CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

# Invariants from Tests in Boogie

## Combining Dynamic and Static Testing Methods to identify Loop Invariants of Boogie Programs

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

Timon Lapawczyk

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

# Invariants from Tests in Boogie

Combining Dynamic and Static Testing Methods to identify Loop Invariants of Boogie Programs

Timon Lapawczyk



Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Invariants from Tests in Boogie
Combining Dynamic and Static Testing Methods to identify Loop Invariants of Boogie Programs
Timon Lapawczyk



Supervisor: Carlo A. Furia, Department of Computer Science and Engineering
Examiner: Wolfgang Ahrendt, Department of Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

# Abstract

Proving implementations with loops against their specifications using automatic verifiers, requires annotations in the form of loop invariants. Loop invariants are properties that must hold for every iteration of a loop and identifying them is generally a difficult endeavour. Invariant inference algorithms can assist this process by generating possible invariants and identifying candidate invariants, using static or dynamic testing.
This thesis examines how symbolic execution can compensate for the lack of traditional dynamic testing methods in the context of Boogie programs, to identify candidate invariants for such programs using a combination of dynamic and static testing. Boogie is a verification language that combines a typed logic and a simple procedural language; this combination makes Boogie programs not executable in general, which makes applying dynamic analysis techniques challenging. This thesis implements an algorithm for invariant inference that generates possible invariants from templates and post-conditions. The identification of candidate invariants is split into disproving as many wrong invariants as possible, using the symbolic execution engine Boogaloo, and identifying candidate invariants from the remaining set of invariants, by proving them with the Boogie verifier.
A detailed analysis, on 15 carefully selected Boogie programs, shows that the combination of dynamic and static testing can be quite powerful, being able to infer invariants sufficient to prove 10 of the 15 programs correct fully automatically. However, the results also suggests a connection between the kinds of possible invariants that are generated and the impact the symbolic execution has on the performance.

Keywords: Computer, science, computer science, engineering, project, thesis, loop invariant, invariant inference, Boogie, Boogaloo, symbolic execution, software verification, formal methods.

# Acknowledgements

At this point I would like to thank my supervisor Carlo A. Furia for his guidance throughout this thesis. He allowed me the freedom to shape the thesis towards the topics which interested me the most, always had great advice when I got stuck and was very supportive in general.

Timon Lapawczyk, Gothenburg, June 2018

# Contents

# 1
# Introduction

With growing interest in formal methods, the need for formally verified programs is growing as well. Having formally verified programs requires two things. A formal specification and a proof that the implementation matches the formal specification. A simplified example for a formal specification includes pre-conditions, which specify properties that hold before a function call, and post-conditions, which specify properties that hold after the function call.

```
pre-condition x >= 0;
{ result = x; }
post-condition result >= 0;
```

Proving an implementation against its formal specification includes a formal proof that, given the pre-conditions, the execution of the function body will result in the post-conditions being true. This task can be highly automated with modern verification tools. Small programs can be verified fully automatically but the undecidability of predicate logic still makes this task an undecidable problem. In practice, most applications of formal verification require a lot more input from the user to verify an implementation automatically. This input can consist of different kinds of annotations, at different points throughout the implementation, which state properties that are true, at these points. The verification tool will have to prove these properties but they can be a means to sort of guide the verification tool through the different sections of longer implementations. One very important form of annotation is the loop invariant, which is a property that has to hold before the first iteration and after every iteration of a loop. The following pseudo code shows a function which calculates the identity of a number $n$ by counting from 0 up to the value of $n$.

---

**Pre-Condition:** $n >= 0$;
**Post-Condition:** $i == n$;
$i = 0$;
**while** $i < n$ **do**
  $i++$;
**end**

---

Verification tools are generally not able to automatically verify the post-condition $i == n$ given the pre-condition $n >= 0$. Adding the property $i <= n$, which is true before the first iteration and after every iteration of the loop, as a loop invariant, enables verification tools to verify this implementation automatically.
While this example is pretty trivial, writing invariants is generally no trivial task. For more complex implementations it usually requires trained engineers with a de-

tailed understanding of the implementation and its specification and experience with the verification tool, to annotate a loop with the correct invariants. However, certain patterns can be found in loop invariants and they often relate closely post-conditions from the specification. This observation raises the question if and how the important step of writing loop invariants can be automated.
Different work has shown that automatic invariant inference can have a significant effect on the capabilities of automatic verification for certain domains of programs. This thesis further explores the capabilities of automatic invariant inference, by implementing an invariant inference algorithm for Boogie [11] programs.
The Boogie intermediate verification language and its like-named verification tool provide a layer on which to build program verifiers for other languages. There are already a hand full of front-ends, translating programs from high level programming languages into Boogie and Boogie already powers static program verifiers such as Dafny [10].
The two main components of invariant inference are the generation of possible invariants and the identification of candidate invariants, which are likely to help with the verification of the post-conditions. The algorithm generates invariants from external templates and the post-conditions from the specification. To identify candidates, wrong invariants have to be sorted out. Static testing methods like the Boogie verification tool can only reach a verdict over the correctness of an invariant if they are able to prove it. Dynamic testing methods on the other hand are able to provide concrete counter examples for wrong invariants and can be much more effective at removing wrong invariants. Programs written in Boogie are not executable and traditional dynamic testing methods are thus not available. Instead, this thesis examines how symbolic execution engines like Boogaloo [13] and Symbooglix [12] can be used instead of traditional dynamic testing. With these tools the invariant inference algorithm is able to combine dynamic and static testing to identify the candidates invariant.
Figure 1.1 visualises how this approach fits into the context of recent research in the field of invariant inference techniques.

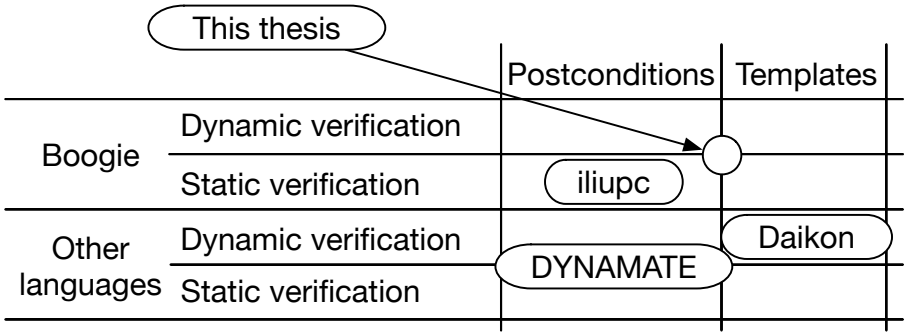


**Figure 1.1:** Context of the thesis compared with previous invariant inference techniques

The Daikon system [6] generates likely invariants for programs of multiple high level languages. Daikon executes the program, observes its effects and reports whether certain properties were true during the observed executions. Daikon uses a base of

templates to generates possible invariants.
The relationship between loop invariants and post-conditions has been explored in past work like the DYNAMATE tool [8], which uses a combination of dynamic analysis and static checking to generate loop invariants from post-conditions for Java programs. Something similar has even been done for the Boogie language in the research project *Inferring loop invariants using post-conditions* (iliupc) [7]. It presents methods and experiments on loop invariant inference from post-conditions for Boogie programs using static testing to identify candidate invariants.

## 1.1 Goals and challenges

The main goal of this thesis is to examine the possibilities of replacing traditional dynamic testing, in order to identify candidate invariants, with symbolic execution engines, like Boogaloo and Symbooglix. To answer the question how a combined approach of dynamic and static testing affects the capabilities to identify candidate invariants, an algorithm for invariant inference is implemented. The algorithm has to include a method to generate a large number of possible invariants from templates and post-conditions, a way to inject possible invariants into Boogie programs and has to invoke and interpret the output of the dynamic and static testing methods to identify candidate invariants. In order to answer the main question of the thesis, a detailed evaluation is necessary, which examines both, the invariants necessary and generated for different programs and the numbers of invariants removed by the different testing methods.
The capabilities of the implemented invariant inference algorithm rely heavily on the interaction with the symbolic execution engines Boogaloo and Symbooglix. These are research tools with a relatively short history and thus potentially buggy. It is also difficult to predict the efforts, necessary to build a framework to interact with the Boogie implementation, in order to inject invariants and to invoke the Boogie verifier for static testing.
Furthermore, the work with the Boogie language and the tools operating on Boogie programs will require to get to know the language and to get comfortable using it. Before the Boogie verification tool and the symbolic execution engines can be used in an automated setting, it will be important to get to know the tools and their peculiarities, which can be difficult due to a lack of documentation.

## 1.2 Structure of the thesis

The thesis starts with Chapter 2, which provides a short introduction into the features of the Boogie language used throughout this thesis and the tools used to test possible invariants. Next, Chapter 3 introduces the algorithm for invariant inference developed in this thesis, alongside a detailed look at its execution on an example program. Chapter 4 presents the results of the algorithm when run on a selection of Boogie files and analyses them, already initiating parts of the discussion, held in Chapter 5. In addition to a discussion on the results of the evaluation, Chapter 5

lists a number of questions, which came up during the work on this thesis and pose interesting topics for future work.

# 2

# Tools and Environment

## 2.1 Boogie

Boogie is the name of both an intermediate verification language and a verification tool. The combination of the two provides a layer on which to build program verifiers for other languages. Such verifiers can leave the difficult task of generating verification conditions to the Boogie verifier and only have to provide a front-end which translates into Boogie programs. The Boogie verifier can then transform from the intermediate representation into logical formulas which are attempted to prove by theorem provers, such as Z3 [5].

The Boogie language has both mathematical and imperative constructs. The mathematical ones consisting of types, constants, functions and axioms and the imperative ones of global variables, procedure declarations and procedure implementations. While the imperative components of Boogie are used to specify sets of execution traces, the mathematical components can be used to describe and constrain them.

The language supports features like parametric polymorphism, partial orders, logical quantification, non determinism, total expressions, partial statements and flexible control flow, using labels and goto statements.

Both the imperative and mathematical components underlie a type system, which consists of basic primitive types such as mathematical integers, floats and booleans and type constructors. One- and two-dimensional arrays can be represented by maps from mathematical integers to another type.

Mappings to Boogie programs from different programming languages, such as Spec# [3], C [4], Dafny [10], Java bytecode with BML [9] and Eiffel [14], are already existent or in development. This gives tools with Boogie programs as their input a large number of possible applications.

### 2.1.1 The Boogie specification language

This section gives a brief introduction into both the mathematical and the imperative constructs of the Boogie language needed for the examples in this thesis. A concise introduction to the Boogie language is given by Furia and Meyer [7, Section 4.2]. This introduction will follow that one closely, introducing the example Boogie procedure *max* along the way, which will also serve as an example for the invariant inference algorithm in Chapter 3. For more detailed information on the Boogie language, please consult the Boogie documentation [11].

### Expressions

The structure of Boogie expressions is similar to that of formulas in standard predicate logic. They consist of constants, variables, equality and arithmetic relations, boolean connectives, simple arithmetic operators, logic quantifiers and an ordering operator. The following expression states that one of the first $n$ elements in $a$ has the value $v$, that is, the value $v$ is contained by the array $a$ of size $n$.

```
(exists j: int :: 0 <= j && j < n && a[j] == v);
```

### Logic functions

For complex expressions or repeated use of them, logic functions are a means to encapsulate them. The expression from above defines the logic function *contains*, which will be used to specify the procedure *max* shown in Listing 2.1.

```
function contains(v:int,a:[int]int,n:int) returns (bool)
{ (exists j: int :: 0 <= j && j < n && a[j] == v) }
```

The logic function includes two of the build in types of Boogie, integer and boolean. Furthermore the array is represented by a map from integer to integer. The type integer represents mathematical integers and maps like these are therefore infinite. The number of elements of which a valid element is expected in the array is specified by $n$.

### Axioms

In addition to the function body, axioms are another way to give meaning to functions like *contains* and to provide the prover with additionally useful properties, like in the one in the following example. If the array $a$ contains the value $v$ in the first $n$ elements, it also contains the value $v$ in the first $n + 1$ elements.

```
axiom (forall v: int, a:[int]int, n:int :: contains(v, a,
    n) ==> contains(v, a, n+1));
```

### Procedures

A Boogie program is a collection of procedures, each of them consisting of a signature, a specification and (optionally) an implementation or body. The Listings 2.1 and 2.2 show excerpts of the Boogie program from Listing 6.1, which includes the procedure *max*. The program begins with another definition of a logic function, *upper_bound*. This function, together with *contains*, plays an important role in the specification of the procedure *max*. After that comes the signature of the procedure, followed by its specification. The signature gives the procedure a name and declares its in- and output arguments, in the case of *max*, $a$ and $n$ as input arguments, and *max* as an output argument.

```
function upper_bound(v:int,a:[int]int,n:int) returns (bool)
{ (forall j: int :: 0 <= j && j < n ==> a[j] <= v) }

procedure max(a:[int]int, n:int) returns (max:int);
 requires n > 0;
 ensures contains (max, a, n);
 ensures upper_bound (max, a, n);
```

**Listing 2.1:** Example Boogie procedure *max*, which returns the maximum of an integer array

### Specifications

The signature is followed by the specification of the procedure, which is a collection of contract clauses: pre-conditions, post-conditions and frame conditions. Pre-conditions are introduced by the keyword `requires` and are formulas which are required to hold upon procedure invocation. Post-conditions are introduced by the keyword `ensures` and are formulas that are guaranteed to hold upon successful termination of the procedure. The specification of *max* contains one pre-condition, which states that the length of the array has to be greater than 0. It also contains two post-conditions, which utilise the previously defined logic functions *contains* and *upper_bound*. The value returned by the procedure *max* has to be contained in the first $n$ elements of $a$ and has to be an upper bound for the first $n$ elements of $a$. The specification does not contain a frame condition, which would be marked with the keyword `modifies` and would contain a list of global variables modified by the procedure, to signalise possible side-effects of a call of procedure *max*.

### Implementations

The implementation of a procedure can either follow the signature directly in the procedure body, like in Listing 3.4, be separated as an `implementation`, like in the example in Listing 2.2 or not exist at all.
Every Boogie implementation begins with the declaration of its local variables. It follows a sequence of program statements. The syntax of most statements used in the implementations in this thesis is pretty straight forward and very close to notations in high level programming languages. While the Boogie language includes a larger set of types of statements, those used in this thesis can be broken down to the ones listed in a simplified syntax in Figure 2.1.
Naturally, when the aim is to infer loop invariants, all interesting programs will contain a loop, in Boogie only while loops. Hence, also this example contains a loop. The loop consists of three parts, the condition, the head and the body. While the loop condition and body are self explanatory, the loop head is the section which specifies the functionality of the loop, using loop invariants.
Note that the implementation does not contain a `return` statement. While it is possible to include such a statement at any point of the implementation to let the procedure return, it is optional to include one at the end of the implementation.

```
implementation max(a:[int]int, n:int) returns (max:int)
{
 var i : int;
 max := a[0];
 i := 1;
 while (i < n)
  invariant i <= n;
  invariant contains (max, a, i);
  invariant upper_bound (max, a, i);
 {
  if (a[i] > max) {
   max := a[i];
  }
  i := i + 1;
 }
}
```

**Listing 2.2:** Implementation of the Boogie procedure *max* from Listing 2.1 returning the maximum of an array

$$
\begin{aligned}
\text{Statement} &::= \text{Assertion} \mid \text{Modification} \mid \text{ConditionalBranch} \mid \text{Loop} \\
\text{Annotation} &::= \textbf{assert}\ \text{Formula} \mid \textbf{assume}\ \text{Formula} \\
\text{Modification} &::= \text{VariableId} := \text{Expression} \\
&\quad \mid \textbf{call}\ [\ \text{VariableId}^{+} :=\ ]\ \text{ProcedureId}\ (\ \text{Expression}^{*}\ ) \\
\text{ConditionalBranch} &::= \textbf{if}\ (\ \text{Formula}\ )\ \text{Statement}^{*}\ [\ \textbf{else}\ \text{Statement}^{*}\ ] \\
\text{Loop} &::= \textbf{while}\ (\ \text{Formula}\ )\ \text{Invariant}^{*}\ \text{Statement}^{*} \\
\text{Invariant} &::= \textbf{invariant}\ \text{Formula}
\end{aligned}
$$

**Figure 2.1:** Simplified abstract syntax of Boogie statements

The class of annotation statements is not represented in this example but is used later in the thesis, like in Listing 3.5. Annotations introduce checks at any program point by stating logic formulas. A program is only correct, if every assertion holds for every execution that reaches it. The property stated by an assumption can be postulated at that point of the program for every execution.

### 2.1.2 The Boogie verification tool

The Boogie verification engine proofs the conformance of a procedure's implementation against its specification. The tool parses the Boogie source file, generates verification conditions and then reports the outcome of feeding the verification conditions into a theorem prover, such as Z3. The outcome of such an attempt can be either successful or unsuccessful. While in the latter case, the tool provides some feedback on what parts of the specification or which annotations were not verified, it does not follow that these are wrong. Verification with Boogie is sound but incomplete. That means a verified procedure is always guaranteed to be correct, while

an unsuccessful verification attempt might simply be caused by limitations of the technology. Especially, when running Boogie with a large number of invariants, it can take quite a while to receive its verdict and every unverified invariant leaves the user with the question if there is an issue with the correctness of the invariant or if the incompleteness of the verifier caused the error.

## 2.2 The symbolic execution engine Boogaloo

Programs written in the Boogie language are not executable, which means that the only feedback on an implementation is given by tools like the Boogie verification engine. While the output of the Boogie verifier is great in the case of a successful verification, it is only of little use when the verification fails. The incompleteness of the verifier leaves the question whether the implementation is wrong or if Boogie is simply not powerful enough to verify a post-condition or another annotation. Boogaloo tries to give a hint on whether a post-condition or annotation is correct or not by trying to find a concrete counter example for it. To do this, Boogaloo creates small tests that fit the procedure's pre-conditions and symbolically executes the implementation. The calculated possible program states are checked against the post-conditions and annotations using SMT constraint solving. If this step leads to a contradiction with some annotation, the test serves as a counter example. Such a counter example acts as a prove that the annotation is wrong for this implementation and easily readable feedback on what changes have to be made. Generating concrete inputs for a non deterministic language like Boogie is by no means a trivial task and Boogaloo is not sound, meaning it is not able to find a counter example for every wrong annotation.
An alternative for a symbolic execution engine for Boogie is Symbooglix. Symbooglix was the main execution engine used in the early stages of the thesis but was replaced by Boogaloo since that one seemed to find counter examples faster.

# 3
# Methods

This chapter describes an invariant inference algorithm for automatic invariant detection and introduces the methods behind the algorithm. To generate all necessary invariants, different kinds of invariants have to be considered. Most of the times, rather simple invariants are needed to specify the bounds of for example an iterator in the loop. Such an invariant could look as follows, to specify the upper bound of the iterator $i$.

```
invariant i <= n;
```

Invariants like this one can easily be generated from a template (Section 3.3) and are likely to state a necessary property for many applications of such an algorithm. Other invariants, like the one in the next example can be more complex and more unique to one application.

```
invariant contains (max, a, i);
```

Especially when the invariant includes logic functions, it is not sufficient to write a template for each one of them. The invariant finder therefore includes invariant generation from post-conditions as well (Section 3.4). In the next sections, first, the general algorithm will be introduced and then, step by step, the different parts necessary for its implementation are discussed in closer detail. The example of *max*, which was already introduced in Chapter 2, will guide through the different steps of the algorithm, to illustrate their effects. The complete input for *max* and its configuration for the algorithm, as well as the annotated output file can be found in Listings 6.1, 6.2 and 6.3.

## 3.1 Invariant finder

The algorithm that incorporates all the functionality developed for this thesis, is called the invariant finder. The algorithm takes a Boogie program as its input and then iterates over all procedures in the program, annotating the ones that require it with candidate invariants. This section roughly depicts the algorithm behind the invariant finder, in pseudo code in Figure 3.1, and provides a road map through the remainder of this chapter, which will go into more detail on the individual parts of the implementation. In the end, this section will also explain certain design choices in closer detail.
The first and last actions of the invariant finder are the input and output operations, which make use of the Boogie parser (Section 3.2). The invariant finder also

uses temporary files to execute Boogaloo on, which also requires functions from the Boogie parser. If a procedure can not be verified automatically, the invariant finder assumes that this is due to a lack of invariants and proceeds to generate invariants from templates (Section 3.3) and from post-conditions (Section 3.4). Since the generation of possible invariant itself takes virtually no time, most of the time the invariant finder is busy checking possible invariants using Boogaloo and Boogie. A lot of the generated invariants do not make any sense and can be sorted out with small counter examples by Boogaloo. This is why, to cover the sheer amount of generated invariants, Boogaloo is run in multiple threads at once. In order to always use the highest possible number of threads, the invariants are generated at once and then injected in waves. After a wave of invariants has been injected, it is time to filter out wrong invariants using dynamic and static testing (Section 3.5). This continues until the procedure can be verified automatically or until the method runs out of possible invariants. In the latter case the invariant finder can be run again using a different configuration (Section 3.7). At the end or whenever the program contains too many invariants at once another kind of invariant filtering comes into play. Redundancy checks remove as many invariants as possible while maintaining the same degree of automatic verification for the post-conditions (Section 3.6).

```
Data: Boogie program
Result: Boogie program annotated with loop invariants
foreach Procedure p in P do
    if p can not be verified then
        Generate invariants from templates;
        Generate invariants from post-conditions;
        while p can not be verified do
            Add wave of invariants;
            Check invariants using Boogaloo;
            Remove wrong invariants;
            Try to verify p;
            Remove not verified invariants;
        end
        Remove redundant and trivial invariants;
    end
end
Print annotated Boogie program;
```

**Figure 3.1:** Pseudo code of simplified structure of the invariant finder

## 3.2 Boogie parser

The implementation of the Boogie verification tool is open-source and the C# source code is available on GitHub [2]. To work as close as possible with the actual structure of Boogie programs and to be able to invoke the Boogie verifier directly on them, the clear choice was to use the parser and type checker included in the Boogie implementation. The Boogie parser returns a single program from a Boogie file.

Such a program is the input to the type checking and the verification condition generation. The invariant finder takes the type checked program as its input. Boogie programs, even for short Boogie files, quickly become overwhelming and complex in their structure. The interesting part needed for this thesis is luckily fairly simple. A program contains a list of top level declarations. These can be for example logic functions, axioms, global variable declarations or procedures and implementations. The last two are the most interesting ones for the invariant inference. A procedure contains its name, input and output arguments and specification. Similarly an implementation also contains its name and input and output arguments. In addition it contains a list of blocks. These blocks are where one can find the loop of an implementation, split into, amongst others, the loop head, which contains a list of assert commands, and the loop body. The list of assert commands are the loop invariants for the loop.
The parsed Boogie program does not contain comments and because each procedure is split into its declaration and implementation, can also have a different appearance when it is serialised using the print function.

## 3.3 Invariants from templates

The generation of invariants from templates consists of multiple steps. Firstly, a template file is processed. Listing 3.1 shows the template file which is used to generate invariants stating that one integer is greater than of equal to another integer. The first line of the file is written as a Boogie comment and contains a list of variable declarations. The most important part of the file is the invariant in the second line, which is written in the syntax of invariants in the Boogie language. The template is parsed using a slightly adapted version of the Boogie parser, which returns an assert command instead of a program. Next, the variables in the template have to be replaced by appropriate candidates. Natural candidates to replace the variables $X$ and $Y$ are global variables and the input and output parameters, as well as the local variables of the procedure.

```
//@ Variables X:int Y:int
invariant X >= Y;
```

**Listing 3.1:** The template `GreaterEquals.bpl`

The variables in the template are replaced by all possible combinations of these candidates, which quickly results in a large number of invariants. For the example of *max* this leads to a total of 9 possible invariants being generated from the template in Listing 3.1.

```
invariant n >= n;
invariant n >= i;
invariant n >= max;
invariant i >= n;
...
invariant max >= max;
```

While it is not required for $max$, it can also be useful to include constants like 0, 1 or $true$ and $false$ as candidates for the replacement of the variables of a template. In fact, a variable in the template invariant can be considered as an expressions, which allows for replacements with not only variables and constants but complex expressions. The invariant $X < Y$ could for example be replaced by something like $(0) < (i + (n + 1))$. This example uses the constants 0 and 1 as well as the two integer variables $i$ and $n$. The template variable $Y$ is replaced by the expression $(i + (n + 1))$. The number of possible replacements thus becomes countless and the possible invariants have to be limited to a reasonable set. Section 3.7 explains to what degree the invariant finder allows for configurations of this set.
An additional mechanism, which is implemented to control the number of generated invariants, is the replacement of the same variable in the template with the same expression in the one invariant. In the example *invariant* $X + X == Y$; all occurrences of $X$ have to be replaced with the same expression in the same invariant.
For testing purposes, a small collection of templates was handwritten. This collection can be extended easily with new templates. However, while there are invariants of common use like the $X >= Y$ example, more complex templates are likely to only fit certain applications. Such templates can not be used for other Boogie programs, because they most probably will not make sense outside of their context. Choosing the right base of templates to apply for every Boogie program, while offering a variety of templates to fit the needs of different programs, becomes a challenge in itself and is where the next section, on invariants from post-conditions, chimes in.

## 3.4 Invariants from post-conditions

While handwritten templates are a good source for simple invariants, most of the times invariants are more complex. Considering the invariants from the implementation of $max$ in Listing 2.2, it becomes apparent that, while even these invariants could be generated from templates, they would require more specific templates. These would result in the generation of a lot of useless invariants for most other procedures. This is the point where the connection between the required invariants to verify post-conditions and the post-condition themselves comes into play. Listing 3.2 shows one of the post-conditions of $max$ and one of its loop invariants, which closely relates to it.

```
ensures contains (max, a, n);
...
invariant contains (max, a, i);
```

**Listing 3.2:** Post-condition from Listing 2.1

This observation originates back to the paper by Furia and Meyer [7], which also presents techniques to modify post-conditions efficiently to generate likely invariants. The invariant finder does something similar, by creating invariant templates from post-conditions. Listing 3.3 shows the template file which would yield the same internal representation of an invariant template as the generation from the post-condition of $max$, shown in Listing 3.2.

```
//@ Variables max:int a:[int]int n:int
invariant contains (max, a, n);
```

**Listing 3.3:** Invariant template from post-condition in Listing 3.2

While the paper suggests techniques to modify the post-conditions to retrieve possible invariants, this algorithms, as of now, implements the same simple replacement technique which is also used for the handwritten templates. For the example of *max*, the invariant finder generates 18 possible invariants from its post-conditions, which are indicated in Listing 2.1.

```
invariant contains(n, a, n);
invariant contains(n, a, i);
...
invariant contains(max, a, max);
invariant upper_bound(n, a, n);
invariant upper_bound(n, a, i);
...
invariant upper_bound(max, a, max);
```

During the creation of the template, multiple occurrences of one variable in the post-condition yield only one variable in the template, as the following example shows.

```
ensures (exists j: int :: 0<=j && j<n && a[j]==max);
...
//@ Variables idVar0:int i:int n:int a:[int]int max:int
invariant (exists boundVar0: int :: identVar0<=i&&i<n&&a[i]==max);
```

This again helps to limit the number of possible invariants which can be generated from this template. It is likely that an invariant candidate which relates to this post-condition has the same pattern. Of course this also prevents the invariant finder from finding certain possible invariants which could be required.
The bound variable $boundVar0$ does not show up in the variable list because its type is already specified in the invariant itself and as a bound variable is not to be replaced. Instead $boundVar0$ itself becomes a replacement candidate for the variables in the template.

## 3.5 Filtering invariants

Despite the measures to limit the number of possible invariants, the invariant finder can quickly generate a large number of possible invariants. These are injected by simply extending the list of assert commands of the loop head. Many of the possible invariants however, are wrong and have to be removed at some point of the algorithm.
The invariant finder uses Boogaloo to dynamically find counter examples for wrong invariants (Section 3.5.1) and Boogie to statically select candidate invariants (Section 3.5.2). The combination of these two tools is instrumental for the success of the

invariant finder. Boogie is sound but incomplete, meaning it can attempt to verify an invariant, but if it fails that does not have any meaning for the correctness of the invariant. Boogaloo on the other hand is complete but unsound. If Boogaloo finds a counter example for an invariant, it is a prove that the invariant is wrong. However, if Boogaloo does not find a counter example, this does not imply that the invariant is correct.

The two tools are chained together. First, as many wrong invariants as possible are removed using Boogaloo. These can be discarded directly as they can not be candidate invariants and it would not make sense to attempt to prove something which is proven to be wrong. Next, Boogie is used in an attempt to prove the remaining invariants. All proven invariants are candidate invariants. The not provable invariants have to be removed, as they might be wrong and can influence the verification result of other wrong invariants, resulting in wrong candidate invariants.

### 3.5.1 Disproving invariants

The filtering of invariants using Boogaloo before selecting candidate invariants with Boogie enables the invariant finder to test much larger amounts of possible invariants at once. Boogie does not scale well with an increasing number of invariants and neither does Boogaloo actually. When a certain number of invariants is reached, Boogie's executions seem to take forever and often only report a small number of invariants which it could not prove. After these invariants have been removed, Boogie needs to be executed again because other invariants might only have been proven on the assumption that the not provable invariants from the previous run were correct. It is not an option to attempt to prove each invariant independently because more complex invariants often rely on each other and can only be proven in a compound. Boogaloo executions on large numbers of invariants usually only report a single counter example, which applies for a small number of invariants, when Boogaloo is actually able to disprove so many more invariants.

However, finding a counter example for one invariant is not dependent on any other invariant. Hence, the process of disproving multiple invariants can be parallelised into one Boogaloo instance for each invariant. Disproved invariants are discarded immediately and invariants for which no counter example could be found within a given timeout are marked and will skip later Boogaloo tests.

For the Boogaloo execution, the program has to be serialised into one temporary Boogie file per invariant. Because the serialisation and repeated parsing of the program results in changes to the tokens of the invariants, which are used to identify invariants, the original program has to be reparsed, from the temporary file, as well.

For the example of *max* the following 9 possible invariants are disproven by Boogaloo.

```
invariant max >= n;
invariant max >= i;
invariant contains(n, a, n);
invariant contains(n, a, max);
invariant contains(n, a, i);
invariant contains(max, a, max);
invariant contains(i, a, n);
invariant contains(i, a, max);
invariant contains(i, a, i);
```

### 3.5.2 Proving invariants

The Boogie verifier operates on the remaining set of possible invariants and tries to select candidate invariants which it can prove. Unlike for the Boogaloo execution, the Boogie verifier is executed directly on the Boogie program. However, the Boogie verifier makes irreversible changes to the program, which result in the Boogie verifier not accepting the same program again. The development of a cloning method for Boogie programs was attempted during this thesis but not successful and thus also the Boogie verifier uses serialisation and reparsing to clone Boogie programs.
The clone of a program is fed into the Boogie verifier, which returns a verification result. The verification result is then evaluated and depending on the evaluation there are three possible next steps for the invariant finder.

- **The procedure is verified completely** This does not only mean, that all added invariants are correct, but also that they are sufficient to verify the procedure's post-conditions. The invariant inference stops for this procedure and all remaining invariants are candidate invariants.
- **All invariants are verified** This means that all remaining invariants are correct and candidate invariants. However, they are not sufficient to verify the procedure's post-conditions and more invariants need to be tested.
- **An invariant is not verified** While it is unknown whether the invariant is actually wrong, it is removed at this point. The whole process of statically testing the invariants has to be repeated, because some other invariant might have only been proven based on the now removed invariant.

For the example of *max* the following 7 invariants are selected as candidate invariants

```
invariant n >= n;
invariant n >= i;
invariant i >= i;
invariant max >= max;
invariant contains(max, a, n);
invariant contains(max, a, i);
invariant upper_bound(max, a, i);
```

while Boogie was not able to prove these 10 invariants, which are thus removed.

```
invariant n >= max;
invariant i >= n;
invariant i >= max;
invariant upper_bound(n, a, n);
invariant upper_bound(max, a, n);
invariant upper_bound(n, a, max);
invariant upper_bound(max, a, max);
invariant upper_bound(i, a, n);
invariant upper_bound(n, a, i);
invariant upper_bound(i, a, max);
```

## 3.6 Redundancy checks

A large number of candidate invariants state trivial properties or are redundant because of other candidates. Hence these candidates are not needed to verify the post-condition and should be removed from the implementation, to not clutter the file and consume additional time during the execution of Boogie. This section describes the redundancy checks implemented for the invariant finder. They are executed whenever the invariant inference for a procedure ends. The invariant finder can be furthermore configured to execute them in between single waves to reduce the number of invariants that have to be checked by Boogie at the same time (Section 3.7). The redundancy checks implement a fairly simple method to prove that an invariant is redundant or trivial, introduced by Galeotti [8, Section 4.3]. Listing 3.4 shows the structure of a Boogie procedure which shall be checked for redundant invariants. All parts that are not needed for the redundancy checks, such as most parts of the implementation, are left out. The procedure contains a loop which is annotated by the two invariants $i$ and $j$.

```
procedure p (in: int) returns (ret: int)
 requires r;
 modifies m;
 ensures e;
{
 var l: int;
 ...
 while (c)
  invariant i;
  invariant j;
 {
  ...
 }
 ...
}
```

**Listing 3.4:** Example Boogie procedure for redundancy check

Listing 3.5 shows the modified procedure $p$ to check whether the invariant $j$ is trivial or implied by invariant $i$.

```
procedure p (in: int) returns (ret: int)
 requires r;
{
 var l: int;
 assume i;
 assert j;
}
```

**Listing 3.5:** Modifies Boogie procedure for redundancy check

This modification could be performed analogously to check whether an invariant is implied by a compound of other invariants.
For the *max* example the redundancy check at the end of the algorithm removes the 3 candidate invariants

```
invariant n >= n;
invariant i >= i;
invariant contains(max, a, n);
```

and results in the final set of candidate invariants

```
invariant n >= i;
invariant max >= max;
invariant contains(max, a, i);
invariant upper_bound(max, a, i);
```

In some cases a candidate invariant which is removed in this step might give the deciding hint to the verifier on which property to use to prove a post-condition or another invariant. However, this can only happen when the redundancy checks are performed between two waves and at this point the only other option is to end the invariant finder all together because the Boogie execution times become too long.

## 3.7 Configuration

The number of invariants generated grows quite fast with the number of replacement variables, templates, constants and operations. Because of scalability issues, discussed in closer detail in Chapter 4, the invariant finder can be configured for the program it is executed on. Listing 3.6 shows the `XML` file for the base configuration. The configurable features of the invariant finder are the handwritten templates, the constants and the operations. Furthermore, the wave size and thus the maximum number of Boogaloo threads can be adjusted, together with a maximum timeout for Boogaloo and a threshold of injected invariants after which a redundancy check is performed. The key `templates` has a list of semicolon separated template names as its value. If one chooses to not use a certain template or wants to add a special template for a program this list can easily be shortened or extended by the name of the respective template. The key `constants` is pretty self explanatory and its value

can be shortened or extended analogously to the value of `templates`. As of right now, the only two operations, to extend variables in the replacement candidates with, are `-1` and `+1`. This means that every variable in the replacement candidates `x`, upon insertion into a template, will be considered as `x`, `x-1` and `x+1`, depending on the semicolon separated operation list in the configuration file. With the value of `multithreading` one can basically adjust the number of invariants added in each wave. This results in more or less Boogaloo threads. During the experiments conducted for this thesis, this number did not have a huge impact, as the system already posed a bottleneck for less than 200 threads. The value of `maxinvariants` determines after every how many injected invariants a redundancy check is performed. This becomes necessary for larger amounts of candidate invariants. If the invariant finder already injected several thousand possible invariants there might be up to a few thousand candidate invariants. These invariants are proven again and again during each wave of invariants that is injected. If the number of invariants in the implementation grows over a few thousand, the execution time of Boogie can grow up to multiple minutes to report a single invariant which it is not able to prove. To keep the invariant finder running, it is thus important to do redundancy checks in between waves, the frequency of which can be adjusted according to how many candidate invariants are expected from the possible invariants. The last setting, `boogalootimeout`, sets the number of seconds the invariant finder waits for each Boogaloo instance to return a counter example. If Boogaloo finds a counter example, it usually finds it in under one second. On the benchmark system there was no success in attempting to optimise the invariant finder with the timeout value but it was vital to include a timeout.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
<appSettings>
<add key="template_directory"
value="/Users/timon/Education/masterthesis/templates/" />
<add key="templates"
value="Implies;Smaller;Greater;SmallerEquals;GreaterEquals;Equals" />
<add key="constants"
value="0" />
<add key="ops"
value="" />
<add key="multithreading"
value="200" />
<add key="maxinvariants"
value="1000" />
<add key="boogalootimeout"
value="5" />
</appSettings>
</configuration>
```

**Listing 3.6:** Base configuration for the invariant finder

## 3.8 Implementational details

This section gives a very basic overview of the architectural choices for the invariant finder and discusses some challenges that arose from it or shaped the architecture in the first place.
All of the main functionality of the invariant finder is written in C# in the visual studio solution invariantFinder. Python scripts enable the use of the configuration files and assist the interaction between the C# implementation and the invocations of Boogaloo.
All Boogie implementation features such as the Boogie parser and verifier are included directly in the C# implementation. A modified Boogie implementation is used which has added functions that act as the interface for the invariant finder to read and write Boogie programs and to perform program verification tasks. Since the code of the Boogie parser is generated and as such scarcely documented, the modifications, such as adapting the parser to also process invariant templates, required a lot of trial and error and debugging and were quite time consuming.
The attempt to use both, static and dynamic testing methods to identify candidate invariants required a certain level of familiarity with the capabilities of the symbolic execution engines and the Boogie verifier and how they report their results. For the static verification part the invariant finder executes the verification section of the Boogie verifier directly via the parsed and annotated Boogie program. While it would have been easier to simply execute the whole verification tool on the program, it is much more attractive to interact directly with the verification section of the Boogie verifier, to receive the most detailed feedback possible on verification errors. Even though this part is better documented than the parser section, it still required a lot of work to make the verification part work on its own.
Another difficulty presented itself in the fact that the verifier makes a lot of irreversible changes to the program. Section 3.5.2 already mentioned that the attempt to write a cloning method for Boogie programs has been made but was unsuccessful. The parsing of Boogie files, which results in the initialisation of a new program triggers a lot of functions and checks which posed a huge obstacle when attempting to create a deep clone of a Boogie program.
In the end the development of a framework to support interaction with Boogie programs in the sense of understanding their structure, injecting invariants at the right place and being able to invoke the testing methods consumed the majority of the time spent on the implementation of the algorithm.

# 4
# Results

To evaluate the capabilities of the invariant finder, a benchmark, including carefully selected Boogie programs, was executed on the method. This chapter goes through the selection process of the different test cases and afterwards presents the results, investigating the different properties of the programs and analysing their effect on the results of the invariant finder. This helps to understand the limitations of the algorithm and how it can be configured to achieve better results for certain programs.

## 4.1 Benchmark selection

The evaluation set consists of 15 Boogie programs, each containing one procedure. Every procedures contains exactly one loop, as the functionality of the invariant finder is limited to procedures containing exactly one loop right now. The example procedures lie within a reasonable level of complexity for the invariant finder. Most of them can be annotated sufficiently by the algorithm, while others will reach its limits. In order to examine how different kinds of specifications and implementations affect the capabilities of the invariant finder, several of the 15 Boogie programs are variations of other programs in the benchmark. There are a total of three Boogie programs containing procedures to calculate the maximum of an array. Analogously three Boogie programs contain procedures to calculate the minimum of an array. Three Boogie programs contain procedures to calculate the square sum of an integer. The benchmark is executed in three steps. Firstly, each boogie program is annotated on is own using the default configuration of the invariant finder. Next, given the awareness of the required invariants to verify the post-conditions of each procedure, the Boogie programs are annotated once more using an optimised configuration. These examples give a good idea on the functionality of the invariant finder, but are no good representation for real world Boogie examples. The handwritten Boogie program *max*, for example, consists of less than 30 lines of code. Compare that to several thousand lines of code one receives when translating a similar Dafny program into a Boogie program. This is why the benchmark contains a third part which tries to simulate some scalability examples by joining the single Boogie files into larger ones. The first one contains all three variations of the *max* procedure and the second one includes all procedures of this benchmark.
All of the procedures require invariants in order for Boogie to verify their post-conditions. However, none of them are annotated with invariants yet. A set of invariants, which is sufficient to verify all the post-conditions of a procedure, is provided as comments in each of the input files to evaluate the output of the invariant

finder. Listing 4.1 shows the commented invariants in the loop head needed to verify the post-conditions of the procedure *max* in the program `max_pred_v1.bpl`.

```
while (i < n)
 //invariant i <= n;
 //invariant contains (max, a, i);
 //invariant upper_bound (max, a, i);
```

**Listing 4.1:** Needed invariants for max_predǐ.bpl

The invariants in this program contain the already introduced logic functions *contains* and *upper_bound*. Hence, the Boogie program will also include their definition.
All benchmark programs, their optimised configuration, as well as the two large test cases and the output of the invariant finder for all test cases can be examined in a bitbucket repository [1].

#### Reference system

The benchmark is executed on a 2014 MacBook Pro (Retina 13-inch, Late 2013) with a 2,4 GHz Intel Core i5 processor and 8 GB of 1600 MHz DDR3 RAM, running macOS High Sierra Version 10.13.4. During the multithreaded Boogaloo execution the system will be a bottleneck and better runtimes might be achieved by executing the method on a more powerful machine. However, a method like the invariant finder is only attractive if it is accessible on even low tier systems and does not take a lot of time to execute.

## 4.2 Benchmark results

For each step of the benchmark, the invariant finder is executed on single Boogie programs containing one or more Boogie procedure. The output of is summarised in different multiple tables for the different steps. Table 4.1 contains the results for the single procedures with the default configuration, Table 4.2 for the single procedures with optimised configurations and Tables 4.3 and 4.4 for the Boogie programs with multiple procedures.
The rows in the tables contain the name of the Boogie program which is the source for the procedure tested, in the column PROC. The column INV and CND give the minimum number of invariants which Boogie needs to verify the procedure and the number of candidate invariants the invariant finder inferred for the procedure. Be aware that a higher number of candidates than necessary invariants does not imply that the procedure can be verified. It is easy to generate invariants that are true but do not aid the verification of a procedure's post-conditions. The next two columns state the numbers of invariants generated from templates and from post-conditions. Information about how successful the different methods were at filtering out invariants is given in the next three columns. DYN gives the number of invariants dynamically sorted out using Boogaloo, STC the number of invariants which Boogie was unable to prove and RED the number of invariants removed because of redundant or too trivial properties. The last two columns specify whether

the candidates are sufficient to verify the post-conditions of the procedure and give the time it took invariant finder to annotate the program. In Tables 4.3 and 4.4 the execution time is given in the caption of the tables instead. During each of the steps of the benchmark, some programs are presented and special features and their effects on the output of the invariant finder analysed.

### 4.2.1 Single procedures with default configuration

Table 4.1 summarises the output of the invariant finder when executed on a single Boogie program at a time with the default configuration, which can be seen in Listing 3.6. With only the constant 0 and no operations, the default configuration adds almost no replacement candidates to the list of parameters and variables of the program. However, this minimal set of replacement candidates combined with a short list of templates is enough to infer a sufficient set of invariants for most of the test cases.

| PROC | INV | CND | FTL | FPC | DYN | STC | RED | PC | T |
|---|---|---|---|---|---|---|---|---|---|
| count | 2 | 6 | 134 | 125 | 72 | 92 | 30 | YES | 0:23 |
| max | 3 | 6 | 89 | 1250 | 717 | 223 | 393 | YES | 1:18 |
| max_pred_v1 | 3 | 6 | 89 | 32 | 50 | 36 | 29 | YES | 0:07 |
| max_pred_v2 | 2 | 3 | 89 | 16 | 50 | 26 | 26 | YES | 0:06 |
| min | 3 | 6 | 89 | 1250 | 757 | 196 | 380 | YES | 1:16 |
| min_pred_v1 | 3 | 6 | 89 | 32 | 54 | 32 | 29 | YES | 0:07 |
| min_pred_v2 | 2 | 3 | 89 | 16 | 49 | 26 | 26 | YES | 0:06 |
| square | 3 | 8 | 133 | 25 | 64 | 55 | 31 | NO | 0:08 |
| square_easy_v1 | 2 | 6 | 88 | 16 | 30 | 46 | 22 | NO | 0:06 |
| square_easy_v2 | 2 | 5 | 113 | 125 | 48 | 108 | 39 | YES | 0:12 |
| small_numbers | 2 | 6 | 134 | 125 | 136 | 31 | 27 | YES | 24:51 |
| even | 2 | 6 | 64 | 27 | 19 | 43 | 23 | YES | 0:13 |
| is_even | 2 | 6 | 63 | 9 | 19 | 27 | 20 | YES | 0:10 |
| merge | 6 | 10 | 257 | 98 | 98 | 134 | 113 | NO | 0:22 |
| binary_search | 4 | 12 | 254 | 2744 | 1108 | 789 | 1089 | NO | 3:55 |

**Table 4.1:** Running all procedures separately with the default configuration

#### Max and min

The *max* example is one of the examples alongside which the invariant finder was developed. As such it exists with three different sets of post-conditions which can be seen in Listing 4.2. All *max* examples share the same implementation and their post-conditions could be verified using the same set of invariants. However, with the main source of invariants in the default configuration being the post-conditions, they end up with different candidate invariants.
As Table 4.1 shows, there are a lot more invariants generated for the first specification of *max*. This is the consequence of the larger number of variables in the post-conditions and the different number of post-conditions compared to the other

specifications. The results show that the majority of the generated invariants are sorted out by Boogaloo, which is how the invariant finder is able to finish the annotation in less than 80 seconds.

```
max.bpl:
 ensures (exists i : int :: 0<=i && i<n && a[i]==max);
 ensures (forall i : int :: 0<=i && i<n ==> a[i]<=max);
max_pred_v1.bpl:
 ensures contains (max, a, n);
 ensures upper_bound (max, a, n);
max_pred_v2.bpl:
 ensures is_max (max, a, n);
```

**Listing 4.2:** Different ways to specify the procedure *max*

The three versions of the *min* example are, simply put, the exact same programs as the *max* examples with all >= replaced by <= and vice versa. This does not have any affect on the invariant generation. The results show that the invariant finder generates the same number of invariants from templates and post-conditions. The difference lies in the numbers of invariants sorted out by Boogaloo and Boogie. While the rough proportions are still in tact, there is a significant difference in invariants sorted out by Boogaloo. The number of invariants not provable by Boogie is dependent on the number of invariants disproved by Boogaloo, since it is not possible to verify a wrong invariant. The fact that Boogaloo and Boogie can perform significantly different in really similar invariants and contexts is merely an observation here, as it does not affect the final outcome of the invariant finder. The later examples of larger Boogie files will however uncover more critical differences in the performance of Boogaloo and Boogie when the task of disproving and verifying invariants becomes more difficult.

### Square

The procedure *square* calculates the square $n \cdot n$ of an integer $n$ and exists in three versions as well. Contrary to *max* and *min* however, the different versions have different specifications and different implementations. The first version builds the square by calculating the square sum of the iterator $i$ during each iteration of the loop until $i$ reaches $n$. The two versions of *square_easy* calculate $i \cdot n$ during each iteration of the loop, where $i$ is again the iterator which becomes $n$ in the end. Listing 4.3 shows the post-conditions for the different procedures and a selection of the necessary invariants to verify the post-conditions.
For the first example, the invariant finder can easily generate the first invariant from the post-condition. The second invariant however, can not be generated from the pattern of the post-condition. Even with the right constants and operations, the invariant finder will only replace the two occurrences of $n$ in the post-condition with the same expressions in a generated invariant. Similarly it is not possible to generate the necessary invariant for the second example, since it would require a replacement of $n$ with $i$ and with $n$ in the same invariant. The third example is not practical, as

it introduces an additional parameter $x$ to the *square_easy* procedure which must be equal to $n$.

```
square.bpl:
 ensures r == n*n;
 invariant r == i*i;
 invariant x == 2*i + 1;
square_easy_v1.bpl:
 ensures r == n*n;
 invariant r == i*n;
square_easy_v2.bpl:
 ensures r == x*n;
 invariant r == i * n;
```

**Listing 4.3:** The post-conditions and a selection of the required invariants from the *square* examples

Because in this case the post-condition can be written with two different variables in the product. From that the invariant finder is able to generate the invariant in the last line.

An easier way to achieve this would be to add a template with a body like $x == y*z$ or to allow the replacement of multiple occurrences of the same variable with different expressions, which will be discussed in Section 5.1.

### Small_numbers

The test case *small_numbers* presents a strange and unexpected result. The file contains a procedure which is comfortably in the target subset of the Boogie language for which the invariant finder should return a fully annotated loop. The procedure has been properly annotated be previous versions of the invariant finder but in the latest version, the invariants are injected in a certain order that results in a problem with Boogie. For one temporary state of the generation, the execution of Boogie takes an unusually long time to stop. Further investigation unveiled that it is the combination the following three invariants, which causes this.

```
invariant r <= max * i;
invariant r <= r * r;
invariant r <= i * n;
```

The investigation on why this happens does not fit into the context of this thesis but it is a good example for how working with verification tools like Boogie can be challenging, even for simple examples. In the optimised execution the number of invariants added at a time has been set to 1, in the configuration file, in order not to trigger this edge case. The execution time can be expected to be slightly shorter when using multithreading properly.

### 4.2.2 Single procedures with optimised configuration

Table 4.2 summarises the output of the invariant finder when executed on a single Boogie program at a time with a configuration which is optimised to its procedure. Except for the procedures which were not annotated sufficiently with the default configuration, which are discussed at some point in this chapter, the changes in the configuration file mainly consist of removed templates and constants. Most procedures actually only required one template and no constants at all.

| PROC | INV | CND | FTL | FPC | DYN | STC | RED | PC | T |
|---|---|---|---|---|---|---|---|---|---|
| count | 2 | 5 | 16 | 64 | 6 | 66 | 3 | YES | 0:17 |
| max | 3 | 6 | 16 | 1250 | 683 | 203 | 374 | YES | 1:22 |
| max_pred_v1 | 3 | 4 | 9 | 18 | 9 | 11 | 3 | YES | 0:02 |
| max_pred_v2 | 2 | 3 | 9 | 9 | 9 | 4 | 2 | YES | 0:01 |
| min | 3 | 6 | 16 | 1250 | 723 | 174 | 363 | YES | 1:20 |
| min_pred_v1 | 3 | 4 | 9 | 18 | 13 | 7 | 3 | YES | 0:02 |
| min_pred_v2 | 2 | 3 | 9 | 9 | 9 | 4 | 2 | YES | 0:02 |
| square | 3 | 6 | 810 | 81 | 578 | 286 | 21 | YES | 0:46 |
| square_easy_v1 | 2 | 3 | 36 | 9 | 0 | 38 | 4 | YES | 0:04 |
| square_easy_v2 | 2 | 3 | 16 | 64 | 0 | 67 | 10 | YES | 0:06 |
| small_numbers | 2 | 3 | 16 | 64 | 39 | 9 | 5 | YES | 0:10 |
| even | 2 | 3 | 4 | 18 | 0 | 18 | 1 | YES | 0:07 |
| is_even | 2 | 3 | 4 | 6 | 0 | 6 | 1 | YES | 0:06 |
| merge | 6 | 22 | 8384 | 72 | 266 | 1292 | 6876 | NO | 23:47 |
| binary_search | 4 | 48 | 288 | 22464 | 7044 | 2235 | 7673 | YES | 27:13 |

**Table 4.2:** Running all procedures separately with optimised configurations

The executions of the invariant finder with optimised settings for the programs *merge* and *binary_search* are the first ones that result in larger amounts of possible invariants. It is interesting to see how 8456 possible invariants for *merge* and 17001 possible invariants for *binary_search* are tested in surprisingly similar time. This observation can be explained by the large number of possible invariants that are disproved by Boogaloo for *binary_search* compared to the relatively small number for *merge*.

#### Merge

The Boogie procedure *merge* implements the merge operation needed for algorithms like merge sort. It remains the only procedure which is not annotated sufficiently by the invariant finder, even with an optimised configuration. While it would have been easy to write the necessary templates to fit the needs of this procedure, that would not really be the point of automatic invariant inference. To have a tool choose the invariants without the need for additional input, the required invariants must fit some sort of pattern. The creation of a new handwritten template is only justified if it is interesting to a larger set of programs. Otherwise such a template will help to solve a special case but will explode the execution time for simpler

procedures which do not require this template. Furthermore, coming up with the right template already requires a good idea of what the required invariant might look like. Listing 4.4 shows the invariants necessary to verify the single post-condition of *merge*, $sorted(b, start, n)$.

```
invariant (forall j, k: int :: l<=j && j<k && k<mid ==> a[j]<=a[k]);
invariant (forall j, k: int :: r<=j && j<k && k<n ==> a[j]<=a[k]);
invariant l<mid ==> (forall j: int :: start<=j&&j<i==>b[j]<=a[l]);
invariant r<n ==> (forall j: int :: start<=j && j<i ==> b[j]<=a[r]);
invariant n-i == (mid-l) + (n-r);
invariant sorted (b, start, i);
```

**Listing 4.4:** Necessary invariants for *merge*

The invariants for this example are more complex than what is needed for for example *max* and only one of the invariants relates to the post-condition. So how can the invariant finder guess the remaining invariants? It can not. Finding the right templates to generate these invariants is a task just as tedious as coming up with the invariants in the first place. Add the time needed to run the invariant finder over and over again and this becomes completely unpractical. The only change made in the configuration for *merge* was the addition of the template from Listing 4.5.

```
//@ Variables A:int B:int C:int D:int E:[int]int
invariant (forall boundVar0, boundVar1: int :: A <= B && B < C && C
   < D ==> E[B] <= E[C]);
```

**Listing 4.5:** Invariant templates from ForAll2ElemSmallerEquals.bpl

This template can be applied to other interesting implementations that are related to sorting algorithms as well and is thus not too specific to *merge*. It could be included in a collection of templates which are interesting for such procedures. When that type of procedure is identified, one could simply activate the generation from the templates of this collection. In the case of *merge* however, the addition of this template just caused a higher number of possible invariants which did not even lead to interesting candidate invariants. While the 8456 invariants surely included some invariants close to the first two invariants from Listing 4.4, Boogie was not able to verify them without the other required invariants. This further classifies that the complexity of this example lies beyond the capabilities of the invariant finder.

### Binary_search

The default configuration was not able to find the required invariants for *binary_search* and at first it seemed like the steps to configure the invariant finder for this example would take more effort than they actually did. The only change that was made was to remove unnecessary templates and to add the operation +1. This resulted in every replacement candidate $x$ being extended to the expression $(x+1)$ and with that the expression $(high+1)$ was added to the replacement candidates. The expression $(high+1)$ is necessary for the required invariant for *binary_search*, which can be seen in Listing 4.6.

```
invariant !contains (a, fromindex, low, key);
invariant !contains (a, high + 1, toindex, key);
```

**Listing 4.6:** Selection of invariants necessary for *binary_search*

But how did this change result in the generation of sufficient invariants? The post-conditions of *binary_search* do not include the right pattern for these invariants. Very interestingly this did not pose a hurdle for the generation, as the invariant finder simply generated the invariants in Listing 4.7.

```
invariant low==low ==> !contains(a, fromIndex, low, key);
invariant toIndex==toIndex ==> !contains(a, high + 1, toIndex, key);
```

**Listing 4.7:** Selection of candidates generated for *binary_search*

It becomes apparent quickly that the first expression of the implications can simply be replaced by *true* and they are thus equivalent to the invariants from Listing 4.6. While these are not the candidates someone would have written by hand, they serve their purpose in both aiding the verification of *binary_search*'s post-conditions as well as making the user aware of what invariants were necessary.

### 4.2.3 Programs with multiple procedures

To examine how the invariant finder performs on larger Boogie files, the invariant finder was executed on two Boogie files containing more than one procedure. While the results documented in Table 4.3 show the outcomes when executing the invariant finder on a Boogie file containing all variations of the procedure *max*, Table 4.4 shows the results when including all programs except *small_numbers*, which is not considered due to the unusually long execution time of Boogie, in one Boogie file. Unfortunately the results show that the invariant finder does not scale well. Compared to the single file execution, when the invariant finder was able to annotate all three variations of *max* sufficiently, the test case with all variations of *max* in one file only results in the verification of two of them. In the Boogie program containing all the procedures only one of the variations of *max* gets annotated sufficiently.

| PROC | INV | CND | FTL | FPC | DYN | STC | RED | PC |
|---|---|---|---|---|---|---|---|---|
| max | 3 | 6 | 89 | 1250 | 717 | 223 | 393 | YES |
| max_pred_v1 | 3 | 6 | 89 | 32 | 50 | 46 | 19 | YES |
| max_pred_v2 | 2 | 4 | 89 | 16 | 42 | 46 | 13 | NO |

**Table 4.3:** Running all *max* procedures in one program. T=1:32

The main difference lies in the number of invariants disproved by Boogaloo and the number of invariants not verified by Boogie. The general trend seems to be that, with growing program length, the number of invariants disproved by Boogaloo decreases. On the other hand, more invariants are sorted out using Boogie, because they can not be verified. A portion of these invariants are surely ones that were missed by Boogaloo and are actually wrong. However, some of the not proved invariants must also be correct invariants which Boogie was able to verify in smaller programs.

For both observations there is no easy explanation as the calculations of Boogaloo as well as those performed by Boogie are highly complex and, as was already visible in the *max* and *min* example, there is no way of knowing in advance which invariants Boogaloo will be able to disprove and which invariants can be proven by Boogie. The invariant finder attempts to narrow the task down as much as possible for both Boogaloo and Boogie. Both tools are executed with only one target procedure and in the case of Boogaloo such a procedure contains only one invariant at a time. Still, the added complexity from the bigger files has a significant influence on the capabilities of Boogaloo and Boogie.
As a result of the smaller number of wrong invariants disproved by Boogaloo the runtime is expected to go up. The effect of Boogie not being able to verify as many invariants, results in a smaller number of programs being annotated sufficiently because more candidate invariants are wrongfully removed.
In the execution of single procedures, the procedure *max* resulted in 717 disproved and 223 not verified invariants. The amount of invariants that are sorted out because the invariant finder has to assume that they are wrong grew for that particular example from 952 to 987. The decreased amount of invariants disproved by Boogaloo might not only affect the execution time but also the outcome of Boogie, since Boogie has to check more invariants at the same time. While, from this evaluation set, it can not be pinpointed whether the problem for Boogie is the larger number of invariants or the larger context, it is apparent that scalability is an issue with Boogie and the verification tasks should be kept as simple as possible.
In the example of all procedures, as seen in Table 4.4, the different capabilities of Boogie verifying the invariants of the very similar procedures *max* and *min* becomes even more apparent than in the separate tests. Now, not only the numbers of invariants removed by dynamic and static methods differ but even the number of procedures annotated sufficiently. Only one variation of *max* is annotated sufficiently while two variations of *min* are annotated sufficiently.

| PROC | INV | CND | FTL | FPC | DYN | STC | RED | PC |
|---|---|---|---|---|---|---|---|---|
| count | 2 | 6 | 134 | 125 | 72 | 103 | 19 | YES |
| max | 3 | 6 | 89 | 1250 | 647 | 340 | 346 | NO |
| max_pred_v1 | 3 | 6 | 89 | 32 | 43 | 54 | 18 | YES |
| max_pred_v2 | 2 | 4 | 89 | 16 | 33 | 53 | 15 | NO |
| min | 3 | 6 | 89 | 1250 | 576 | 465 | 292 | NO |
| min_pred_v1 | 3 | 6 | 89 | 32 | 30 | 65 | 20 | YES |
| min_pred_v2 | 2 | 3 | 89 | 16 | 26 | 58 | 18 | YES |
| square | 3 | 4 | 133 | 25 | 47 | 93 | 14 | NO |
| square_easy_v1 | 2 | 6 | 88 | 16 | 30 | 51 | 17 | NO |
| square_easy_v2 | 2 | 5 | 113 | 125 | 48 | 132 | 15 | YES |
| even | 2 | 0 | 64 | 27 | 34 | 41 | 16 | NO |
| is_even | 2 | 1 | 63 | 9 | 18 | 39 | 14 | NO |
| merge | 6 | 10 | 257 | 98 | 98 | 152 | 95 | NO |
| binary_search | 4 | 12 | 254 | 2744 | 1108 | 789 | 1089 | NO |

**Table 4.4:** Running all procedures in one program. T=11:02

# 5
# Conclusions

This chapter continues the discussion, which already started together with the evaluation in Chapter 4, and gives suggestions for future work on this topic, followed by a short conclusion.

## 5.1 Discussion

The observations during the evaluation in Chapter 4 suggest that the combination of dynamic and static testing to identify candidate invariants can be quite powerful. The two approaches complement each other as was already described in Section 3.5. Especially when the invariant finder generates a lot of wrong invariants, the addition of dynamic testing helps a lot with the performance of the identification of candidate invariants. This results in the invariant finder actually performing quite well on the evaluation set, considering the simple replacement techniques used to generate possible invariants from templates and post-conditions.
Overall the algorithm for invariant inference developed in this thesis should not be considered more than a prototype. Its framework however, could be deployed to include more complex techniques to replace the variables from templates and single terms from post-conditions. Similar to the concept of injecting and testing possible invariants in waves, the generation of invariants could start with simple expressions to replace the terms of the templates with and grow more complex with each wave of generation. Section 3.3 described how, in order to limit the number of possible invariants, the same variable in a template is always replaced with the same expression in the same invariant. This is one obvious example where a second wave of invariants which does not include this restriction could increase the number of candidate invariants.
The main reason behind restricting the number of possible invariants at some was the shier amount of possible invariants generated from complex hand written templates. The next paragraph will explain how the number of templates required for the successful annotation in the benchmark was kept quite low in the end, which is why removing the restrictions in waves could be a good approach for this set of programs.
The invariant finder considers handwritten templates and post-conditions as sources for the generation of possible invariants. During the longest part of the thesis, the only source were the handwritten templates and a number of them was created alongside the algorithm. With the integration of templates from post-conditions, most of the hand written templates became obsolete and the templates from post-conditions proved themselves much more effective. Especially with the benchmark

results, the question arises how many templates are actually needed for most applications, since in the end most candidate invariants came from templates created from post-conditions. Furthermore the tests with optimised configurations showed that most of the times only the template $X >= Y$ was used. Considering that most of the implementations in the benchmark have the loop condition $i < n$, the question arises whether the loop condition could act as a source for possible invariants instead. Together with more advanced techniques to replace the variables in the templates, this could possibly result in a faster generation of the desired candidate invariants.

If the generation of possible invariants can be improved to include more candidate invariants and less wrong invariants, the effects of dynamic testing on these sets of invariants have to be reevaluated. Furthermore it would be interesting to investigate in more detail if the dynamic testing primarily improves the selection of candidate invariants if a lot of wrong invariants are generated or if the dynamic method is especially effective at finding counter examples for certain types of invariants. It stands to reason that it could be easier to find a counter example for a logic formula which contains a for all quantifier compared to an exists quantifier. It would be interesting to examine whether Boogaloo shows such patterns in practice.

The results of the benchmark in this thesis also do not allow for a final conclusion on which limitation of the testing methods were due to the tools and which might have been due to the system, the benchmark was performed on. Especially when Boogaloo was running, the system nearly froze, as all its computational power was used up by Boogaloo. Considering how well the dynamic testing can be parallelised, it would be interesting to see if the testing scales well on more powerful systems.

The Boogie files that can be handled by the invariant finder are limited to procedures which include at most one loop. While it would be interesting to expand the target programs to more complex structures with multiple or nested loops, the benchmark programs were already complicated enough to show up the limits of the algorithm. It would still be interesting to investigate whether there exists a certain subset of Boogie programs which has multiple loops and could benefit from the invariant finder.

A major effort during the implementation of the invariant finder was the direct interaction with parts of the Boogie implementation, some of the difficulties with which are described in Section 3.8. While the integration of Boogie functionalities is robust enough to handle the examples from the benchmarks and even more complex files, attempts to run real world examples like programs translated from Dafny code were unsuccessful. The integration of the Boogie functionalities poses a limitation for the processing of such programs and it is difficult to estimate how much time would have to be invested into understanding the Boogie implementation to improving the integration, which is why it was not further attempted in this thesis.

Section 2.2 mentions an alternative symbolic execution engine in Symbooglix. The method was used in the early stages of the development of the invariant finder. However, it turned out that, while Symbooglix might be able to find counter examples for more wrong invariants, Boogaloo was able to find a counter example, for the invariants for which it could find one, significantly faster. This property made it the preferred choice for the invariant finder.

## 5.2 Future work

The future work suggestions that initiate in this work basically split into three parts: continue the development of the invariant inference techniques of the invariant finder, examine the possible invariants rejected by Boogaloo in closer detail, and test the alternative symbolic execution engine Symbooglix in the same setting.
The invariant inference technique developed in this thesis serves well as a prototype for invariant generation from different sources and candidate invariant selection through a combination of dynamic and static testing. The discussion section describes how it would be interesting to implement more advanced replacement methods for the variables in the templates. Time constraints also did not allow to fit the invariant finder to more complex Boogie files and procedures containing multiple loops. The invariant finder could be developed further to be able to process such files and investigate if there is a possible application for more complex programs.
The combination of dynamic and static testing for the selection of invariant candidates should be tested in a benchmark with more information on the invariants. What percentage of the invariants is actually wrong and how many of these can Boogaloo disprove. Furthermore, is there a certain pattern of wrong invariants for which Boogaloo is more likely to find a counter example?
This thesis was not able to achieve practical results using Symbooglix to disprove possible invariants. More investigation into the options of and possibly improved settings for Symbooglix would be interesting.

## 5.3 Conclusions

This thesis resulted in an invariant inference algorithm for the Boogie intermediate verification language which uses a combination of dynamic and static testing to identify candidate loop invariants. The algorithm's process of generating possible invariants is very simple and results in a lot of wrong invariants but was already run successful on small examples. The evaluation of the algorithm shows that the combination of dynamic and static testing can be powerful at quickly identifying candidate invariants out of large sets of invariants. The use of Boogaloo for the dynamic testing part can boost the number of possible invariants the algorithm can evaluate to up to 1000 invariants per minute.

# Bibliography


[1] Bitbucket repository containing the benchmark for the invariant finder. https://bitbucket.org/lapawczykt/invariant-finder-benchmark/src/master/.

[2] Boogie github repository. https://github.com/boogie-org/boogie.

[3] 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. https://www.microsoft.com/en-us/research/wp-content/uploads/2005/01/krml160.pdf.

[4] Shaunak Chatterjee, Shuvendu K Lahiri, Shaz Qadeer, and Zvonimir Rakamarić. A reachability predicate for analyzing low-level software. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 19–33. Springer, 2007. https://link.springer.com/content/pdf/10.1007/978-3-540-71209-1_4.pdf.

[5] 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. https://link.springer.com/content/pdf/10.1007/978-3-540-78800-3_24.pdf.

[6] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007. https://homes.cs.washington.edu/~mernst/pubs/daikon-tool-scp2007.pdf.

[7] Carlo A Furia and Bertrand Meyer. Inferring loop invariants using postconditions. *Fields of Logic and Computation*, 6300:277–300, 2010. http://bugcounting.net/pubs/yg70-post.pdf.

[8] Juan P Galeotti, Carlo A Furia, Eva May, Gordon Fraser, and Andreas Zeller. Inferring loop invariants by mutation, dynamic analysis, and static checking. *IEEE Transactions on Software Engineering*, 41(10):1019–1037, 2015. http://bugcounting.net/pubs/tse15-dynamate.pdf.

[9] Hermann Lehner and Peter Müller. Formal translation of bytecode into boogiepl. *Electronic Notes in Theoretical Computer Science*, 190(1):35–50, 2007. https://www.researchgate.net/profile/Peter_Mueller14/publication/223702915_Formal_Translation_of_Bytecode_into_BoogiePL/links/53ee631e0cf26b9b7dc972bd.pdf.

[10] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial In-*

*telligence and Reasoning*, pages 348–370. Springer, 2010. http://homepage.divms.uiowa.edu/~tinelli/classes/181/Spring10/Papers/Lein10.pdf.

[11] Rustan Leino. This is boogie 2. Microsoft Research, June 2008. https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/.

[12] Daniel Liew, Cristian Cadar, and Alastair F Donaldson. Symbooglix: A symbolic execution engine for boogie programs. In *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*, pages 45–56. IEEE, 2016.

[13] Nadia Polikarpova, Carlo A Furia, and Scott West. To run what no one has run before: Executing an intermediate verification language. In *International Conference on Runtime Verification*, pages 251–268. Springer, 2013. http://bugcounting.net/pubs/rv13.pdf.

[14] Julian Tschannen, Carlo A Furia, Martin Nordio, and Bertrand Meyer. Verifying eiffel programs with boogie. *arXiv preprint arXiv:1106.4700*, 2011. https://arxiv.org/pdf/1106.4700.pdf.

# 6
# Appendix

```
function contains(v:int,a:[int]int,n:int) returns (bool)
{ (exists j: int :: 0 <= j && j < n && a[j] == v) }

function upper_bound(v:int,a:[int]int,n:int) returns (
   bool)
{ (forall j: int :: 0 <= j && j < n ==> a[j] <= v) }

procedure max(a:[int]int, n:int) returns (max:int);
 requires n > 0;
 ensures contains (max, a, n);
 ensures upper_bound (max, a, n);

implementation max(a:[int]int, n:int) returns (max:int)
{
 var i : int;
 max := a[0];
 i := 1;
 while (i < n)
  invariant i <= n;
  invariant contains (max, a, i);
  invariant upper_bound (max, a, i);
 {
  if (a[i] > max) {
   max := a[i];
  }
  i := i + 1;
 }
}
```

**Listing 6.1:** The Boogie file `max_pred_v1_short.bpl`. It includes the invariants needed to verify the post-conditions. These need to be commented out before executing the invariant finder

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
<appSettings>
<add key="template_directory"
value="/Users/timon/Education/masterthesis/templates/" />
<add key="templates"
value="GreaterEquals" />
<add key="constants"
value="" />
<add key="ops"
value="" />
<add key="multithreading"
value="200" />
<add key="maxinvariants"
value="1000" />
<add key="boogalootimeout"
value="5" />
</appSettings>
</configuration>
```

**Listing 6.2:** Invariant finder configuration for the Boogie program from Listing 6.1

```
implementation max(a:[int]int, n:int) returns (max:int)
{
 var i: int;
 max := a[0];
 i := 1;
 while (i < n)
  invariant n >= i;
  invariant max >= max;
  invariant contains(max, a, i);
  invariant upper_bound(max, a, i);
  //invariant {:DYN} max >= n;
  //invariant {:DYN} max >= i;
  //invariant {:DYN} contains(n, a, n);
  //invariant {:DYN} contains(n, a, max);
  //invariant {:DYN} contains(n, a, i);
  //invariant {:DYN} contains(max, a, max);
  //invariant {:DYN} contains(i, a, n);
  //invariant {:DYN} contains(i, a, max);
  //invariant {:DYN} contains(i, a, i);
  //invariant {:STC} n >= max;
  //invariant {:STC} i >= n;
  //invariant {:STC} i >= max;
  //invariant {:STC} upper_bound(n, a, n);
  //invariant {:STC} upper_bound(max, a, n);
  //invariant {:STC} upper_bound(n, a, max);
  //invariant {:STC} upper_bound(max, a, max);
  //invariant {:STC} upper_bound(i, a, n);
  //invariant {:STC} upper_bound(n, a, i);
  //invariant {:STC} upper_bound(i, a, max);
  //invariant {:RED} n >= n;
  //invariant {:RED} i >= i;
  //invariant {:RED} contains(max, a, n);
 {
  if (a[i] > max) {
    max := a[i];
  }
  i := i + 1;
 }
}
```

**Listing 6.3:** The annotated implementation of the procedure *max* from Listing 6.1 when the invariant finder is executed with the configuration from Listing 6.2