branch be provided, for while-loops it executes the loop body a non-deterministic number of times.

2.1.2 The Boogie Verification Tool

The Boogie verification tool is the default verifier for the Boogie language; it generates verification conditions from some input code and feeds these to an SMT-solver, Z3 by default, in order to verify the correctness of the program. Different techniques are employed to increase the efficiency of verification, some of which are presented here.

Instead of generating one big verification condition for the whole program, Boogie creates verification conditions for each procedure. Each procedure's body is independently verified against its contract, creating a collection of verification conditions, the conjunction of which represents the entire program. The main benefit of this is that it supports modular verification: when a procedure call is performed the effects of that procedure are limited to the specification in its contract. Thus, a callee of a procedure whose body has been changed does not require re-verification unless its contract has been modified as well.

Had Boogie not used the modular verification method, and instead created a single verification condition for a program, it would require the inlining of all procedure calls. This would blow up the input size for the verification immensely, making deductive verification infeasible. As such, this method is adopted by practically all deductive verifiers today. This could lead to programs being very large if there are a lot of procedure calls, or possibly infinite for certain recursive designs. Generally, since the SMT-problem is NP-Hard, it is beneficial to keep the input size small.

2.2 SiDECAR

SiDECAR is a framework developed by Bianculli et al. [7] which provides a platform for incremental verification. It builds an incremental parser from an Operator Precedence Grammar (OPG) [12] representation of a language, on top of which incremental verification can be built. Between different versions of a program the parser identifies which part(s) of the abstract syntax tree (AST) need to be updated, the minimal context, and only revisits the nodes that belong to those parts. Attribute grammars [15], specifically synthesized attributes, are used to allow each node of the AST knowledge, not only of its own semantics, but also those of its children. By encoding the verification condition generation and verification process into these attributes, the incrementality of the parser can be used for the verification itself.

2.2.1 Operator-Precedence Grammar

A context-free grammar (CFG) consists of four elements: a finite set of terminal symbols, a finite set of non-terminal symbols, a set of productions defining non-terminals

```
Expr ::= EMul
       | EAdd
EAdd ::= EAdd '+' EMul
       | EMul '+' EMul
EMul ::= EMul '*' 'n'
       | 'n'
```

Figure 2.3: OPG for a simple arithmetic expression

```
value(Expr)   ::= value(EMul)
                |  value(EAdd)
value(EAdd₀)  ::= value(EAdd₁) + value(EMul)
                |  value(EMul₀) + value(EMul₁)
value(EMul₀)  ::= value(EMul₁) * value('n')
                |  value('n')
```

Figure 2.4: Attribute grammar for a simple arithmetic expression

as a combination of terminal and non-terminal symbols and finally a starting symbol defining the root of the grammar. The sets of non-terminal and terminal symbols must be disjoint. Producing a CFG is relatively easy; however CFGs make no statements pertaining to the precedence relations between terminal symbols, making the task of parsing a language whose grammar is defined as a CFG more arduous.

Operator precedence grammars (OPG), first introduced by R.W. Floyd in 1963 [12], are a subset of CFGs obtained by applying stricter rules to how the grammar is constructed, specifically all productions must be in operator form. A production is in operator form if the right hand side is nonempty and has no adjacent non-terminals. These restrictions make it possible to define binary precedence relations between terminals; given two terminals one can yield precedence to the other, take precedence over the other or they can have equal precedence. The precedence relations can be calculated in an automatic fashion, reducing the ambiguities that need to be handled when designing the parser. OPGs also have the locality property, allowing parsing to start from any point of a sentence making it ideal for incremental parsing.

Figure 2.3 shows an OPG representation of a simple arithmetic expression containing addition and multiplication. In this example `Expr` is the starting symbol, `Expr`, `EMul` and `EAdd` are non-terminals and `‘+’`, `‘*’` and `‘n’` are terminals, where `‘n’` represents integer numbers. Precedence relations are calculated between any two terminals `a`, `b` that can be adjacent to each other if non-terminals are ignored. If `a` and `b` exist in the same production, such as `‘n’` and `‘*’` they have equal precedence (`‘n’ = ‘*’`), if `a` exists in a production that is called before `b` such as `‘*’` and `‘+’` `a` takes precedence over `b` (`‘*’ > ‘+’`) and in the reverse situation, such as `‘+’` and `‘*’`, `a` yields precedence to `b` (`‘+’ < ‘*’`).

2.2.2 Attribute Grammars

Attribute grammars are an extension to CFGs, introduced by Donald Knuth [15], where semantic and context-sensitive information can be attached to each production in the grammar. The attributes are divided into two categories: synthesized attributes and inherited attributes. Synthesized attributes are passed from child nodes to parent nodes and inherited attributes are passed from parent nodes to child nodes. Figure 2.4 shows the attributes of the grammar from Figure 2.3, describing the semantic evaluation of the productions. Subscripts are used to differentiate between different occurrences of the same non-terminal. The attributes of every production is dependent only on information from non-terminals that will be children of the production, thus all attributes in the example are synthesized.

SiDECAR parses a program in a bottom-up fashion, as such it only uses synthesized attributes, since those attributes will always be available at the time it traverses a node. To support inherited attributes, the whole AST would have to be available before any semantic evaluation can be done. Using only synthesized attributes allow a high degree of concurrency, which helps speed up the parsing. Additionally, it makes it possible to parse different independent branches at the same time, since the computation of any node can be started as soon as the attributes of its children are available.

2.3 SMT-solvers

The most common method to verify that a program conforms to its specification is to translate it into verification conditions, logical formulas constructed from the code in such a way that proving their validity proves the validity of the entire program. SMT-solvers take one or more logic formulas and check whether they are satisfiable or not, i.e. if there is any set of values for all variables such that the formulas hold. For verification of programs it is not enough to check that they are satisfiable, they must be valid, i.e they must hold for all possible variable assignments. To check for validity the formulas are negated before they get fed to the SMT-solver. If the SMT-solver is unable to find a set of assignments to satisfy the negated verification condition it is considered valid. In essence the check ensures that there are no combination of assignments that break the formula (satisfies its inverse).

2.3.1 Z3

Z3 is the SMT-solver that is used in this project. It is developed by the RiSE team at Microsoft Research and specifically targeted at solving problems arising from software verification and software analysis, and consequently has integrated support for a wide selection of theories [10]. Z3 supports the reuse of proofs, or partial proofs, which is essential to the incrementality of the verification process. By manipulating

```
(declare-const x Int)
(declare-const y Int)
(assert (> x 10))
(assert (< y x))
(check-sat)
(get-model)
```

(a) Input code

```
sat
(model
    (define-fun x () Int
        11)
    (define-fun y () Int
        10)
)
```

(b) Model satisfying the input formula

Figure 2.5: Input and output for a simple Z3 verification task

the stack and reusing proofs it is likely possible to make very efficient incremental verification although Z3 will reuse proofs or partial proofs as long as they exist in the current session of the solver.

Z3 follows the SMT-LIB standard for SMT-solvers, an initiative to standardize the functionality and structure of different solvers such that they can more easily be interchanged [5]. The SMT-LIB standard provides a common language for input and output with a program library that supports the constructs of many different programming languages. It also provides a rigorous benchmark to test the correctness and speed of a verifier.

The input language uses prefix notation for example `+ 5 4` instead of the more common infix notation: `5 + 4`. Figure 2.5 shows a simple Z3 verification task and the output that it generates. As can be seen in Figure 2.5a the formulas we are trying to satisfy are `x > 10` and `y < x` and as can be seen in Figure 2.5b the solver satisfies these formulas by setting the value of `x` to `11` and `y` to `10`. Because of the prefix notation it becomes necessary to include a lot of parentheses for any non-trivial verification condition, so to increase readability in future examples involving Z3 code the infix notation is used instead.

3
Implementation

In this chapter we present the design and implementation of our incremental verifier, the challenges faced along with the solutions. In section 3.1, we show the process of transforming the Boogie grammar into OPG form. Section 3.2 describes the implementation of our pre-processor, while section 3.3 describes the use of SiDECAR to generate verification conditions. Finally, section 3.4 describes the version control that enables the use of incrementality in the verifier.

3.1 Transforming grammar to OPG form

SiDECAR requires that the input grammar is in OPG form, as such once the language subset had been chosen the grammar was converted, the result of which can be found in Appendix A. The subset was chosen so that the language would retain most of its major features, such as if-then-else cases, while-loops, and procedure calls. This allows the creation of realistic test scenarios, which should be adequate to provide a proof of concept for incremental verification, while still being feasible to implement during the time frame of the project.

Rewriting the grammar into OPG form is a challenging process in which rules have to be rewritten such that no production has two adjacent non-terminals. In Figure 3.1 the process of translating the rule E0 from the original Boogie grammar is shown. Figure 3.1a shows the original rule, where `E1`, `EquivOp` and `E0` are all adjacent non-terminals. This rule is simple to fix since `EquivOp` is only reduced to one terminal, which therefore can be inlined as `'<==>'`, the result can be seen in Figure 3.1b. Most rules are not this easy to fix however, and may therefore need to be split into several new rules, but the same process as described above applies.

Transforming the Boogie grammar into OPG is not enough in order to use it with

```
E0 ::= E1 | E1 EquivOp E0
EquivOp ::= '<==>'
```

(a) The original rule

```
E0 ::= E1 | E1 '<==>' E0
```

(b) The translated rule

Figure 3.1: Translating the E0 rule into OPG form

```
Stmt ::= havoc Id,+
```

(a) Original Boogie rule

```
Stmt ::= havoc idList
idList ::= idList ',' Id | Id
```

(b) Rewritten rule

Figure 3.2: An example of translating the havoc statement to remove the superscript notation from Id$^{,+}$

```
'procedure' ID pSig ')' specList '{' stmtList '}'
```

Figure 3.3: procecureDecl rule after being rewritten into operator form

SiDECAR, since SiDECAR lacks support for some grammar notations commonly used for context free grammars, including Boogie. The original definition of the grammar rules in Boogie uses superscript symbols such as * to denote zero or more repetitions, $^+$ for one or more repetitions, ,* for zero or more comma-separated symbols, $^{,+}$ for 1-or-more comma-separated symbols, and $^?$ for optional symbols. SiDECAR does not support this syntax; instead, productions have to be multiplied to cover the different cases. Figure 3.2 shows an example where this is done for the havoc statement rule, where the original rule can be seen in Figure 3.2a. To convert it into a format that SiDECAR accepts it has to be split into two productions, the result of which can be seen in Figure 3.2b. This process of multiplying and expanding rules severely impacts the readability of the grammar and subsequently makes it harder to work with. The full grammar used in this thesis can be seen in Appendix A.

3.2 The pre-processor

For the incremental parser created using SiDECAR, described in section 3.3, to work a lexer is needed to tokenize the input source code. This is because SiDECAR requires all the code to be inserted as tokens rather than raw source code. As a result the ANTLR4[1] framework was used to create a lexer that prepares the input before it is sent to SiDECAR.

As the project has grown so has the functionality of the lexer. At first, it was only supposed tokenize the input, but now also assists with variable renaming and with some of the challenges faced when dealing with OPG, and as such it acts like a pre-processor.

3.2.1 Assistance with OPG

As mentioned in section 3.1, converting the grammar to an OPG was a challenging process. After the final grammar had been selected and inserted into SiDECAR, two

[1]http://www.antlr.org/index.html

```
program ::= 'assert' expr
expr ::= e0
e0   ::= e1 | e0 '==' e0
e1   ::= IDENTIFIER
       | INTEGER
```

(a) Simple example grammar

```
program ::= 'assert' e0
          | 'assert' e1
e0 ::= e0 '==' e0 | e0 '==' e1
     | e1 '==' e0 | e1 '==' e1
e1 ::= IDENTIFIER
     | INTEGER
```

(b) Extension of the grammar from Figure 3.4a to deal with reductions

Figure 3.4: Example of a simple grammar translated using one method of dealing with reductions

```
program ::= 'assert' expr
expr ::= 'exp' e0
e0   ::= 'exprJump' e1 | e0 '==' e0
e1   ::= IDENTIFIER | INTEGER
```

Figure 3.5: Extension of the grammar from Figure 3.4a with additional terminals added by the pre-processor to prevent reductions

issues were discovered: instances of parser ambiguities and unwanted reductions.

The ANTLR parser had no issues dealing with the grammar; however, SiDECAR did with certain rules. One instance of this was with the procedureDecl rule, seen in Figure 3.3, where SiDECAR detected ambiguities with the ')' character of the procedureDecl and pSig reductions. The information SiDECAR has when parsing such a rule is insufficient to know which production it should reduce to. One simple way of solving this was to introduce a special terminal character after the ')' character for the procedureDecl rule in SiDECAR, solving the ambiguity between the productions. The special character is added by the pre-processor and as such the internal grammar used by SiDECAR is slightly different from the grammar of the Boogie language while still behaving in the intended way. As a result minor grammatical changes can be made to the internal representation of the grammar without impacting how a program is constructed by the user.

Another issue faced was that SiDECAR immediately reduces non-terminals as much as possible, rendering grammar rules that only refers to other rules being reduced to an incorrect form. To illustrate this, consider the example found in Figure 3.4. Figure 3.4a shows a simple example of a grammar in OPG form, where `IDENTIFIER` represents a variable name. With the input `assert foo` one would normally expect it to be evaluated to `'assert' expr`, but because of the aforementioned behavior SiDECAR reduces it to `'assert' e1`. This causes a mismatch since that production does not explicitly exist. To overcome this initially, every permutation that could occur for the affected rules were created. The result of such an approach is illustrated in Figure 3.4b, as can be observed the resulting grammar now consist of eight rules instead of the original six. With a more complex chain of expressions this blow-up is significant, since each new level of expression would yield a new production for every rule containing expressions. This makes the method unscalable since the

```
x = 0;
x = x + 1;
```

(a) Original code

```
x0 = 0;
x1 = x0 + 1;
```

(b) The code with variables renamed

Figure 3.6: Variable renaming for a simple piece of code

number of rules grew too big, making the grammar complicated and cumbersome to work with. To avoid this problem a solution where the pre-processor inserts dummy terminal tokens into the code was adopted, similar to solution for the ambiguity issues described earlier in this section. This hinders SiDECAR from immediately reducing the rules so that they can be properly matched instead.

The resulting grammar from inserting terminals can be found in Figure 3.5. Now the grammar is back to its original six rules, and since this is done in the pre-processor, it will not affect the grammar used when constructing a program. The user input will still follow the grammar listed in Figure 3.4a, but internally it will be converted to the grammar in Figure 3.5.

3.2.2 Variable renaming

The Z3 SMT-solver used in this project does not support variables, as such constants are used in a single static assignment scheme to simulate their functionality. This requires each variable assignment in the program to use a fresh variable and each reference to refer to the last declared instance of that variable. Variable renaming is done by the pre-processor, again to not impact how a program is constructed. Figure 3.6 shows an example of how a simple procedure is converted to single static assignment by the pre-processor. In Figure 3.6a the variable `x` is used in the regular fashion, and in Figure 3.6b `x` has been duplicated so that each variable is only assigned to once and so that each reference to a variable points to the last one declared.

In the ANTLR4-powered pre-processor each variable is renamed. For an assignment, first any references to variables on the right-hand side is replaced with references to the last generated instance of those variables, then a new fresh variable is assigned to the left-hand side.

For items such as if-cases, renaming variables within them becomes more complicated, since at the next reference of that variable, after the if-case, it is not know which instance of the variable should be use. To illustrate this issue: Figure 3.7 shows the original code. When the variables are renamed the program does not know which instance of x to use at the `y:=x;` statement. The solution adopted in this project is to keep track of whichever variables are used within the if-cases and afterwards create a new variable that is assigned the correct instance, based on the original condition of the if-case. This is illustrated in Figure 3.8.

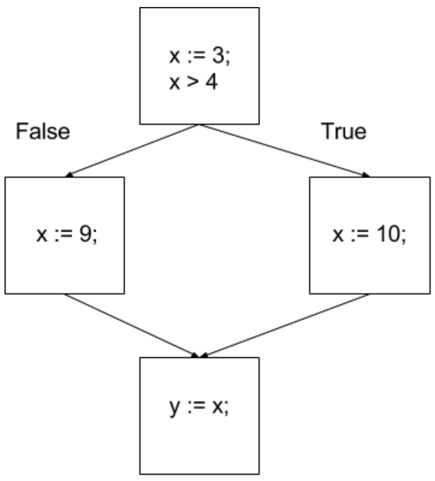


Figure 3.7: Example of a program with an if-case

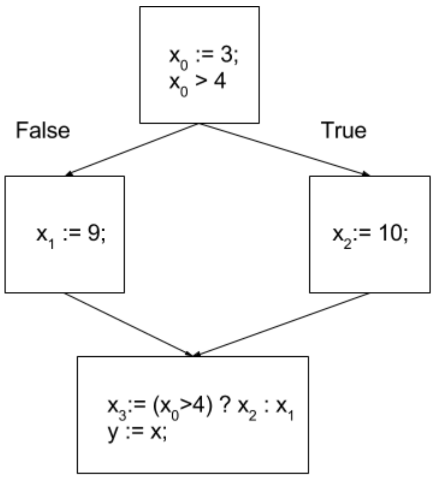


Figure 3.8: Example program from Figure 3.7 rewritten to use single static assignment

Table 3.1: Weakest precondition of predicate Q through a statement S for Boogie

Statement S	W(S,Q)
x := e	Q [x $\mapsto$ e]
havoc x	Q [x $\mapsto$ x']
assert e	e $\wedge$ Q
assume e	e $\implies$ Q
S ; T	W(S,W(T,Q))
if (b) then {T} else {E}	(b $\implies$ W(T,Q)) $\wedge$ ($\neg$ b $\implies$ W(E,Q))
while (b) invariant J {b}	$\begin{cases} \text{J} \\ \text{W(havoc t(B); assume J} \wedge \text{b; b, J)} \\ \text{W(havoc t(B); assume J} \wedge\neg \text{ b, Q)} \end{cases}$
call t := P(a)	W(assert p(P(a)); havoc t, f(P); assume q(P(a)), Q)
procedure P(a) requires R ensures E {B}	R(a) $\implies$ W(B,E(a))

3.3 Incremental verification condition generation

To generate a minimal verification condition the tool SiDECAR, presented in section 2.2, is used for its incrementality. The pre-processor prepares the input for the SiDECAR-powered parser, which handles the generation of verification conditions that can be passed to the SMT-solver for verification. Verification conditions are generally generated in the same manner as in Boogie, shown in Table 3.1 [3]; however, the implementation chosen in this project differs slightly for some statements, which are described in this chapter.

For assignments, rather than using the backwards substitution from Table 3.1, a separate assertion is added outside of the main formula, establishing that the value for a given variable can only be a specific value. This approach works due to the already performed variable renaming into single static assignment form described in section 3.2.2.

3.3.1 If-cases

In Figure 3.9 the conversion of an if-case is displayed, from the Boogie form in Figure 3.9a to the VC in Figure 3.9b. The final VC is slightly different from the one displayed in Table 3.1; however, the end result is equivalent due to the line `<ite for any assignments>` in the figure. The `ite` command chooses between two effects based on some input condition, this makes it possible to choose between instances of a variable depending on which branch was taken in the particular verification path. As described in section 3.2.2, any assignments to a variable within one or both of the two branches for the conditional creates an additional 'if-then-else' conditional that ensures that any side effects T or F might have will carry over to the rest of the program `Q`. As an end result, the final result from the verification condition ends

```
if (c) {
        T;
} else {
        F;
}
Q;
```

(a) Simple if-case

```
(c => (T and Q))
and
((not c) => (F and Q))
and
<ite for any assignments>
and
Q
```

(b) The generated verification condition

Figure 3.9: The transformation of a common if-case (a) into a verification condition (b)

up the same.

3.3.2 While-loops

As discussed in section 2.1.1, loops are harder to reason about since there is no way for the verifier to know how many times the loop will be executed, except in the most trivial cases. To solve this, a loop invariant is attached to each loop to help understand how its execution affects the loop target, i.e. all variables modified by the loop [14]. The loop invariant must hold before and after each iteration of the loop and be designed in such a way that it, together with the negation of the loop condition, provides a postcondition for the loop.

Verification of a loop is done in three steps: first the loop invariant is checked to hold initially, i.e. before any iteration of the loop as been executed, called the initiation. The second step, called consecution, checks that the invariant is indeed invariant. All variables of the loop target are assigned non-deterministic values, the loop condition and the invariant is assumed and the loop body is executed once. This corresponds to an arbitrary execution of the loop, after which the loop invariant should still hold. When initiation and consecution is done, the execution of the loop can be symbolized by assigning non-deterministic values to the loop target and assuming the invariant as well as the negation of the loop condition, this is called continuation. If the loop invariant is sufficiently specified this assumption should give the solver no ambiguities over which values to assign the variables of the loop target.

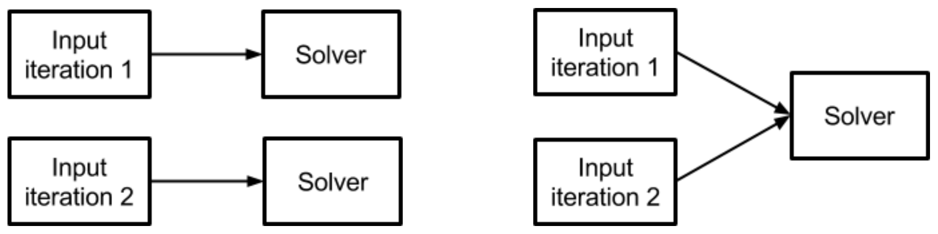
```
while (x < y)
invariant x <= y;
{
  x := x + 1;
}
```

(a) Simple while loop

```
(x_0 <= y_0) and
((x_1 <= y_1) and (x_1 < y_1)
    => (x_2 = x_1 + 1) and
    (x_2 <= y_1)) and
((x_3 <= y_2) and
    not(x_3 < y_2)) => Q)
```

(b) The generated verification condition

Figure 3.10: Verification generation from a simple while loop



(a) Non-incremental solving of two iterations of a program

(b) Incremental solving of two iterations of a program

Figure 3.11: Two different ways of interacting with a verifier.

Figure 3.10 shows the verification condition generated from the loop example discussed in section 2.1.1 (repeated in 3.10a) with some syntax simplifications to increase readability. `x_0` and `y_0` are bound variables and dependent on whatever value they might have been assigned earlier in the program, while `x_1, x_3, y_1` and `y_2` are free variables. The first row of Figure 3.10b represents the initiation step of the verification process, rows 2-4 represent the consecution and rows 5-6 represent the continuation. `Q` in the continuation step is the verification condition generated for the rest of the procedure.

3.3.3 Procedures and procedure calls

The original Boogie verifier uses modular verification for procedures [17], to achieve modular verification for this project multiple solvers are used. For each procedure a new solver is created, allowing for procedures to be solved individually, without interfering with each other, and will only be required to be re-checked if there has been a change to them between iterations.

As can be seen in Table 3.1, the verification condition for a procedure is very straightforward and it ends up being: precondition $\implies$ body $\implies$ postcondition. As such, the program can be considered invalid if there is an assertion in the body that does not hold, or if there is an assignment in the body that causes the postconditions to not hold.

Calling a procedure from another one will use the contract of the called method, as can be seen for the call entry in Table 3.1. This means that when procedure P is called, only its preconditions, p(P(a)), are checked to be correct, and any effect the procedure P might have is assumed from its postconditions, q(P(a)). Only using the contract works since the called procedure will be checked separately regardless of whether it has been called or not, and is required to be valid for the program to be considered valid.

```
assert (=> version0 VC0)
assert (=> version1 VC1)
(check-sat (not version0) version1)
```

Figure 3.12: Example of how an updated verification, VC1, can be checked without the older version, VC0, interfeering

3.4 Version control

Figure 3.11 illustrates two different ways of feeding verification inputs to the SMT-solver, depending on what method of verification is used, incremental or non-incremental. Typically, non-incremental verifiers are rerun between each iteration, as shown in Figure 3.11a, while the verifier developed in this project is designed to continue running between verifications so that the instance of the SMT-solver is reused, as shown in Figure 3.11b. In this project the target SMT-solver is Z3, which can be used to incrementally solve a provided model [11], this design where the same instance is reused allows for the incremental support within Z3 to reuse previously computed proofs. However, this requires that the stack be modified so that previous iterations of a program does not interfere with the solving of the new one.

In SMT-solvers a common way of changing the stack is to use `push`, which will mark a point on the stack, and `pop` which will remove instructions from the stack to the last point marked by push [6]. However, this only allows for modifications of the top of the stack.

Another method presented by Niethammer [21] is to instead use assumptions for version control, where an implication is used with a boolean on the lefthand side and the verification condition of a program on the righthand side so that the verification condition can be toggled as active or inactive. An example of this can be seen in Figure 3.12, `VC0` and `VC1` are verification conditions for different iterations of a program. The downside of this method is that the stack grows with each new version, and therefore uses more memory. This is the method used in this project. Once the verification condition has been generated by the incremental parser an implication and version flag is added to it before being passed to the SMT-solver.

4
Results

This chapter describes how the final product from the project was tested, and displays the results from those tests. In section 4.1, we present the cases, how they are constructed, and how they are used. Section 4.2 describes the method behind collecting the test results, and section 4.3 presents the results and some interesting outliers.

4.1 Test cases

The test cases were selected to contain all features of the Boogie language supported by our tool. For most of the tests they were made artificially larger by duplicating the content of their main procedure (or equivalent) to make them run longer. This was done after it was discovered that the initial tests ran very quickly, usually around 1-2 second, making the measured times brittle. With the longer tests the quality of the results increased and the fluctuations disappeared. For most of the test cases, changes between iterations were made to feel realistic, sometimes large and sometimes small.

All tests were constructed to consist of three iterations. This decision was made to be able to construct a large number of tests that are comparable to each other. The first four test cases were also extended with an additional two iterations to examine how more iterations would affect the results. All iterations of the different test cases are invalid, except the last iteration, which is valid.

The degree of modification between iterations for cases with three iterations and five iterations can be seen in Table 4.1 and Table 4.2 respectively. Following is a brief description of the test cases[1]:

- Test0 contains a large set of if-then-else cases following each other in one procedure.
- Test1 contains a large number of assignments, with eight procedures. The value of each assignment is completely changed between iterations.

[1]Full tests available at: https://github.com/anttila/incrementalBoogie/tree/master/testprograms

Table 4.1: The changes between each pair of iterations for test cases with three iterations

Test name:	1-2:	2-3:
Test0	80%	1%
Test1	98%	98%
Test2	98%	98%
Test3	3%	13%
Test4	11%	1%
Test5	29%	24%
Test6	90%	70%
Test7	3%	57%
Test8	80%	84%

Table 4.2: The changes between each pair of iterations for test cases with five iterations

Test name:	1-2:	2-3:	3-4:	4-5:
Test0	80%	2%	1%	75%
Test1	98%	98%	98%	98%
Test2	98%	98%	98%	98%
Test3	3%	62%	1%	3%

- Test2 is the same test as Test1, but it has double the amount of procedures.
- Test3 contains a large number of while-loops in one procedure.
- Test4 contains an arithmetic test with five procedures. The test uses a large number of the language constructs: procedures, calls, havoc, asserts, assumes, assignments, while-loops and if-cases.
- Test5 contains an alarm clock with seven procedures. Similar to Test4 it essentially uses all language constructs.
- Test6 contains a large number of cascaded if-cases in one procedure. The difference between the iterations is very large: 90% and 70%.
- Test7 contains a large number of cascaded if-cases as well. However, in this test the difference is not as large as in Test6: 3% and 57%.
- Test8 is the same as Test0 but with much larger changes between iterations 2 and 3, 84% as opposed to 1%.

4.2 Method of collecting results

Test results were collected by running each test ten times for both incremental and non-incremental verification, and then reported using the median result. Time was measured using a Python script that measured time from execution of the program, to finish. The time measured contains the total time for all iterations. Between each pair of test cases the solver is restarted to clear its cache.

The computer used for testing was an i5-4690K processor, with 16 GB of RAM, using Windows 10 with minimal processes running. Java 1.8.0_131 and Z3 version 4.5.0 were used, both using their 64-bit versions.

4.3 Results

The results of the collected test data can be found in Table 4.3 for the cases of three iterations, visualized in Figure 4.1. For the tests with five iterations the results can be found in Table 4.4, visualized in Figure 4.2.

As can be seen in the "Time saved"-column in Table 4.3, in most cases there is a performance gain of at least 30% when using the incremental solver on programs with three iterations. There are a couple of particularly interesting results: Test6 and Test8 have a similar degree of change between iterations yet Test8 runs considerably faster than the Test6, which interestingly sees a marginal increase in runtime. A similar situation is present in Test3 and Test4 where Test3 runs considerably faster than Test4. Finally, in test Test1 and Test2 the values of the assignments are completely changed yet we still see a considerable speed-up in verification. Possible reasons as to why this is the case is discussed in chapter 5.

When running Test0, Test1, Test2 and Test3 using five iterations, seen in Table 4.4, the results are around 10 percentage points higher in the "Time saved"-column than the results for three iterations seen in Table 4.3, with the exception of Test0 which had a decreased time-save by 20 percentage points.

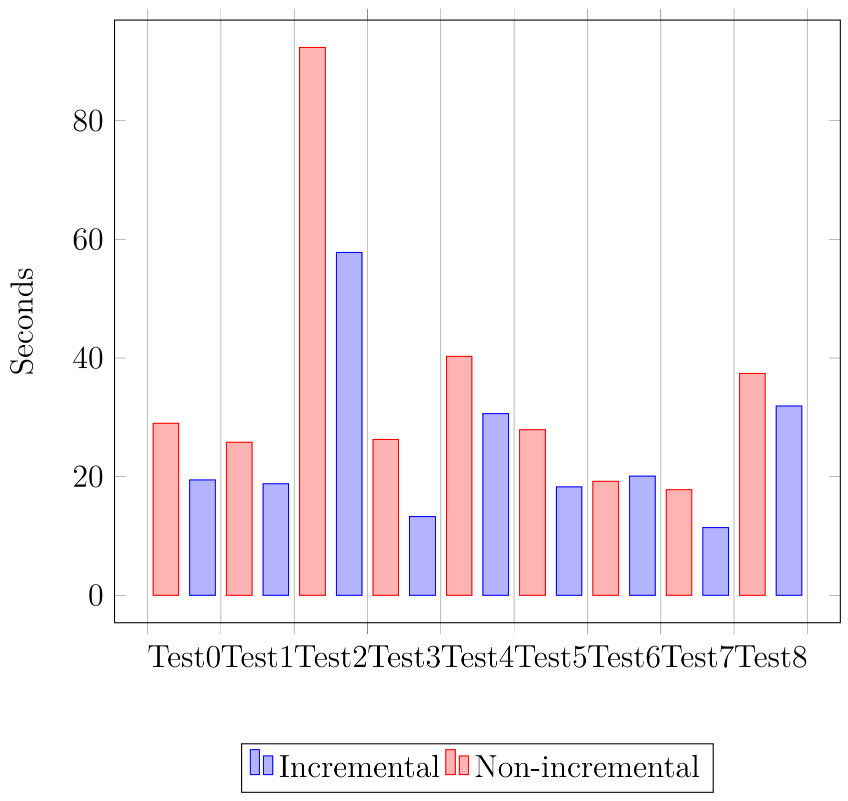


Figure 4.1: Visual representation of the results for programs tested with three iterations, gathered in Table 4.3

Table 4.3: Results from running the test suite against the final program for the project with three iterations for each Boogie program, displaying the difference between using the incremental and non-incremental modes

Test name:	Incremental time (s):	Non-incremental time (s):	Time saved:
Test0	19.46	29.00	32%
Test1	18.81	25.81	27%
Test2	57.80	92.35	37%
Test3	13.28	26.27	49%
Test4	30.63	40.28	24%
Test5	18.29	27.92	34%
Test6	20.10	19.22	0%
Test7	11.42	17.81	36%
Test8	31.92	37.40	15%

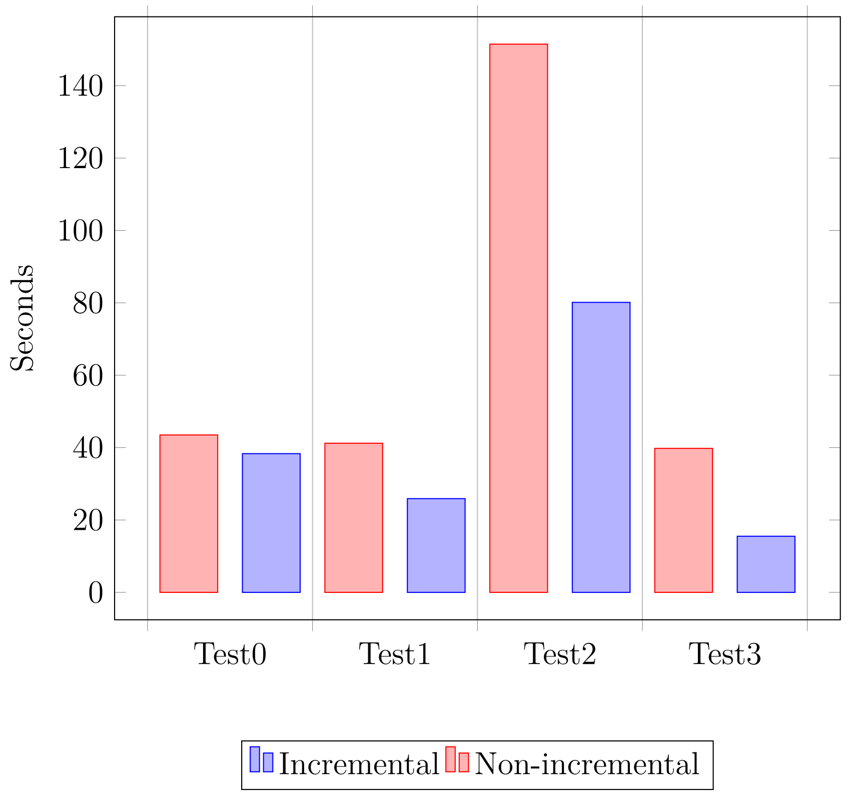


Figure 4.2: Visual representation of the results for programs tested with five iterations, gathered in Table 4.4

Table 4.4: Results from running the test suite against the final program for the project with five iterations for each Boogie program, displaying the difference between using the incremental and non-incremental modes

Test name:	Incremental time (s):	Non-incremental time (s):	Time saved:
Test0	38.33	43.49	12%
Test1	25.90	41.21	37%
Test2	80.13	151.47	47%
Test3	15.51	39.79	61%

5
Discussion

In this chapter we discuss different aspects of the project in its entirety. In section 5.1, we look at and analyze the results from the evaluation in chapter 4. Section 5.2 discusses the limitations on the test cases we could build, as imposed by the restricted language that the verifier supports. Finally, in section 5.3, we propose a different method of handling the grammar when interacting with SiDECAR.

5.1 Experimental testing

The results, presented in chapter 4, show promise for incremental verification as a technique for formal deductive verification. As expected, smaller changes between iterations lead to an increased time save when comparing incremental verification to non-incremental. In our experience, when programming it is quite typical that modifications are relatively small between different versions of a program, especially when debugging.

From the results of our tests with an increased number of iterations it can be seen that more iterations of a program often lead to quite substantial increases in the time saved for each program, which was an expected result. The first iteration of a program being verified will always be completely fresh, meaning that it receives no benefits from the incrementality of the verifier. The more iterations of a program the less significance the time of the initial iteration has on the overall result. However, more iterations also consume more memory, so at some point it is likely that the increase in memory usage might cause issues and result in diminishing returns on the incremental technique.

Something we did not formally test was the possible impact of the end result of the verification, i.e. whether the program was valid or not. We expect that it takes more time to prove that a program is valid rather than invalid, since in the latter case the solver only has to find one set of assignments that breaks the verification conditions, whereas in the former it has to make sure there are no such cases at all. Some informal testing showed no indication that this impacts the speed-up gained using the incremental technique. This makes sense to us since the solver still has to do the same task for both techniques, but nevertheless it would have been interesting to perform formal testing of these scenarios as well.

In the following couple of paragraphs a discussion regarding the cause of the outlying results, shown in section 4.3, is presented. It is important to note that these are speculations, it would require a more deep-going analysis to confirm that this is the case; this is made more difficult by the amount of heuristics and back-end optimizations involved in the SMT-solver.

Test6, which is the test with cascading if-cases and large changes between iterations, is the only test that sees an increase in run time with the incremental technique. We speculate that there are two reasons for this result. First, the changes between iterations are so significant that the overhead gets proportionally larger since very few previous proofs can be reused. However, Test8, which is the test with a series of if-cases, has a similar degree of change between iterations and still manages a discernible performance increase, leading us to the second reason: the complexity of nested ifs make it harder to reuse proofs.

Test3, containing a large number of loops in a single procedure, has a considerably larger speed-up than Test4, which tested different arithmetic properties, again with a similar degree of change between iterations. In this case it is harder to justify this behavior; it could be that the complexity of the while-loops in Test3 is low enough that they are easy for the verifier to check, or that the multiple procedures in Test4 have a larger impact on the effectiveness of the incrementality.

Test1 and Test2, which are the same tests consisting of a set of assignments except that Test2 has twice as many procedures, both have the values in their assignments completely changed between iterations but still give rise to a considerable increase in verification speed using incrementality. Thus, it seems that the structure of a program has a big impact on the degree to which proofs can be reused. Additionally, Test2 has a somewhat larger speed-up compared to Test1 which seems to undermine the argument made in the last paragraph. However, most procedures in Test4 are a lot smaller than those in Test1 and Test2, so it is possible in this case that the incrementality gained inside the procedures overtake the performance lost by the overhead of verifying different procedures. But, again, since the SMT-solver is more or less a black box, this kind of analysis is highly speculative.

Finally, Test0, a number of if-cases following each other, with five iterations had a significant decrease in speed-up when compared to the three iteration version. We do not know, or even have an idea with enough credibility to present as a speculation, on why this is. Further testing could give some clues as to why this is the case but even then it would be hard to determine the exact cause.

5.2 Input language limitations

In order for the project to be finished within the given time frame some limitations on which language constructs to support had to be done, shown in section 1.3.

Consequently, this limited the space of possible test cases, making it harder to produce realistic test programs. In particular, arrays and quantifiers are used in lots of algorithms, for example binary search or bubble sort, which are typically useful for testing. Since loops are often used to iterate over lists and arrays, and quantifiers are required to reason about much of the behavior of loops, our syntactic restrictions also limit the complexity of the while loops used in testing. However, while it would be interesting to be able to handle and build tests using these constructs, we believe the tests that were performed to be rigorous enough to show that incremental verification holds potential for future research and development.

5.3 Implementation

As an afterthought, the method of using the pre-processor, mentioned in section 3.2.1, to insert terminals could have been used to create a simpler grammar. While SiDECAR requires the grammar to be in OPG form, ANTLR does not, meaning that the original grammar for Boogie could have been used as it was, without being rewritten to OPG form. If the grammar had been used as-is, the internal SiDECAR grammar could have had dummy characters inserted between any non-terminals, which would be handled by the pre-processor. This would have resulted in a much simpler grammar to read and work with.

An observation that was made during testing is that the parsing of the program is very fast, often no more than a couple of percent of the overall verification time. This brings into question the necessity of using an incremental parser for the verification condition generation. Since the produced VC will be the same by either method and regular parsers are a lot easier to work with, it is possible that using incremental parsing might not be worth the extra work. However, we do not use SiDECAR to its full potential, so it is possible that the incremental parser gives access to some optimization techniques that could not be implemented in a regular parser.

6
Conclusion

In this thesis we have presented the design, implementation and evaluation of an incremental verifier targeting a subset of the Boogie language. The incrementality is built on top of an incremental parser, built using the SiDECAR tool for incremental verification, by creating verification conditions through the parsing and feeding these to the Z3 SMT-solver. In this way, only the verification conditions for the parts of the AST that has changed are regenerated and old proofs can be reused when verifying a new iteration of a program.

The resulting incremental verifier has been evaluated using a series of test cases designed to incorporate all different constructions of the subset language in realistic test scenarios, as well as some extreme cases to evaluate how the tool performs for certain constructs that generate more complex verification conditions. The results are promising, with time saves ranging from 15% to 49% when comparing incremental to non-incremental verification for three iterations, with the exception of one test scenario where the performance was slightly reduced. Using additional iterations seem to improve the results further in most of the tested cases.

A long term goal for formal verification is to have an environment where verification can be done in real time, much like syntax checking in a development environment, such as Eclipse or Visual Studio. While this is not feasible with the tool developed in this project it would be possible, for smaller projects, to do the verification with each new build of the program. However, as programs become larger it's likely that verification using our tool is only feasible as part of the process of integrating a piece of code into the larger project.

The results of this thesis gives motive for further research in incremental verification as a promising method to speed up the process of deductive verification.

6.1 Future work

It would be interesting to further develop this project to add the missing language features that were mentioned in chapter 1.3 as limitations, to see how the project would scale. Additionally, looking into possible performance optimizations could render the run time shorter. One such case would be to fully implement backward substitution as mentioned in section 3.1, which would essentially remove assignments from the verification condition.

Bibliography

[1] Stephan Arlt and Martin Schäf. *Joogie: Infeasible Code Detection for Java*, pages 767–773. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[2] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*, pages 364–387. Springer, 2005.

[3] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '05, pages 82–87, New York, NY, USA, 2005. ACM.

[4] Mike Barnett, Rustan Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. In *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362, pages 49–69. Springer, January 2005.

[5] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[6] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The SMT-LIB Standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, page 14, 2010.

[7] D. Bianculli, A. Filieri, C. Ghezzi, and D. Mandrioli. A syntactic-semantic approach to incremental verification. *arXiv preprint arXiv:1304.8034*, 2013.

[8] D. Bianculli, A. Filieri, C. Ghezzi, D. Mandrioli, and A. M. Rizzi. Syntax-driven program verification of matching logic properties. In *Formal Methods in Software Engineering (FormaliSE), 2015 IEEE/ACM 3rd FME Workshop on*, pages 68–74. IEEE, 2015.

[9] Shaunak Chatterjee, Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamarić. *A Reachability Predicate for Analyzing Low-Level Software*, pages 19–33. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[10] Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*, pages 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[11] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[12] Robert W. Floyd. Syntactic analysis and operator precedence. *J. ACM*, 10(3):316–333, July 1963.

[13] C. A. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1(4):271–281, December 1972.

[14] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[15] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Correction: Mathematical Systems Theory 5(1): 95-96 (1971).

[16] D. Kroening and C.S. Păsăreanu. *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings*. Number del 1 in Lecture Notes in Computer Science. Springer International Publishing, 2015.

[17] K. Rustan M. Leino. This is Boogie 2. *Manuscript KRML*, 178:131, 2008.

[18] K. Rustan M. Leino and W. Schulte. Verification Condition Splitting. January 2008.

[19] Rustan Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370. Springer Berlin Heidelberg, April 2010.

[20] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[21] P. Niethammer. Syntax-directed incremental verification of java modeling language contracts. unpublished thesis, 2016.

[22] Zvonimir Rakamarić and Michael Emmi. SMACK: Decoupling source language details from verifier implementations. In Armin Biere and Roderick Bloem, editors, *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*, volume 8559 of *Lecture Notes in Computer Science*, pages 106–113. Springer, 2014.

[23] A. Stefanescu. Matchc: A matching logic reachability verifier using the K framework. *Electronic Notes in Theoretical Computer Science*, 304:183–198, 2014.

[24] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Verifying eiffel programs with boogie. *CoRR*, abs/1106.4700, 2011.

A

Appendix 1: Boogie Grammar subset

```
declList: axiomDecl
        | constantDecl
        | varDecl
        | functionDecl
        | procedureDecl
        ;

axiomDecl:
        'axiom' expr ';'
        | 'axiom' expr ';' declList
        ;

constantDecl:
        'const' idType ';'
        | 'const' idType ';' declList
        ;

idType:
        id ':' type;

functionDecl:
        'function' ID fSig '{' expr '}'
        | 'function' ID fSig '{' expr '}' declList
        ;

fSig:
        '(' fArgList ')' 'returns' ')' fArg ')'
        | '(' ')' 'returns' '(' fArg ')'
        ;

fArgList:
        fArgList ',' fArg
        | fArg;

fArg:
        idType;

varDecl:
        'var' idType ';'
        | 'var' idType ';' declList
        ;

procedureDecl:
        'procedure' ID pSig ')' specList '{' localVarDeclList ';'
           stmtList '}'
        | 'procedure' ID pSig ')' specList '{' localVarDeclList ';'
            stmtList '}' declList
        | 'procedure' ID pSig ')' specList '{' stmtList '}'
        | 'procedure' ID pSig ')' specList '{' stmtList '}'
           declList
        | 'procedure' ID pSig ')' '{' localVarDeclList ';' stmtList
            '}'
        | 'procedure' ID pSig ')' '{' localVarDeclList ';' stmtList
            '}' declList
        | 'procedure' ID pSig ')' '{' stmtList '}'
        | 'procedure' ID pSig ')' '{' stmtList '}' declList
        | 'procedure' ID pSig ')' '{'  '}'
        ;

pSig:
        '(' idTypeCommaList ')' outParameters
        | '(' ')' outParameters
        ;

outParameters:
        'returns' '(' idType
        | 'returns' '('
        ;

localVarDeclList:
        'var' idType
        | 'var' idType ';' localVarDeclList
        ;

idTypeCommaList:
        idTypeCommaList ',' idType
        | idType;

specList:
        'requires' expr ';'
        | 'requires' expr ';'specList
        | 'modifies' ID ';'
        | 'modifies' ID ';' specList
        | 'ensures' expr ';'
        | 'ensures' expr ';' specList
        ;

idCommaList:
        idCommaList ',' id
        | id
        ;

type:
        mapType
        | typeAtom
        ;
```

```
typeAtom:
        'int'
        | 'bool'
        ;

mapType:
        '[' 'int' ']' type
        ;

exprCommaList:
        expr ',' exprCommaList
        | expr
        ;
expr: e0
;

stmtList:
        'assert' expr ';'
        | 'assert' expr ';' stmtList
        | 'assume' expr ';'
        | 'assume' expr ';' stmtList
        | 'havoc' idCommaList ';'
        | 'havoc' idCommaList ';' stmtList
        | lhs ':=' expr ';'
        | lhs ':=' expr ';' stmtList
        | 'call' callLhs ID '(' exprCommaList ')' ';'
        | 'call' callLhs ID '(' exprCommaList ')' ';' stmtList
        | 'call' ID '(' exprCommaList ')' ';'
        | 'call' ID '(' exprCommaList ')' ';' stmtList
        | ifStmt
        | 'while' '(' expr ')' loopInv '{' stmtList '}'
        | 'while' '(' expr ')' loopInv '{' stmtList '}' stmtList
        | 'break' ';'
        | 'break' ';' stmtList
        | 'return' ';'
        | 'return' ';' stmtList
        ;

lhs:
        id '[' expr ']'
        | id
        ;

callLhs:
        id ':='
        ;

ifStmt:
        'if' '(' expr ')' '{' stmtList '}'
        | 'if' '(' expr ')' '{' stmtList '}' else_
        | 'if' '(' expr ')' '{' stmtList '}' stmtList
        ;

else_:
        'else' ifStmt
        | 'else' '{' stmtList '}'
        | 'else' '{' stmtList '}' stmtList
        ;

loopInv:
        'invariant' expr ';'
        ;

// Expr:
e0:
        e1
        | e1 '<==>' e0
        ;

e1:
        e2
        | e2 '==>' e1
        ;

e2:
        e3
        | e2 '||' e3
        | e2 '&&' e3
        ;

e3:
        e5
        | e5 '==' e5
        | e5 '<' e5
        | e5  '>' e5
        | e5 '!=' e5
        | e5'<=' e5
        | e5 '>=' e5
        ;

e5:
        e6
        | e5 '+' e6
        | e5 '-' e6
        ;

e6:
        e7
        | e6 '*' e7
        | e6 '/' e7
        ;

e7:
        e8
        | '!' e8
        ;

e8:
        e9
        | e9 '[' expr ':=' expr ']'
        | e9 '[' expr ']'
        ;

e9:
        'false'
        | 'true'
        | integer
        | id #EId
        | id funcApplication
        | 'old' '(' expr ')'
        | '(' 'forall' idTypeCommaList '::' expr ')'
        | '(' 'exists' idTypeCommaList '::' expr ')'
        | '(' 'forall' idTypeCommaList '::' triggerList '}' expr ')
           '
        | '(' 'exists' idTypeCommaList '::' triggerList '}' expr ')
           '
        | '(' expr ')'
        ;

triggerList:
        '{' exprCommaList
        | '{' exprCommaList '}' triggerList
        ;

funcApplication:
        '(' exprCommaList ')'
        | '(' ')'
        ;
```