VERIFICATION OF PROGRAMS WITH Z3

By

EWA ROMANOWICZ, B.SC.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree of

Master of Science
Department of Computing and Software
McMaster University

MASTER OF SCIENCE (2010) McMaster University
(Department of Computing and Software) Hamilton, Ontario

TITLE: Verification of programs with Z3

AUTHOR: Ewa Romanowicz, B.Sc. (York University)

SUPERVISOR: Dr. Ryszard Janicki

NUMBER OF PAGES: viii, 74

Abstract

Fixing the errors in programs is usually very labor-intensive and thus an expensive task. It is also known to be prone to human error thus not fully reliable. There have been many methods of program verification developed, however they still require a lot of human input and interaction throughout the process. There is an increasing need for an automated software verification tool that would reduce human interaction to the minimum. Satisfiability Modulo Theories (SMT) solvers, a series of SAT-solvers such as Z3 looked initially to be a proper and easy to use tool. Its syntax is fairly uncomplicated and it seems to be quite efficient.

In this thesis, Z3 is used to find loop invariants, prove some properties of concurrent programs written in Owicki-Gries style and prove some properties of recursive programs. It appears that — in general — Z3 does not work as well as expected in all areas to which it was applied.

Acknowledgements

I would like to thank my supervisor, Dr. Ryszard Janicki, for his help in the last four years, for his guidance and extreme patience during my struggles to juggle school and work commitments. I appreciate his faith that this thesis will be completed successfully, his sharing of knowledge and his enthusiastic approach to my work.

Thanks to all the professors at McMaster and York Universities that I had a chance to interact with and who made a positive impact on my quest toward completion of this thesis and this degree.

Thanks to my former colleagues who helped the classes to be more fun and enjoyable.

Special thanks to Daniel Zingaro for being a good friend and for being optimistic and confident in my work, even when I wasn't. I would like to also thank him for allowing me to waste precious time when I needed to procrastinate and be lazy.

Also, special thanks to Ian Anderson for providing final improvement comments, editing the finished thesis paper, being very supportive and confident in my work in the last steps of the process.

Finally I would like to thank my Mom for always being there for me and scolding me about my school work when I totally deserved it.

Contents

Chapter 1

Introduction

One of the most crucial problems in a program is making sure that it carries out its intended function [24]; because of that, Program Verification is an important part of software design [38] and is the subject of numerous studies. It allows for the detection of logical errors in early stages of design [2] and lowers the costs of programming error [24]. According to Dijkstra (1976), a programming language should be thought of first and foremost as an algorithm-oriented system of mathematical notation and then as something to be run on a machine. When programs are proved correct, it is possible to place a great reliance on outcomes of the programs, and predict with their properties with confidence [24].

In the late sixties and early seventies, a technique for verification and analysis of computer programs based on a calculus of relations (or equivalently predicates) was proposed by R. W. Floyd [20] and quickly exploited by others [6], [30]. Despite many theoretical and methodological advantages (it emphasizes calculation instead of proving as in the more popular method of C. A. R. Hoare [24]), the technique has never become widely accepted. This may be because of the huge amount of symbolic computations that need to be performed for even relatively simple cases [9], even though in most cases these are rather easy (though lengthy) predicate calculus computations.

The B-method for program verification became popular and there are many versions of HOL theorem provers that can assist us with proving the correctness of programs, however there became a need for a faster prover that would process huge programs with hundreds of variables with minimal user interaction and within the shortest possible time [4], [22].

The situation has dramatically changed today, as we have very powerful tools supporting symbolic computation such as Maple [32] and Mathematica [43], and relatively easy to use theorem provers such as Simplify [15] or Z3 [34]. The problem is still non-trivial, as the most general cases are undecidable, but for many practical cases an efficient solution seems to be feasible. Experiments with Maple were presented in [8], [9], the theoretical foundation of the approach in [8] and a tool in [9], [47]. The main idea is to represent programs not as relations or predicates, but as symbolic recurrence relations; this of course imposes many restrictions, on the structure of the programs that can be handled. Despite those restrictions, the method is still applicable to a large variety of numerical programs. This thesis will represent programs as predicates (in special formats) and try to use theorem provers as a medium for analysis.

In 1962, M. Davis, G. Logemann and D. Loveland from New York University [12] explained how satisfiability (satisfiability modulo theories (SMT)) can be used as the mechanism for proving first order logic formulas. They mentioned in their article an example that generated more than 500 quantifier-free lines and used just a couple of minutes to calculate it to be valid. Highly efficient SAT-solvers were introduced as a result of the excellent performance of the algorithm [4].

A wide range of SMT solvers were developed, each introducing an improvement to the proceeding version. The more popular SMT solvers that have been developed include: ABLogic, Barcelogic, Yices, Simplify and Z3. The common goal for all the SMT Solvers is to provide fast and reliable program verification with minimal user input.

Simplify was by developed by Compaq as an efficient SMT solver the input for which is a first-order formula with quantifiers [14]. It provides counter examples for invalid formulas. Microsoft also recently developed the Z3 solver [34]that introduced, among other things, improvements over Simplify theorem prover by implementing enhanced E-matching that increases efficiency of solving quantifiers. The features of these solvers include the ability to work with free function and predicate symbols, real and integer arithmetic, bitvectors and arrays.

In theory the correctness of more practical programs can be proved as a result of the advances in SAT-solving technology [36]. This thesis will explore the different types of programs and their verification using Z3 SMT solver, including recursion. The main and common characteristic between these programs is that the recursion block depends on the value of the previous iterations. We will observe the behavior of

the SMT solvers when the program iterates n times, where n is an arbitrary number of iterations and where iteration n depends on the previous iteration, $n - 1$.

Chapter 2 will explain program verification and how proof obligations are generated. The use of different automatic theorem provers will be explained in Chapter 3. Loop invariants and program verification by calculating relations will be explained in Chapter 4, together with examples of using Z3 in verifying program correctness. Chapter 5 will explain; using examples, how concurrent programs can be proved using SMT solvers and the Owicki/Gries method. Chapter 6 will explain how SMT solvers can be used to prove recursive programs. Conclusion and discussion on future work will appear in the last chapter.

Chapter 2

Program Verification

2.1 Background and History

During development of computer programs, there is significant effort put into ensuring that the finished program is working as designed, that it produces reliable results and that it terminates for any possible allowed input [27] [44] [42].

Since the beginning of software engineering, testing has been the most popular way of verifying programs to be correct. Testing ensures that for a given set of test inputs, the output of the program produces expected results [7]. There have been many testing techniques developed, but it has been discovered that testing does not provide the required level of assurance that the program is correct for any possible input [42]. It is unclear if the selected inputs from often infinite possibilities assure that every bug in the code will be revealed [42]. It is said that testing can show errors, however it can never show the absence of errors [45].

While testing is still widely used in the industry, there has been significant research towards finding a definite and reliable program verification method that would allow verification of program correctness for any possible case.

Formal verification has been introduced as an alternative to testing. It allows representation of a behavior of a program in mathematical terms which then can be proved [42]. It has also been a subject of numerous publications [31].

Back in 1947, the first idea of proving program correctness was outlined by Goldstein and von Neumann. They explained a possible proof technique that used vectors of program variables after several execution steps. The idea, however, failed for logically complex and large programs [45].

Floyd developed a method from the idea introduced by Goldstein and von Neumann [20]. He formalized proof of correctness, equivalence and termination by introducing assertions in the program flow with the use of program flow charts. The essence of his approach indicated that the correctness of the output follows directly from the correctness of the input [45].

Hoare expanded Floyd's ideas and the use of axioms in reasoning about computer programs [24]. Just as Floyd, Hoare claimed that the validity of a program depends on the values of variables before the start of the program. He defined the postcondition as the property of the values after the program is executed and the precondition as the value taken by the variables before the program is initiated. This definition is known as a Hoare triple and uses the following notation

$$\{P\}Q\{R\}$$

which means that "if the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion" [24].

Using the new notation, Hoare redefined Floyd's idea of program verification that used flow charts and was able to construct formal proofs of simple programs [24]. These formal proofs are the basis for program verification that is done today.

2.2 Axiomatic Approach To Program Verification

Axiom of Assignment

The assignment statement is of the form

x:=f

where x is a variable and f is an arbitrary programming language expression. In order for the assertion $P(x)$ to be true as a postcondition, $P(f)$ has to be true as a precondition. In the new Hoare notation, the axiom is expressed as follows:

$$\vdash P_0\{x := f\}P$$

which means that P_0 is obtained from P by substituting f for all occurrences of x.

Rules of Consequence

Taking the "Axiom of Assignment" as a basis of programs, Hoare introduced rules of inference in order to be able to deduct more theorems. The "rule of consequence" was defined as follows:

$$If \vdash P\{Q\}R \; and \vdash R \supset S \; then \vdash P\{Q\}S$$

$$If \vdash P\{Q\}R \; and \vdash S \supset P \; then \vdash S\{Q\}R$$

If execution of program Q ensures that R is true, then any assertion logically impled by R is also true. If P is known to be a precondition for Q in order to obtain R, then any assertion that logically implies P is true.

Rule of Composition

A program is a sequence of statements that execute in order. The "rule of composition" states that the postcondition of the first set of statements is a precondition to the set of the statements following the first. If the second part of the program produces correct results, then the whole program is correct if the precondition of the first set of statements is satisfied [24]. This has been formalized using the following notation:

$$If \vdash P\{Q_1\}R_1 \; and \vdash R_1\{Q_2\}R \; then \vdash P\{Q_1; Q_2\}R$$

Rule of Iteration

The last rule — the "rule of iteration" — has been defined for fragments of programs such as S, that are executed until the condition B is false.

$$\text{while B do S}$$

The formalization includes an assertion P that is always true before and after the statements in S are completed.

$$If \vdash P \wedge B\{S\}P \; then \vdash P\{while \; B \; do \; S\}\neg B \wedge P$$

The rules were later extended to define recursive procedures, functional calls in programs, coroutines and unconditional jumps [45].

2.3 Weakest Precondition

The rules and definitions introduced by Hoare [24] did not guarantee program termination. In 1975, Edsger Dijkstra tightened Hoare's definition of preconditions by introducing a definition of the weakest preconditions [16].

$$wp(S, R)$$

The weakest precondition as per Dijkstra guarantees that after executing statements S the program will terminate and satisfy the post-condition R. The properties of wp were defined as follows: (taken from [16]).

1. For any S, we have for all states: $wp(S, F) = F$ (the so-called Law of the Excluded Miracle).

2. For any S and any two post-conditions, such that for all states $P \Rightarrow Q$, we have for all states: $wp(S, P) \Rightarrow wp(S, Q)$.

3. For any S and any two post-conditions P and Q, we have for all states $(wp(S, P) \wedge wp(S, Q)) = wp(S, P \wedge Q)$.

4. For any deterministic S and any post-conditions P and Q, we have for all states $(wp(S, P) \vee wp(S, Q)) = wp(S, P \vee Q)$.

Knowing the program statements S, $wp(S, R)$ can be derived from any post-condition R.

The program statements have been formalized by Dijkstra [16] using guarded commands, in order to be able to derive weakest preconditions easily.

A guarded command is a building block for alternative and repetitive constructs such that the boolean expression has to evaluate to true before the statements that follows can be executed.

<guarded command> ::= <guard> →<guarded list>
<guard> ::= <boolean expression>
<guarded list> ::= <statement> |; <statement> |
<guarded command set> ::= <guarded command> □ <guarded command set>
<alternative construct> ::= if <guarded command set> fi
<repetitive construct> ::= do <guarded command set> od

<statement> ::= <alternative construct> | <repetitive construct> | "other statements"

where { } denotes zero or more instances.

Using the above notation, Dijkstra defined a way of obtaining weakest preconditions to be as follows:

Skip Statement

Weakest precondition of an empty statement is defined as

$$wp("skip", R) = R$$

Assignment Statement

For assignment $x := E$ the weakest precondition is defined as:

$$wp("x := E", R) = R_E^x$$

Where R_E^x is R with all occurrences of x replaced by E.

Consecutive Statements

For statements S_1 and S_2 such that they appear consecutively in the program and are separated by ; , the weakest precondition can be expressed as:

$$wp("S_1; S_2", R) = wp(S_1, wp(S_2, R)).$$

Alternative Statement

The alternative construct of syntax:

$$if\ B_1 \rightarrow SL_1 \ \square ... \square\ B_n \rightarrow SL_n\ fi;$$

the weakest precondition can be obtained as follows:

$$wp(IF, R) = (BB\ and\ (\forall i : 1 \leq i \leq n : B_i \Rightarrow wp(SL_i, R)))$$

where BB denotes

$$(\exists i : 1 \leq i \leq n : B_i)$$

With respect to the weakest preconditions, the following theorem has been derived:

THEOREM 1: From $(\forall i : 1 \leq i \leq n : (Q \text{ and } B_i) \Rightarrow wp(SL_i, R))$ *for all states we can conclude that* $(Q \text{ and } BB) \Rightarrow wp(IF, R)$ *holds for all states.*

Also, when t denotes an integer function defined on state space and $wdec(S, t)$ is a weakest precondition such that S guarantees to terminate and that t is decreased by at least one, the following is true:

THEOREM 2: From $(\forall i : 1 \leq i \leq n : (Q \text{ and } B_i) \Rightarrow wdec(SL_i, t))$ *for all states we can conclude that* $(Q \text{ and } BB) \Rightarrow wdec(IF, t)$ *holds for all states.*

Repetitive Statement

The repetitive statement have been defined by Dijkstra as:

$$do\ B_1 \rightarrow SL_1 \Box ... \Box B_n \rightarrow SL_n\ od$$

Taking variable k as number of iterations the following can be defined:

$$H_k(R) = (R \text{ and non } BB) \text{ for } k = 0 \text{ and for } k > 0\ H_k(R) = (wp(IF, H_{k-1}(R)) \text{ or } H_0(R))$$

Due to the fact that k is an induction variable the following theorems can be proved using induction:

THEOREM 3: If we have for all states $(P \text{ and } BB) \Rightarrow (wp(IF, P) \text{ and } wdec(IF, t)$ *and* $t \geq 0)$ *we can conclude that we have for all states* $P \Rightarrow wp(DO, P \text{ and non } BB)$.

THEOREM 4: From $(P \text{ and } BB) \Rightarrow wp(IF, P)$ *for all states, we can conclude that we have for all states* $(P \text{ and } (wp(DO, T)) \Rightarrow wp(DO, P \text{ and non } BB)$.

In the above theorems, P is referred to as "the invariant relation" and t "the variant function" [16].

2.4 Proof Obligations

Djikstra and Feijen [17] explained that the program should be looked at as a state transition where each statement transitions from one state of the program to another. Based on the state transition a functional specification can be defined as the relation between the initial and final state of the program in four points:

1. the declaration of local variables;
2. the precondition, traditionally put in braces (following the Hoare triple notation);
3. the program statement;
4. the postcondition, also put in braces;

Each state transition should be proved individually by following simple automated rules called "proof obligations". Combination of all the subsequent statements can assure the correctness of the state change from program precondition to the program postcondition. The proof obligations were defined as follows:

The Postulate of skip

The skip statement as a state change can be defined as $[x : int \ \{P\}skip\{Q\}]$ and requires the following proof:

$$P \Rightarrow Q$$

The Postulate of Assignment

The assignment statement as a state change is defined as $[x, y : int \ \{P\}x := E\{Q\}]$ and requires the following proof:

$$P \Rightarrow Q^x_E$$

Where Q^x_E denotes that every occurrence of x in Q is replaced by E.

The Postulate of Concatenation

The concatenation of statements as a state change is defined as $[x : int\ \{P\}S_0; S_1\{Q\}]$ can be proven by separately proving each of the concatenated statements such that there is a predicate H that:

$$[x : int\ \{P\}S_0\{H\}] \text{ and } [x : int\ \{H\}S_1\{Q\}]$$

The Postulate of the Alternative Statement

The alternative statement as a state change is defined as:

$[x : int$
$\{P\}$
$\quad if\ B_0 \rightarrow S_0$
$\quad \square\ B_1 \rightarrow S1$
$\quad fi$
$\{Q\}]$

can be proven by ensuring that:

$$P \Rightarrow B_0 \vee B1$$

and each of the statements within the alternative statement with respect to the initial precondition can be shown correct by proving the following:

$$[x : int\ \{P \wedge B_0\}S_0\{Q\}]$$

$$[x : int\ \{P \wedge B_0\}S_0\{Q\}]$$

The Postulate of the Repetitive Statement

The repetitive statement as a state change is defined as:

$[x : int$
$\{P\}$
$\quad do\ B_0 \rightarrow S_0$
$\quad \square\ B_1 \rightarrow S1$
$\quad od$

$\{Q\}]$

can be proven by ensuring that there is a predicate H (called an invariant) and an integer function vf (called a variant function) such that all the following holds: The invariant holds before the loop

$$P \Rightarrow H$$

On every iteration the invariant holds before and after the iteration, and value of vf decreases on every iteration.

$$|[x : int\{H \wedge B_0 \wedge vf = VF\}; S_0\{H \wedge vf < VF\}]|$$

$$|[x : int\{H \wedge B_1 \wedge vf = VF\}; S_1\{H \wedge vf < VF\}]|$$

On each iteration where either guard B_0 or B_1 is true, the value of vf is greater than 0.

$$H \wedge (B_0 \vee B_1) \Rightarrow vf \geq 0$$

In the case where neither of the guards is true, the postcondition should be achieved.

$$H \wedge \neg B_0 \wedge \neg B_1 \Rightarrow Q$$

Chapter 3

Automatic theorem provers

The use of the method of generating proof obligations allowed predicate calculus to prove program correctness for sequential programs [19]. This method has been widely used and implemented as a part of many other methods and in proving non sequential programs. Construction of these proofs, however, was extremely difficult, especially with large programs. The disadvantage of the verification and analysis of computer programs based on a calculus of relations is the amount of symbolic computation that it requires [9].

Provers that are able to prove logical expressions have been developed. The B-method for program verification became popular and many versions of HOL theorem provers were created, such as HOL-light or Isabele HOL, that can assist with proving the correctness of programs. These provers however still required lots of manual input as proof obligations had to be generated manually, or the prover required extensive, manual guidance through the proof.

There became a need for a faster prover that would process huge programs with hundreds of variables with minimal user interaction and within the shortest possible time [4].

3.1 DPLL

It was noticed that the quantifiers caused most of the problems with program verification and that there were many proof procedures available. These procedures were able to confirm validity of the formula, however when the formula was invalid it involved seeking "forever" [13]. Wang and Gilmore were able to develop programs that use the

quantification theory procedures, however these programs were only able to validate simple formulas and run into serious problems with more complicated examples [13].

In 1960, M. Davis and H.Putnam described a uniform procedure for quantification theory which worked with complicated formulas successfully [12] [13]. In the algorithm, a quantifier-free formula is derived by replacing existential quantifiers in a prenex form formula with a function symbol. It has been noted that that this process does not affect consistency [13]. Using the DPLL (Davis-Putnam-Logemann-Loveland) algorithm, the quantifier-free formula is checked for consistency by applying the elimination of one-literal clauses rule (Rule I), applying the affirmative-negative rule (Rule II) and eliminate atomic formulas rule (Rule III). These rules are repeated until it is decided that the conjunction obtained initially was consistent or inconsistent. (Details on this algorithm can be found in [12] and [13].)

3.2 SAT Solvers

The example of DPLL mentioned in [12] and [13] generated more than 500 quantifier-free lines and was calculated to be valid in a few of minutes. Highly efficient SAT-solvers were introduced as a result of the excellent performance of the algorithm [4]

Satisfiability of a formula is the existence of a substitution which makes variables either true or false, so that the original formula with a substitution is a tautology [3]. This algorithm has been used to calculate validity of first-order logic formulas, whose implementation details are described in reference [14], however for formulas which are not logically valid the algorithm entered an infinite loop without giving a result.

The major modern DPLL-based SAT solvers do not use the original DPLL algorithm, but improvements have been introduced as described in [36] and called Abstract DPLL. The solvers used these days use the Abstract DPLL with Learning mechanism. The DPLL procedure is restarted when not enough progress is made, so that knowledge from the previous run is used in the subsequent run of the algorithm to search in a faster way [36].

Satisfiability modulo theories also expand boolean satisfiability by including equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers and other first-order theories. SMT-solvers have proven highly scalable, efficient and suitable for integrating theory reasoning [34].

3.2.1 Simplify Theorem Prover

Simplify was developed by Compaq as a prototype of a SMT theorem prover. It was constructed to accept any first-order logic formula together with quantifiers and obtain a result stating whether the formula was valid or invalid [14]. Invalid formulas generate a trace and error message for the user. The mechanism used by Simplify is to check for satisfiability of the negation of the formula it is validating [14].

M. Davis and H. Putnam [13] describe the satisfiability algorithm, which terminates and yields the validity of a given formula. For formulas which are not logically valid the algorithm enters an infinite loop without giving a result. Simplify uses this algorithm in order to calculate validity of the given first-order logic formula, the implementation details of which are described in reference [14].

Simplify also implements the Simplex algorithm originally introduced by George B. Dantzig [11] [1]. This tableaus technique improves performance over using the original satisfiability algorithm only. The description of the Simplex algorithm is also included in reference [14].

A well known approach for quantifier reasoning with ground decisions - E-matching algorithm - is also used which would later be implemented in other SMT-solvers [33].

An interesting ability of Simplify is detection and reporting of errors in the formulas. It maintains a conjunction of literals that characterizes the formula and prints the conjunction to the user as a counterexample. For example if we want to check validity of

$$x \geq 0 \Rightarrow x > 10$$

Simplify would report an error as the negation of the formula is not satisfiable and the counterexample presented would be as follows [14]:

$$x \geq 0 \wedge x \leq 10$$

In case of inability to calculate the value of a procedure, Simplify returns the formula with a procedure unsolved which indicates where the problem occurred.

Simplify uses a lisp-like syntax to enter formulas and terms. The boolean connectives include

```
AND OR NOT IMPLIES IFF EXPLIES
```

and relations contain

```
EQ NEQ < <= > >= DISTINCT
```

The operations on literals include only

```
+ - * select store
```

where select and store are operations on an array.
Simplify also has the ability to add new formulas and assumptions to be used in other proofs using the

```
BG_PUSH
```

command.

3.2.2 Z3 Theorem Prover

Satisfiability modulo theories (SMT) expand boolean satisfiability (SAT) by including equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers, and other first-order theories [34]. Satisfiability Modulo Theories solvers have proven highly scalable, efficient and suitable for integrating theory reasoning [33].

In September 2007, Microsoft Research introduced a new SMT-solver called Z3, targeted at solving problems in software verification and analysis [34]. This solver uses a well-known approach for incorporating quantifier reasoning with ground decision procedures originally used in the Simplify theorem prover [18] and supports three input formats - SMT-LIB, Simplify and DIMACS format. It can also be called using an ANSI C API, a .NET API, and an OCaml API.

The architecture of Z3 is an integration of a modern DPLL-based SAT solver, core theory solver which handles equalities and uninterpreted functions, satellite solvers and an E-matching abstract machine [34].

The approach is however improved in Z3 and the algorithms identify matches on E-graphs incrementally and efficiently. Experimental results reference [33] show substantial performance improvements over existing SMT solvers.

E-matching takes a set of ground equations E, a ground term t and a term p that contains variables as input. The output is the set of substitutions Θ, modulo E, over the variables in p, such that $E \models t \simeq \Theta(p)$. The two substitutions are called equivalent if the right side of the equation of each substitution is pairwise congruent modulo E [33].

Z3 introduces a notion of an index on E-graphs (E-matching code trees) which allow performing matching against several patterns simultaneously. These indexes when combined with the inverted path index can find patterns that may potentially match after modifications in the E-graph. The overhead of using the improved E-matching algorithm is searching and maintaining sets of patters for quantified formulas so that retrieval can be efficient [33].

Chapter 4

Loop Invariants

The basic technique to find the loop invariants was proposed by C.A.R. Hoare [24]. Finding them however is not an easy process, especially in large programs, and was prone to create problems. Extensive research has been conducted to find an automatic way of finding loop invariants.

In sixties a technique using calculus of relations was proposed for verification and analysis of computer programs [5], [6], [30]. The technique has never become widely accepted because of the huge amount of symbolic calculations that need to be performed even though it has many theoretical and methodological advantages (for example that it emphasized calculation and not proving).

It is easy to compute polynomial invariants of loops due to symbolic computation tools available now.

4.1 Program Verification by calculating relations

Carette, Janicki and Zhai [9] developed a program that reads in the text of a program and returns the symbolic calculation of the output. The article uses an example of program factorial that calculates $n!$. To validate the program there were three critical pieces of information that were required — the initial condition being the value of the variables before the loop executed, the recurrence statements inside the loop that are executed when the loop is progressing and loop termination which is the number of times recurrence statements are executed before the loop exits. Using the above information we can symbolically execute the program and validate it against the post conditions.

The results of this section were published as [25].

4.1.1 Verification of Factorial program

This section will explain more details of program verification by calculating relations on the factorial program and showing how it can be verified using Z3. The factorial program can be represented in Maple syntax as follows [9]:

```
factorial:=proc(n::posint)
local i, fac;
    i:=1;
    fac:=1;
    while i < n do
    begin
        i:=i+1;
        fac:=fac*i;
    end;
    return fac
end proc;
```

It is relatively easy to prove using calculus of relations and predicates that the above program computes $n!$. An example of such proof can be found, for example, in [6], [8], [9]. Most such proofs require some amount of human ingenuity, usually to "see" a pattern which, in most cases, is impossible to mechanize. The technique proposed in [8] and implemented in [46] translates the above program into the recurrence relations:

$$\begin{aligned} \mathrm{i}(0) &= 1 \\ \mathrm{fac}(0) &= 1 \\ \mathrm{i}(\mathrm{k}+1) &= \mathrm{i}(\mathrm{k})+1 \\ \mathrm{fac}(\mathrm{k}+1)&=\mathrm{fac}(\mathrm{k}) \cdot \mathrm{i}(\mathrm{k}+1) \end{aligned}$$

and then the recurrence relations are solved symbolically using powerful *Maple* tools. The answer produced is the text "n!".

Relations	Recurrence and Initial Condition
StateTransition(i,1)	Initial Condition: i(0)=1
StateTransition(fac,1)	Initial Condition: fac(0)=1
FixedPoint(i < n,	Loop Termination: $t = min\{k \geq 0 \mid i(k) \geq n\}$
StateTransition(i,i+1)	Recurrence: $i(k+1) = i(k) + 1$
StateTransition(fac,fac*i),)	Recurrence: $fac(k+1) = fac(k) * i(k+1)$
return fac	fac(t)

With Simplify and Z3 we proceed differently. We have to produce two predicate expressions (in the appropriate format) — one that represents the program, and another that represents the presumed outcome of the program; we then compare them. While the tool presented in [9], [46] was able to automatically translate Maple programs into appropriate recurrences, so far we do not have such possibilities implemented for Z3 or Simplify.

The math satements can be expressed in Simplify input format as assumptions about the program. For the above program, the BG_PUSH statements can be applied as follows to define both of the initial conditions:

```
(BG_PUSH (FORALL (n ) (EQ (fac 1 n) 1)))
```

```
(BG_PUSH (FORALL (n )(EQ(i 1 n) 1)))
```

and the body of the loop (where k is the number of iterations of the loop):

```
(BG_PUSH (FORALL (n k) (IMPLIES (AND (>= n (i k n)) (> k 1)) (EQ
(fac k n) (* (fac (- k 1) n ) (i k n))))))
```

```
(BG_PUSH (FORALL (n k) (IMPLIES (AND (>= n (i k n)) (> k 1)) (EQ (i
k n) (+ (i (- k 1) n) 1)))))
```

The program can be invoked by calling function *fac* and passing number n to cancluate $n!$.

```
(BG_PUSH (FORALL (n ) (IMPLIES (> n 0) (EQ (fact n) (fac n n)))))
```

In order to check the validity a special function had to be created called *factorial* that calculates $n!$ using the mathematical definition. In Simplify syntax the definition of factorial is as follows:

```
(BG_PUSH (EQ (factorial 1) 1))

(BG_PUSH (FORALL (n ) (IMPLIES (> n 1) (EQ (factorial n) (*
(factorial (- n 1)) n)))))
```

The validity is checked by comparing the value of $n!$ received by factorial (mathematical equation) and the program we are trying to verify by checking the equality of the two $n!$ obtained:

```
(EQ (factorial 5) (fact 5))
```

If Z3 returns Valid after entering the above, the program might be correct. Note that we can show that for instance *factorial 7 = fac 7*, *factorial 17 = fac 17*, but we cannot show that *factorial k = fac k* for a variable k. We believe that this is more than testing [35], because this equality means equivalence of appropriate predicate expressions generated during both calculations, however this is less than a formal proof.

4.2 Examples

Similar to the factorial example presented in previous section, other programs can be represented in Simplify format. To simplify the notation, Java will be used to write out algorithms of the programs analyzed.

4.2.1 Sum of Elements of the Array

Since Z3 does not have the required mathematical formulas, in order to validate the program we need to create the mathematical representation of the sum of elements in Simplify format. The correct sum of elements will be expressed as SumElm.

```
(BG_PUSH (FORALL (a k) (IFF (> k 1) (EQ (SumElm a k) (+ (select a k)
(SumElm a (- k 1)))))))

(BG_PUSH (FORALL (a k) (IFF (EQ k 1) (EQ (SumElm a k) (select a
k)))))
```

The program that we are trying to validate is as follows:

```
public int SumArray(int[] a)
{
    int s = 0;
    int k = 0;
    int n = a.length;
    while (k<n){
        s = s+a[i];
        k = k+1;
    }
    return s;
}
```

The statements in Simplify format that represent the program and are entered into Simplify are the following:

Initial conditions

```
(BG_PUSH (FORALL (a k) (IFF (EQ k 1) (EQ (SumArray a k) (select a
k)))))
```

```
(BG_PUSH (FORALL (a k) (IFF (EQ k 1) (EQ (i k) 1))))
```

Recurrence Equations

```
(BG_PUSH (FORALL (a k) (IFF (> k 1) (EQ (SumArray a k) (+ (SumArray
a (- k 1)) (select a (i k)))))))
```

```
(BG_PUSH (FORALL (k) (IFF (> k 1) (EQ (i k) (+ (i (- k 1)) 1)))))
```

Where k again starts at the loop termination value. Then we can test for any *array* a of example length 6 by entering the following equation statement:

```
(EQ (SumElm a 6) (SumArray a 6))
```

Z3 validates this program successfully. Note that in this way we have proved that the above program calculates the sum of *any* array of length 6. We can do it (at least in theory) for any concrete number, i.e. for any concrete length, but not for an arbitrary n.

4.2.2 Quick Sort

For sorting algorithms we need to validate the program by making sure the newly obtained array is actually sorted. The following will check the array and return 1 if it is sorted and 0 if it is not sorted.

```
(BG_PUSH (FORALL (a i) (IMPLIES (EQ i 1) (EQ (sorted a i) 0))))

(BG_PUSH (FORALL (a i) (IMPLIES (AND (> i 1) (>= (select a i)
(select a (- i 1)))) (EQ (sorted a i) (sorted a (-i 1)) ))))

(BG_PUSH (FORALL (a i) (IMPLIES (AND (> i 1) (< (select a i) (select
a (- i 1)))) (EQ (sorted a i) 1) )))
```

The Quick Sort algorithm is as follows:

```
public int[] quicksort (int[] a, int p, int r)
{
    if (p < r){
        int q = partition(a, p, r);
    }
    quicksort (a, p, q-1);
    quicksort (a, q+1, r);
    return a;
}

public int partition(int[] a, int p, int r)
{
    int x = a[r];
    int i = p - 1;
    for (int j=p; j<r; j++)
    {
        if (a[j] <= x)
        {
            i = i + 1;
            exchange(a[i], a[j]);
            exchange(a[i+1], a[r]);
```

```
        }
    }
    return (i + 1);
}
```

The Quick sort can be represented by the following statements in Simplify syntax:

```
(BG_PUSH (FORALL (a n p r) (IFF (< p r) (EQ (quickSort a n p
r) (quickSort (quickSort a n (+ (partition (changes a n p r) n p r)
1) r) n p (- (partition (changes a n p r) n p r) 1))))))

(BG_PUSH (FORALL (a n p r) (IFF (>= p r) (EQ (quickSort a n p r)
a))))

(BG_PUSH (FORALL (a n p r j) (EQ (changes a n p r) (LoopOne a n p r
(select a r) (- p 1) p))))

(BG_PUSH (FORALL (a n p r j) (EQ (partition a n p r) (Loop a n p r
(select a r) (- p 1) p))))

(BG_PUSH (FORALL (a n p r x i j) (IFF (AND (< j r) (<= (select a j)
x)) (EQ (Loop a n p r x i j) (Loop a n p r x (+ i 1) (+ j 1))))))

(BG_PUSH (FORALL (a n p r x i j) (IFF (>= j r) (EQ (Loop a n p r x i
j) (+ i 1)))))

(BG_PUSH (FORALL (a n p r x i j) (IFF (AND (< j r) (<= (select a j)
x)) (EQ (LoopOne a n p r x i j) (LoopOne a n p r x (+ i 1) (+ j
1))))))

(BG_PUSH (FORALL (a n p r x i j) (IFF (>= j r) (EQ (LoopOne a n p r
x i j) (exchange a n i j r)))))

(BG_PUSH (FORALL (a n i j r) (EQ (exchage a n i j r) (swap (swap a n
i j) n (+ i 1) r))))
```

```
(BG_PUSH (FORALL (a n i j) (EQ (swap a n i j) (store (store (store a
(select a j) (+ n 1)) (select a i) j) (select a (+ n 1)) j))))
```

To test the algorithm the following needs to be executed:

```
(BG_PUSH (FORALL (a) (EQ (Array a) (store (store (store (store
a 1 5) 2 2) 3 4) 4 6))))

(EQ (sorted (quickSort (Array a) 4 1 4) 4) 1)
```

Z3 validates this program successfully, however the solution is less general than the solution for Sum as it only validates a concrete array.

4.2.3 Merge Sort

Before the Merge sort example can be presented, it is important to note that Simplify syntax has no built-in division. The division of elements has to be explicitly defined in order to be able to use it.

```
(BG_PUSH (FORALL (a b i c) (IFF (> c a) (EQ (div a b i c) (- i
1)))))

(BG_PUSH (FORALL (a b i c) (IFF (<= c a) (EQ (div a b i c) (div a b
(+ i 1) (+ c b))))))

(BG_PUSH (FORALL (a b i c) (EQ (divide a b) (div a b 0 0))))
```

In Merge sort, the array is divided into subarrays containing parts of the original array. The function that splits the array needs to be defined.

```
(BG_PUSH (FORALL (b) (EQ (Array b) (store b 1 0))))

(BG_PUSH (FORALL (a m n k) (IMPLIES (> k (- n m)) (EQ (subArray a m
n k) (store (Array b) (select a k) 1)))))

(BG_PUSH (FORALL (a m n k) (IMPLIES (<= k (- n m)) (EQ (subArray a m
n k) (store (subArray a m n (+ k 1)) (select a (+ k m)) k)))))
```

The merge sort algorithm is as follows:

```
public int[] mergesort(int m)
{
    int[] left = new int[0];
    int[] right = new int[0];
    int middle;
    if (m.length <= 1){return m}
    else {
        middle = (m.length/2)
        for (int i=0; i<=middle; i++){
            left[left.length+1]=m[i];
        }
        for (int i=(middle+1); i<=m.length; i++){
            right[right.length+1]=m[i];
        }
        left = mergesort(left);
        right = mergesort(right);
        return = merge(left, right);
    }
}

public int[] merge (int[] left,int[] right)
{
    int[] result = new int[0];
    while (left.length > 0 && right.length > 0)
    {
        if (left[0] <= right[0]){
            append(left[0], result);
            left = rest(left);
        } else {
            append(right[0],result);
            right = rest(right);
        }
    if (left.length} > 0 {
        append(rest(left), result);
```

```
    }
    if (right.length > 0){
        append(rest(right),result);
    }
    return result;
}
```

The mergesort function can be defined in Simplify as follows:
The initial values are defined(`subArray` and `divide` are explicitly used):

```
(BG_PUSH (FORALL (a m n) (IFF (<= (- n m) 1) (EQ (mergesort a m n)
(subArray a m n 1)))))

(BG_PUSH (FORALL (a m n) (IFF (> (- n m) 1) (EQ (middle a m n)
(divide (- n m) 2)))))
```

The recurrences for Mergesort are:

```
(BG_PUSH (FORALL (result) (EQ (Array result) (store result 1 0))))

(BG_PUSH (FORALL (a m n) (IFF (> (- n m) 1) (EQ (mergesort a m n)
(merge (mergesort a m (middle a m n)) (- (middle a m n) m)
(mergesort a (+ 1 (middle a m n)) n) (- n (+ 1 (middle a m n))) 0
(Array result))))))
```

The second step that is used by Mergesort is merging the sorted pieces of the array into the main array which at the end will be sorted. It can be represented by the following predicate (in Simplify format):

```
(BG_PUSH (FORALL (a m k c) (EQ (appendfirst a m k c) (store c
(select a 1) k))))

(BG_PUSH (FORALL (a m b n k c) (IFF (AND (> m 0) (AND (> n 0) (<=
(select a 1) (select b 1)))) (EQ (merge a m b n k c) (merge (rest a
m (- m 1)) (- m 1) b n (+ k 1) (appendfirst a m k c))))))

(BG_PUSH (FORALL (a m b n k c) (IFF (AND (> m 0) (AND (> n 0) (>
```

```
(select a 1) (select b 1)))) (EQ (merge a m b n k c) (merge a m
(rest b n (- n 1)) (- n 1) (+ k 1) (appendfirst b n k c))))))

(BG_PUSH (FORALL (a m k c t) (IFF (> t 0) (EQ (appendrest a m k c t)
(store c (select (appendrest a m k c (+ t 1)) t) (+ k 1))))))

(BG_PUSH (FORALL (a m k c t) (IFF (EQ t m) (EQ (appendrest a m k c
t) (store c (select a t) (+ k 1))))))

(BG_PUSH (FORALL (a m b n k c) (IFF (AND (> m 0) (<= n 0)) (EQ
(merge a m b n k c) (appendrest a m k c 1)))))

(BG_PUSH (FORALL (a m b n k c) (IFF (AND (<= m 0) (> n 0)) (EQ
(merge a m b n k c) (appendrest b n k c 1)))))

(BG_PUSH (FORALL (a m k) (IFF (EQ k 1) (EQ (rest a m k) (store a
(select a 2) k)))))

(BG_PUSH (FORALL (a m k) (IFF (< k (- m 1)) (EQ (rest a m k) (rest
(store a (select a (+ k 1)) k) m (- k 1))))))

(BG_PUSH (FORALL (a m b n k c) (IFF (AND (<= n 0) (<= m 0)) (EQ
(merge a m b n k c) c))))
```

The test uses a defined Array a and checks if the array is sorted.

```
(BG_PUSH (FORALL (a) (EQ (Array a) (store (store (store (store a 1
5) 2 2) 3 4) 4 6))))

(EQ (sorted (mergesort (Array a) 1 4) 4) 1)
```

In this example Z3 is unable to validate the algorithm successfully and it does not provide a counterexample. The same definition of this sorting algorithm, when plugged into Simplify, results in a successful proof that marks the algorithm valid.

4.2.4 Bubble Sort

The Bubble Sort algorithm is as follows

```
public int[] bubblesort(int[] a)
{
    for (int i=1 ; i<= a.length; i++){
        for (int j=a.length; j>i; j--){
            if (a[j] < a[j-1]){
                swap(a[j], a[j-1]);
            }
        }
    }
    return a;
}
```

The initial conditions are as follows:

```
(BG_PUSH (FORALL (a n j i) (IFF (EQ j i) (EQ (bbsort a n i j)
(bbsort a n (- i 1) n)))))

(BG_PUSH (FORALL (a n j i) (IFF (EQ i 0) (EQ (bbsort a n i j) a))))
```

The recurrence equations are:

```
(BG_PUSH (FORALL (a n j i) (IFF (AND (>= i 1) (AND (>= j n) (AND
(NEQ i j) (< (select a j) (select a (-j 1)))))) (EQ (bbsort a n i j)
(opt3 (opt2 (opt1 (bbsort a n i (- j 1)) n i j) n i j) n i j)))))

(BG_PUSH (FORALL (a n j i) (EQ (opt1 a n i j) (store a (select a j)
(+ n 1)))))

(BG_PUSH (FORALL (a n j i) (EQ (opt2 a n i j) (store a (select a (-
j 1)) j))))

(BG_PUSH (FORALL (a n j i) (EQ (opt3 a n i j) (store a (select a (+
n 1)) (- j 1)))))
```

The calling function is bubblesort taking an array a of size n.

```
(BG_PUSH (FORALL (a n j i) (EQ (bubblesort a n) (bbsort a n n n))))
```

In order to test the code we need to test it on a given array Array a.

```
(BG_PUSH (FORALL (a) (EQ (Array a) (store (store (store (store a 1
5) 2 2) 3 4) 4 6))))
```

The array received from running the code should be sorted. The following command will check that it is indeed sorted:

```
(EQ (sorted (bubblesort (Array a) 4) 4) 1)
```

4.2.5 Insertion Sort

The sorting algorithm is as follows:

```
public int[] Insert(int[] a)
{
    a=sort(a);
    int i=a.length;
    while ((i>0)&&(a[i-1]>x){
        a[i]=a[i-1];
        i=i-1;
    }
    a[i]=x;
    return a;
}
```

The initial Condition is describing that if the value of i is 0 then the element should be inserted as a first element of the array.

```
(BG_PUSH (FORALL (a i x) (IFF (EQ i 0) (EQ (insert a i x) (store a 1
x)))))
```

The Recurrence equations are:

```
(BG_PUSH (FORALL (a i x) (IFF (AND (> i 1) (> (select a i) x)) (EQ
(insert a i x) (insert (store a (+ i 1) (select a i)) (-i 1) x)))))

(BG_PUSH (FORALL (a i x) (IFF (AND (> i 1) (<= (select a i) x)) (EQ
(insert a i x) (store a (+ i 1) x)))))
```

Again, a given array needs to be used. Here the array has 4 elements and we are trying to add the 5th element '3'.

```
(BG_PUSH (FORALL (a) (EQ (Array a) (store (store (store (store a 1
5) 2 2) 3 4) 4 6))))
```

Again, we check running the command to see if it is sorted.

```
(EQ (sorted (insert (bubblesort (Array a) 4) 4 3) 5) 1)
```

4.2.6 Selection Sort

The selection Sort algorithm is as follows:

```
public int[] selectionSort(int[] a)
{
    int min;
    for (int i=0; i<(size-1); i++)
    {
        min = i;
        for (int j=i+1; j<size; j++)
        {
            if (a[j] < a[min]){
                min = j;
            }
        }
        swap(a[i], a[min]);
    }
    return a;
}
```

The main call of the procedure is selectionSort:

```
(BG_PUSH (FORALL (a n i j min) (EQ (selectionSort a n) (selsort a n
(- n 2) (- n 1) (- n 1)))))
```

The initial conditions are as follows:

```
(BG_PUSH (FORALL (a n j i min) (IFF (EQ j i) (EQ (selsort a n i j
min) (swap a n i j min)))))

(BG_PUSH (FORALL (a n j i min) (IFF (EQ i 0) (EQ (selsort a n i j
min) a))))
```

The recurrence equations are:

```
(BG_PUSH (FORALL (a n j i min) (IMPLIES (AND (< (select a j) (select
a min)) (AND (> i 0) (> j i))) (EQ (selsort a n i j min) (selsort a
n i (- j 1) j)))))

(BG_PUSH (FORALL (a n j i min) (IMPLIES (AND (>= (select a j)
(select a min)) (AND (> i 0) (> j i))) (EQ (selsort a n i j min)
(selsort a n i (- j 1) min)))))
```

The swap algorithm can be presented as follows:

```
(BG_PUSH (FORALL (a n j i min) (EQ (swap a n i j min) (opt3 (opt2
(opt1 (selsort a n (- i 1) i min) n i min) n i min) n i min))))

(BG_PUSH (FORALL (a n i min) (EQ (opt1 a n i min) (store a (select a
min) (+ n 1)))))

(BG_PUSH (FORALL (a n i min) (EQ (opt2 a n i min) (store a (select a
(- min 1)) min)))) (BG_PUSH (FORALL (a n i min) (EQ (opt3 a n i min)
(store a (select a (+ n 1)) (- min 1)))))
```

Testing of the algorithm:

```
 (BG_PUSH (FORALL (a) (EQ (Array a) (store (store (store (store
a 1 5) 2 2) 3 4) 4 6))))

(EQ (sorted (selectionSort (Array a) 4) 4) 1)
```

4.2.7 Shell Sort

The Shell Sort Algorithm is:

```
public int[] shellSort(int A[], int size) {
    int i, j, increment, temp;
    increment = size/2;
    while (increment > 0)
    {
        for (i=increment; i < size; i++)
        {
            j = i;
            temp=A[i];
            while ((j >= increment) && (A[j-increment] > temp))
            {
                A[j] = A[j - increment];
                j = j - increment;
            }
            A[j] = temp;
        if (increment == 2){
            increment = 1;}
        else {
            increment = (int)(increment / 2.2);}
        }
    }
    return a;
}
```

Main function of shell Sort is:

```
(BG_PUSH (FORALL (a size) (EQ (shellSort a size) ( loopOne a size
(divide size 2)))))

(BG_PUSH (FORALL (size) (IFF (EQ (divide size 2) 2) (EQ (increment
size) 1))))
```

```
(BG_PUSH (FORALL (size) (IFF (NEQ (divide size 2) 2) (EQ (increment
size) (divide size 3)))))
```

The recurrence equations are:

```
(BG_PUSH (FORALL (a size inc temp i j) (IFF (> inc 0) (EQ
(loopOne a size inc) (loopOne (loopTwo a size inc inc) size
(increment size))))))

(BG_PUSH (FORALL (a size inc temp i j) (IFF (<= inc 0) (EQ (loopOne
a size inc) a))))

(BG_PUSH (FORALL (a size inc temp i j) (IFF (< i size) (EQ (loopTwo
a size inc i) (loopTwo (LoopThree a size inc (select a i) i i) size
inc (+ i 1))))))

(BG_PUSH (FORALL (a size inc temp i j) (IFF (>= i size) (EQ (LoopTwo
a size inc i) a))))

(BG_PUSH (FORALL (a size inc temp i j) (IFF (AND (>= j inc) (>
(select a (- j inc)) temp)) (EQ (LoopThree a size inc temp i j)
(assign a j (select (LoopThree a size inc temp i (- j inc)) (- j
inc)))))))

(BG_PUSH (FORALL (a size inc temp i j) (IFF (OR (< j inc) (<=
(select a (- j inc)) temp)) (EQ (LoopThree a size inc temp i j)
(assign a j temp)))))
```

Testing of the algorithm:

```
(BG_PUSH (FORALL (a) (EQ (Array a) (store (store (store (store
a 1 5) 2 2) 3 4) 4 6))))

(EQ (sorted (shellSort (Array a) 4) 4) 1)
```

Which returns valid as expected.

This chapter shows that Z3 has trouble with verifying program by calculating relations. It is unable to handle complicated calculations in the Simplify format and programs that do not use integers [15]. For example a construction of a proof of a program that creates Chebyshev polynomials is impossible in Z3.

Large values specified for input also limit Z3 significantly. For example, when calculating the sum of the array elements, the sum for array of length 20 returned valid in less than a second, however if the same command was run for 30 elements the SumArray ran for extended amount of time without being able to reach a conclusion. Similar results have been seen with other examples provided in this chapter.

The requirement of working on specific values of variables is also an important limitation that makes Z3 unfit as a prover of program correctness for arbitrary values of program variables. When working with the factorial example, the value of p had to be specified and when working with arrays, Z3 was unable to work with arbitrary array. In the instance of SumArray, Z3 attempted to validate the program for any array of length p, however results were inconclusive.

When working with sorting algorithms, Z3 had to work with an exact array, and did not even account for assumptions of the elements. For example when an assumption was created that element a is less than element b in the array, Z3 did not return an error, however it took a very long time to complete the validation, which makes the result inconclusive.

The lack of understanding of mathematical concepts such as factorial has been shown to be a limitation of Z3 as well, where a mathematical definition had to be created. The user-created definition of factorial has not been validated and thus cannot be assumed to be correct.

Chapter 5

Concurrency

While sequential programs can be annotated with assertions which reflect proof obligations, it is not clear how concurrent programs should be annotated and what proof obligations are required to validate them. This chapter will introduce the core of the Owicki/Gries Theory and show examples of validating such programs using Z3. There are many models of concurrency and Owicki/Gries method is one of the oldest and simplest which is a good starting point for proving concurrent programs with Z3. It can be noted that if Z3 is unable to successfully use this model for proving program correctness, it will probably not work with other models either.

5.1 Owicki/Gries Theory

Susan Speer Owicki and David Gries developed a set of rules that allow correct annotation of a multiprogram and define the proof obligations required. The rules have been defined as follows [19]:

Rule of Global Correctness

Assertion P in a component is globally correct whenever for each $\{Q\}S$ - i.e. for each atomic statement S with pre-assertion Q — taken from a different component,

$$\{P \wedge Q \wedge I\}S\{P \wedge I\}$$

is a correct Hoare-triple.

Rule of Local Correctness

For local correctness of an assertion P in a component, we distinguish two cases.

- If P is the (one and only) initial assertion of the component, it is locally correct whenever it is implied by the precondition of the multiprogram as a whole
- If P is textually preceded by $\{Q\}S$, i.e. by atomic statement S with pre-assertion Q, it is locally correct whenever {Q}S{P} is a correct Hoare-triple.

Rule of Postcondition

Postcondition R of a multiprogram is correct whenever

- all components are guaranteed to terminate
- R is implied by the conjunction of the post-assertions of the individual components.

Rule of Private Variables

For an assertion in a component that depends on private variables of that component only, it suffices to prove local correctness, because its global correctness is guaranteed.

Rule of Orthogonality

An assertion is maintained by all assignments to variables not occurring in it

Rule of Disjointness

Assertion P is (globally) correct under $\{Q\}S$ if $[P \wedge Q \Rightarrow false]$.

Rule of Progress

Statement $\{Q\}$ if $B \rightarrow S$ fi is a component guaranteed to terminate if and only if the rest of the system, when constrained to Q, will, in a finite number of steps, converge to a state in which B is true.

Rule of Absence of total deadlock

A configuration of guarded statements containing one such statement per component, is deadlock free whenever it is possible to supply each guarded statement in the configuration with a correct pre-assertion in such a way that the conjunction of the pre-assertions implies the disjunction of the guards

The multiprogram as a whole is free of total deadlock whenever all such configurations are deadlock free.

An example of investigate the disjunction of the guards can be as follows:

While a first program component of the multiprogram has the following guard:

$$if\ B1 \rightarrow skip\ fi$$

The second has the following:

$$if\ B2 \rightarrow skip\ fi$$

The following needs to be true:

$$B1 \vee B2$$

The following examples will show how concurrent programs can be proved using the Owicki/Gries method and Z3.

5.2 Single statement concurrency

5.2.1 Example 1

The following simple program only checks for local and global correctness for programs A and B which contain a single assignment statement. Starting with $x = 0$, the program A adds 1 to x and the program B adds 2. The annotated program as per Owicki/Gries can be defined as follows:

```
   Pre: x=0
A:
   {x=0 ∨ x=2}
      x:=x+1;
   {x=1 ∨ x=3}
B:
```

{x=0 ∨ x=1}
x:=x+2;
{x=2 ∨ x=3}

The proof obligations are defined as follows:

Local Correctness

For program *A*:
x=0 ⇒ (x=0 ∨ x=2)
(x=0 ∨ x=2) ⇒ (1=x+1 ∨ 3=x+1)

For program *B*:
x=0 ⇒ (x=0 ∨ x=1) (x=0 ∨ x=1) ⇒ (2=x+2 ∨ 3=x+2)

Global Correctness

For program *A*:
(x=0 ∨ x=2) ∧ (x=0 ∨ x=1) ⇒ 0=x+2 ∨ 2=x+2

For program *B*:
(x=0 ∨ x=1) ∧ (x=0 ∨ x=2) ⇒ 0=x+1 ∨ 1=x+1

In simplify format

```
(IMPLIES (EQ x 0) (OR (EQ x 0) (EQ x 2)))

(IMPLIES (OR (EQ x 0) (EQ x 2)) (OR (EQ  1 (+ x 1)) (EQ 3 (+ x 1))))

(IMPLIES (EQ x 0) (OR (EQ x 0) (EQ x 1)))

(IMPLIES (OR (EQ x 0) (EQ x 1)) (OR (EQ 2 (+ x 2)) (EQ 3 (+ x 2))))

(IMPLIES (AND (OR (EQ x 0) (EQ x 2)) (OR (EQ x 0) (EQ x 1))) (OR (EQ 0 (+ x 2)
(EQ 2 (+ x 2))))

(IMPLIES (AND (OR (EQ x 0) (EQ x 1)) (OR (EQ x 0) (EQ x 2))) (OR (EQ 0 (+ x 1)
```

```
(EQ 1 (+ x 1))))
```

When the proof obligations are entered into Z3, the following is returned:
1: Valid.
2: Valid.
3: Valid.
4: Valid.
5: Valid.
6: Valid.

5.2.2 Example 2

The second example add additional complexity of possible deadlock. The multiprogram is as follows and represent a well known consumer and supplier algorithm:

Pre: in=0 $\wedge$ out=0
Prod:
*[if in $<$ C+out $\rightarrow$ skip fi
{in $<$ C+out}
in:=in+1;
]
Cons:
*[if out $<$ in $\rightarrow$ skip fi
{out $<$ in}
out := out+1;
]
Inv: out $\leq$ in $\wedge$ in $\leq$ C + out

The proof obligations are defined as follows:

local Correctness

((in=0 $\wedge$ out=0) $\Rightarrow$ in $<$ C+out) $\vee$ ((in=0 $\wedge$ out=0) $\Rightarrow$ out $<$ in)

Global correctness

in < C+out ∧ out < in ⇒ in < C+out+1
out < in ∧ in < C+out ⇒ out < in+1

Deadlock

in < C+out ∨ out < in

In Simplify Syntax:

```
(BG_PUSH (AND (<= out in) (<= in (+ C out))))

(BG_PUSH (<= 1 C))

(OR (IMPLIES (AND (EQ in 0) (EQ out 0)) (< in (+ C out))) (IMPLIES (AND (EQ in
(EQ out 0)) (< out in)))

(IMPLIES (AND (< in (+ C out)) (< out in)) (< in (+ C (+ out 1))))

(IMPLIES (AND (< in (+ C out)) (< out in)) (< out (+ in 1))) (OR (< in (+ C ou
(< out in))
```

When the proof obligations are entered as Z3 input, the following is returned:

1: Valid.
2: Valid.
3: Valid.
4: Valid.

5.2.3 Example 3

Use of private variables

Pre: x=0 ∧ a=0 ∧ b=0
A:

{a=0}
x,a := x+1, a+1;
{a=1}
B:
{b=0}
x,b := x+1, b+1;
{b=1}
Post: a+b=2

Proof obligations are defined as follows:

Precondition obligation

$x=0 \wedge a=0 \wedge b=0 \Rightarrow a=0$
$x=0 \wedge a=0 \wedge b=0 \Rightarrow b=0$

local correctness

$a=0 \Rightarrow a+1=1$
$b=0 \Rightarrow b+1=1$

global correctness

$a=0 \wedge b=0 \Rightarrow a=0$
$b=0 \wedge a=0 \Rightarrow b=0$

postcondition

$a=1 \wedge b=1 \Rightarrow a+b=2$

In Simplify format, this can be written as follows:

```
(IMPLIES (AND (EQ x 0) (AND (EQ a 0) (EQ b 0))) (EQ a 0))
(IMPLIES (AND (EQ x 0) (AND (EQ a 0) (EQ b 0))) (EQ b 0))
(IMPLIES (EQ a 0) (EQ (+ a 1) 1))
(IMPLIES (EQ b 0) (EQ (+ b 1) 1))
(IMPLIES (AND (EQ a 0) (EQ b 0)) (EQ a 0))
(IMPLIES (AND (EQ b 0) (EQ a 0)) (EQ b 0))
(IMPLIES (AND (EQ a 1) (EQ b 1)) (EQ (+ a b) 2))
```

After executing these statements under Z3, we get the following indication that our program is correct:

1: Valid.
2: Valid.
3: Valid.
4: Valid.
5: Valid.
6: Valid.
7: Valid.

5.2.4 Example 4

Private variables

Pre: x=0 $\wedge$ ($\forall$ j : dj = 0)
Comp.i *[{di = 0}
x,di := x+1, 1+di;
{di = 1}
x,di := x-1, -1 + di;
{di = 0}
]

Proof obligations are as follows:

Precondition obligation

$\forall$ i :: x=0 $\wedge$ ($\forall$ j :: dj = 0) $\Rightarrow$ di=0

local correctness

$\forall$ i :: di=0 $\Rightarrow$ di+1 = 1
$\forall$ i :: di=1 $\Rightarrow$ di-1 = 0

global correctness

di=0 $\wedge$ dj=0 $\Rightarrow$ di=0
di=0 $\wedge$ dj=1 $\Rightarrow$ di=0
di=1 $\wedge$ dj=0 $\Rightarrow$ di=1
di=1 $\wedge$ dj=1 $\Rightarrow$ di=1

In Simplify format this can be represented as follows:

```
(FORALL (i) (IMPLIES (AND (EQ x 0) (FORALL (j) (EQ (d j) 0))) (EQ (d i) 0)))
(FORALL (i) (IMPLIES (EQ (d i) 0) (EQ (+ (d i) 1) 1)))
(FORALL (i) (IMPLIES (EQ (d i) 1) (EQ (+ (- 0 1) (d i)) 0)))
(IMPLIES (AND (EQ (d i) 0) (EQ (d j) 0)) (EQ (d i) 0))
(IMPLIES (AND (EQ (d i) 0) (EQ (d j) 1)) (EQ (d i) 0))
(IMPLIES (AND (EQ (d i) 1) (EQ (d j) 0)) (EQ (d i) 1))
(IMPLIES (AND (EQ (d i) 1) (EQ (d j) 1)) (EQ (d i) 1))
```

Comments:
Z3 does not understand negative numbers such as (- 1) and complains about wrong arguments when Simplify is able to validate such formulas. Changing the definition of -1 to (- 0 1) is able to validate via Z3.

After executing the formulas we get the following:

1: Valid.
2: Valid.
3: Valid.
4: Valid.
5: Valid.
6: Valid.
7: Valid.

5.3 Relation of a Single Statement

As shown in Example 4 from the previous section, the Owicki/Gries method works well with Z3 and allows validation of concurrent programs, however in order to create

a fully automated program prover, it would be required to minimize the number of annotations specified and remove the invariants used in Owicki/Gries method. In order to accomplish this, proving by calculating relations can be used.

To start, a fairly trivial example of a program that is concurrent to itself can be designed. A program that splits into multiple threads where each thread runs separately would be an example of a program concurrent to itself. In the following example, the program will split n times, where n is a finite number. The program itself will look as follows:

Pre: {x=0}
Fork.k *[x:=x+1]
Inv: {$\forall$ k : x = k)}
Post: {$\forall$ n : x = n)}

The value of k represents the number of concurrent programs that are currently finished, where $n - k$ is the number of programs that are still to run.

In order to prove the program using relations, the program needs to be rewritten, and it looks as follows:

Pre: {x(0)=0}
Fork.k x(k)=x(k-1)+1;
Post: {$\forall$ n : x(n)=n}

Local correctness

This can be proven by using the relational method from Chapter 4, so that the proof obligation statements are as follows:

Pre: x(0)=0
statement: ($\forall$ k>0 : x(k)=x(k-1)+1)
To prove: x(1)=1

Please note that in the 'To prove' statement, k is one as each program runs only once.

The Simplify format proof would be as follows:

```
(BG_PUSH (FORALL (k) (IMPLIES (EQ k 0) (EQ (x k) 0))))

(BG_PUSH (FORALL (k) (IMPLIES (> k 0) (EQ (x k) (+ (x (- k 1)) 1)))))
(EQ (x 1) 1)
```

Which returns valid in Z3.

Global correctness

We can notice that our program has only one statement, and it either finishes or not. Global correctness is assured by reaching the correct postcondition.

The following are statements required for the proof obligation:

Initial confition:
($\forall$ k=0, n>0 : x(k)=0)

Recurance:
($\forall$ k>0, n>0 : x(k)=x(k-1)+1)

To prove:
($\forall$ n>0 : x(n) = n)

In Simplify format it would be as follows:

```
(BG_PUSH (FORALL (k) (IMPLIES (EQ k 0) (EQ (x 0) 0))))

(BG_PUSH (FORALL (k) (IMPLIES (> k 0) (EQ (x k) (+ (x (- k 1)) 1)))))
```

What we want to prove is as follows:

```
(FORALL (n) (IMPLIES (> n 0) (EQ (x n) n)))
```

However Z3 is unable to calculate it and marks it as invalid. After providing the value of n, the program is validated successfully:

```
(EQ (x 5) 5)
```

In the above example, we could specifically say that k is the number of programs that finished executing the single statement, then terminate the program when the value of $k = n$, thus all programs finished.

5.4 Relation of Multiple Disjoint Statements

The validation of a multiprogram with a single statement with calculating relations is not complicated as has been shown in the previous example. The complexity of the proof increases with the number of statements within the program. To illustrate this, let us look at the following example:

Pre: x=0 ∧ y=0
Comp.i *[x:=x+1;
{y<x}
y:=y+1;
Inv: y≤x

In relational format, the program will look as follows:

Pre: x(0)=y(0)=0
Comp.i x(i) = x(i-1)+1;
y(i) = x(i)+1;
Termination: (∀ k, x(k)=y(k)=k)

This example is very similar to the one before, however we now have two statements that we will be executing.

Local Correctness

Once again the local correctness is an execution of the program when $k = 1$. The following is required to be true after executing the statements once:

(∀ k=1 : y(1)=1 ∧ x(1)=1)

This can be entered in the simplify format as follows:

```
(BG_PUSH (FORALL (k) (IMPLIES (EQ k 0) (AND (EQ (x 0) 0) (EQ (y 0) 0)))))

(BG_PUSH (FORALL (k) (IMPLIES (> k 0) (EQ (x k) (+ (x (- k 1)) 1)))))

(BG_PUSH (FORALL (k) (IMPLIES (> k 0) (EQ (y k) (+ (y (- k 1)) 1)))))

(AND (EQ (y 1) 1) (EQ (x 1) 1))
```

This returns valid.

Global Correctness

One of the rules of global correctness is that when executing a statement from one program, it does collide with a statement from another program. Since statements are disjoint and just adding 1, it is enough to make sure that termination is reached.

In order to show the program terminates and provides the correct values, the following needs to be proved:

$(\forall\ n{>}0 : x(n) = y(n) = n)$

In Simplify format it would be as follows:

```
(BG_PUSH (FORALL (n k) (IMPLIES (EQ k 0) (AND (EQ (x n k) 0) (EQ (y n k) 0))))

(BG_PUSH (FORALL (n k) (IMPLIES (> k 0) (EQ (x n k) (+ 1 (x n (- k 1)))))))

(BG_PUSH (FORALL (n k) (IMPLIES (> k 0) (EQ (y n k) (+ 1 (y n (- k 1)))))))

(FORALL (n) (AND (EQ (x n n) n)  (EQ (y n n) n)))

(AND (EQ (x 5 5) 5) (EQ (y 5 5) 5))

(FORALL (n) (AND (EQ (x n n) n)  (EQ (y n n) n)))
```

This also returns valid.

5.4.1 Relation of Dependant Statements

For global correctness we need to take into account that at any given time during execution, there are some programs that have not started yet, some programs that have executed only the first statement, and some programs that have executed both statements and thus exited.

Pre: x=0 ∧ y=0
*[x:=x+1;
  y:=x]
Post: x=i ∧ y=x

Local Correctness

The local correctness is straightforward and it can be represented in Z3 format as follows

```
(BG_PUSH (FORALL (k) (EQ (x 0) 0)))

(BG_PUSH (FORALL (k) (EQ (y 0) 0)))

(BG_PUSH (FORALL (k) (IMPLIES (> k 0) (EQ (x k) (+ (x (- k 1)) 1)))))

(BG_PUSH (FORALL (k) (IMPLIES (> k 0) (EQ (y k) (x k)))))

(AND (EQ (x 1) 1) (EQ (y 1) 1))
```

which returns valid.

Global Correctness

```
(BG_PUSH (FORALL (k) (EQ (x 0) 0)))

(BG_PUSH (FORALL (k) (EQ (y 0) 0)))

(BG_PUSH (FORALL (k) (IMPLIES (> k 0) (EQ (x k) (+ (x (- k 1) 1))))))
```

```
(BG_PUSH (FORALL (k) (IMPLIES (> k 0) (EQ (y k) (x k)))))

(AND (EQ (x 5) 5) (EQ (y 5) 5))
```

which is also valid

In the example the dependency was only with one statement, such that x was independent of y, but y was dependent of x. The situation changes when both statements are dependent of each other.

5.4.2 Second example

A program that looks as follows can be considered:

Pre: x=0 $\wedge$ y=0
*[x:=y+1;
  y=x]
Post: y=x $\wedge$ x $\leq$ y+1;

As can be seen, both x and y appear in both of the statements.

Local Correctness

With local correctness the proof is simple, as each of the statements are executed only once, so that the proof in Simplify syntax looks as follows:

```
(BG_PUSH (FORALL (k) (EQ (x 0) 0)))

(BG_PUSH (FORALL (k) (EQ (y 0) 0)))

(BG_PUSH (FORALL (k) (IMPLIES (> k 0) (EQ (x k) (+ (y (- k 1)) 1)))))

(BG_PUSH (FORALL (k) (IMPLIES (> k 0) (EQ (y k) (x k)))))

(AND (EQ (y 1) (x 1)) (<= (x 1) (+ (y 0) 1)))
```

returns Valid

Global Correctness

Similarly we can define the global correctness by providing a value of k larger than 1.

```
(BG_PUSH (FORALL (k) (EQ (x 0) 0)))

(BG_PUSH (FORALL (k) (EQ (y 0) 0)))

(BG_PUSH (FORALL (k) (IMPLIES (> k 0) (EQ (x k) (+ (y (- k 1)) 1)))))

(BG_PUSH (FORALL (k) (IMPLIES (> k 0) (EQ (y k) (x k)))))

(AND (EQ (y 5) (x 5)) (<= (x 5) (+ (y 4) 1))
```

which returns valid.

5.4.3 Become more complicated

The example above assumes that both of the statements are executed at once, thus each program starts and finishes before another program starts. In the proof below this assumption is dropped. The execution of the program can be interrupted after the first statement, which significantly changes the values of x and y in postcondition.

For example if we split the program into five executions, and the first statement is executed five times first, then the value of $x(5)$ would be $y(0) + 1$ and the value of y at the end of the execution would be $x(5)$, thus still making the postcondition true as the value of $y(k) = x(k)$ and the value of $x(k) = y(0) + 1$.

The following returns invalid, even though appears to be correct for the scenario described above.

```
(AND (EQ (y 5) (x 5)) (<= (x 5) (+ (y 0) 1)))
```

With this chapter we can conclude that Z3 has a lot of limitations when working with concurrent programs.

It appears to be working well with the Owicki/Gries method to prove the concurrent programs, but it requires a lot of manual input that cannot be easily automated. Attempts to prove using relations was unsuccessful.

There were two significant limitations discovered in Z3 with the relational method. The first showed was with proving multiple programs run in parallel. Z3 required a

specific number of programs to be run in parallel and was unable to calculate for an arbitrary number of programs.

The second limitation was with being able to model a truly concurrent system, when execution of any program can be interrupted between statements by another program running concurrently. Z3 was able to prove only programs that started and finished uninterrupted.

Chapter 6

Recursive Programs

There aren't many methods that allow proving of recursive programs [40]. The most widely known approaches have been described by Zohar Manna [28], M. Foley and C. A. Hoare [21]. These methods verify the correctness of recursive programs by computational and structural induction methods (which prove the properties of a fixpoint [28] [29]), or by using Hoare Logic [21] [37] [40].

This chapter will provide examples of recursive programs proven with Z3 and illustrate the method used, which has been described by Nikolaj Popov and Tudor Jebelean [40], [26] and uses a Theorema system of methodology to derive verification conditions, allows these conditions to be generated automatically and is easy to follow.

6.1 A Practical Approach

The method described by Nikolaj Popov and Tudor Jebelean [40], [26] is based on Hoare logic such that given a program and its specifications, it generates proof obligations that are minimal to prove that the program satisfies the specifications. The method proves partial correctness and termination of a simple recursive program defined as follows in a domain $\mathbb{D}$

$$F[x] = \text{If } Q[x] \text{ then } S[x] \text{ else } C[x,F[R[x]]]$$

where Q is a predicate on $\mathbb{D}$ and S,C,R are functions that have been proved correct. The precondition of the program is I[x] and the postcondition is O[x,y].

Coherence

A program is coherent when all the calls made to its auxiliary programs do not violate their preconditions. The following proof obligations must be true to assure coherence of the program F[x].

$(\forall x : I_F[x])(Q[x] \Rightarrow I_S[x])$
$(\forall x : I_F[x])(\neg Q[x] \Rightarrow I_R[x])$

Partial Correctness

Partial correctness can be expressed as the following proof obligations [26]:

$(\forall x : I_F[x])(Q[x] \Rightarrow O_F[x, S[x]])$
$(\forall x : I_F[x])(\neg Q[x] \Rightarrow I_F[R[x]])$
$(\forall x : I_F[x])(\neg Q[x] \wedge O_F[R[x], F[R[x]]] \Rightarrow O_F[x, C[x, F[R[x]]]])$
$(\forall x : I_F[x])(\neg Q[x] \wedge O_F[R[x], F[R[x]]] \Rightarrow I_C[x, F[R[x]]])$

Termination

If I_S, I_C and I_R are preconditions of S,C and R, and the partial correctness is proven correct, then the termination of F can be expressed with the following proof obligation:

$(\forall x : I_F[x])(F'[x] = 0)$

where

F'[x] = If Q[x] then 0 else F[R[x]]

In other words the Q[x] will eventually be reached, the recursive function will no longer be called and 0 will be returned.

6.1.1 Total Correctness

Combining the Partial Correctness, Termination and Coherence proofs allows us to construct verification conditions:

$(\forall x : I_F[x])(Q[x] \Rightarrow I_S[x])$
$(\forall x : I_F[x])(\neg Q[x] \Rightarrow I_R[x])$
$(\forall x : I_F[x])(Q[x] \Rightarrow O_F[x, S[x]])$
$(\forall x : I_F[x])(\neg Q[x] \Rightarrow I_F[R[x]])$
$(\forall x : I_F[x])(\neg Q[x] \wedge O_F[R[x], F[R[x]]] \Rightarrow O_F[x, C[x, F[R[x]]]])$
$(\forall x : I_F[x])(\neg Q[x] \wedge O_F[R[x], F[R[x]]] \Rightarrow I_C[x, F[R[x]]])$

and a proof of termination of:

F'[x] = If Q[x] then 0 else F'[R[x]]

which is: $(\forall x : I_F[x])(F'[x] = 0)$

6.1.2 Factorial and recursion

In this section, the example of factorial will be revisited as an example of a simple recursive program that, in recursion notation, is as follows:

F[n] = If n=0 then 1 else n*F[n-1]

Where:

Q[n] $\Leftrightarrow$ n=0
S[n] $\Leftrightarrow$ 1
C[n] $\Leftrightarrow$ n*F[n-1]
R[n] $\Leftrightarrow$ n-1

and the domain of n is $\mathbb{N}$.

The following are the proof obligations for the Factorial program:

$(\forall n \in \mathbb{N})(n = 0 \Rightarrow n! = 1)$
$(\forall n \in \mathbb{N})(n \neq 0 \Rightarrow n - 1 \in \mathbb{N})$
$(\forall n \in \mathbb{N})(n \neq 0 \wedge (n-1)! = F(n-1) \Rightarrow n! = n * F(n-1))$

$(\forall n \in \mathbb{N})(n = 0 \Rightarrow true)$
$(\forall n \in \mathbb{N})(n = 0 \Rightarrow n \geq 0)$
$(\forall n \in \mathbb{N})(n \neq 0 \wedge (n-1)! = F(n-1) \Rightarrow n > 0)$

and the termination of:

$$F'[n] = \text{If } n=0 \text{ then } 0 \text{ else } F'[n-1]$$

which is: $(\forall n \in \mathbb{N})(F'[n] = 0)$

These proofs can be easily worked out by hand and Z3 should be able to validate them successfully. First the definition of factorial as $n!$ needs to be recalled.

```
(BG_PUSH (EQ (factorial 1) 1))

(BG_PUSH (FORALL (n ) (IMPLIES (> n 1) (EQ (factorial n) (*
(factorial (- n 1)) n)))))
```

for the purpose of this example the following definition statement also needs to be included:

```
(BG_PUSH (EQ (factorial 0) 1))
```

Also the definition of division must be included as the Simplify format does not include division.

```
(BG_PUSH (FORALL (a b i c) (IFF (> c a) (EQ (div a b i c) (- i
1)))))

(BG_PUSH (FORALL (a b i c) (IFF (<= c a) (EQ (div a b i c) (div a b
(+ i 1) (+ c b))))))

(BG_PUSH (FORALL (a b i c) (EQ (divide a b) (div a b 0 0))))
```

The last item is the definition of the function of F which can be simply defined as:

```
(BG_PUSH (FORALL (n) (EQ (F n) (factorial n))))
```

This brings to the proof obligations defined in this chapter, that in Simplify format are written as follows:

```
(FORALL (n) (IMPLIES (EQ n 0) (EQ (factorial n) 1)))

(FORALL (n) (IMPLIES (> n 0) (>= (- n 1) 0)))

(FORALL (n) (IMPLIES (AND (> n 0) (EQ (factorial (- n 1)) (F (- n 1))))
(EQ (factorial n) (* (F (- n 1)) n))))

(FORALL (n) (IMPLIES (>= n 0) (>=n 0)))

(FORALL (n) (IMPLIES (>= n 0) (>= n 0)))

(FORALL (n) (IMPLIES (AND (> n 0) (EQ (factorial (- n 1)) (F (- n 1))))
(> n 0)))
```

which returns valid for all the proof obligations.

The last proof obligation to verify is termination which is often the hardest to prove [39]. In the example of factorial, the following needs to be proven:

$(\forall n \in \mathbb{N})(F'[n] = 0)$

where

$F'[n] = If\ n = 0\ then\ 0\ else\ F'[n-1]$

The value of n is decremented by 1 on each iteration, meaning n will eventually reach 0 regardless of its initial value of $n \in \mathbb{N}$. This can be expressed in Simplify format as follows where Fp denotes F':

```
(BG_PUSH (FORALL (n) (IMPLIES (EQ n 0) (EQ (Fp n) 0))))

(BG_PUSH (FORALL (n) (IMPLIES (> n 0) (EQ (Fp n) (Fp (- n 1))))))
```

```
(FORALL (n) (IMPLIES (>= n 0) (EQ (Fp n) 0)))
```

Z3 returns invalid as it cannot evaluate (Fpn) for all possible n greater or equal to 0. This can again be omitted by strengthening the precondition of F. For example if we restrict the n to be between 0 and 100, the following new precondition is obtained:

$$I_{F-new[n]} \Leftrightarrow (n \in \mathbb{N} \wedge n \geq 100)$$

which changes the proof obligation to be:

$$(\forall n \in \mathbb{N} \wedge n \geq 100)(F'[n] = 0)$$

and again can be represented in Z3 as:

```
(FORALL (n) (EQ (Fp 100) 0)))
```

This however appears to have too many iterations for Z3 to handle and returns an error. Lowering the value to 50 returns valid.

6.2 Z3 and recursive programs

The example of factorial shows that the value of n needs to be specified so that the termination of the program at the value of n can be calculated. Knowing this Z3 limitation and the fact that Z3 can present recursive programs in their basic form, the following verification can be implemented.

Every recursive program can be represented as the following [28] [39] [26] [40] [37] [29].

$$\text{F[x]} = \text{If Q[x] then S[x] else C[x,F[R[x]]]}$$

In Z3 format it can be defined as a following background push statements for x $\in$ $\mathbb{N}$:

```
(BG_PUSH (IMPLIES (Q[x]) (S[x])))
```

```
(BG_PUSH (FORALL (x) (IMPLIES (not Q[x]) (C[x,F[R[x]]]))))
```

Once the recursive function is defined by the above statement, the result obtained for a given value of x can be compared with the expected result as defined by the postcondition.

6.2.1 Factorial

Let's revisit the factorial example again which is defined recursively as:

F[n] = If n=0 then 1 else n*F[n-1]

The factorial example, can be defined under Z3 with the following statement:

```
(BG_PUSH (EQ (F 1) 1))

(BG_PUSH (FORALL (n) (IMPLIES (> n 0) (EQ (F n) (* n (F (- n 1)))))))
```

This can be again proved with the following statement with an arbitrary value of n and with using the definition of factorial from the previous chapter.

```
(EQ (F 5) (factorial 5))
```

Which returns valid.

6.2.2 Other Examples

Fibonacci

The Fibonacci sequence can be defined as follows:

F[n] = If n<2 then 1 else F[n-1]*F[n-2]

Which can then be defined in Z3 with the following background push statements:

```
(BG_PUSH (FORALL (n ) (EQ (F 1) 1)))

(BG_PUSH (FORALL (n ) (EQ (F 0) 1)))

(BG_PUSH (IMPLIES (> n 1) (EQ (F n) (+ (F (- n 2)) (F (- n 1))))))
```

And can be proved by providing a value of n. For example, for $n = 1$, the formula is as follows:

```
(EQ (F 1) 1)
```

and it returns valid.

However for n greater than 1, Z3 is unable to calculate the sequence as it has trouble calculating the combination of recursive calls. For example for $n = 2$

```
(EQ (F 2) 1)
```

returns invalid.

Nested Recursion

A more complicated example is one that consists of two independent functions where one of the functions has nested recursion. For this we can take the example from Zohar Manna [28], which is as follows:

$$F1[x] \Leftarrow \text{if } x>10 \text{ then } x\text{-}10 \text{ else } F1[F1[x+13]]$$
$$F2[x] \Leftarrow \text{if } x>10 \text{ then } x\text{-}10 \text{ else } F2[x+3]$$

In Z3 format this will look as follows:

```
(BG_PUSH (FORALL (x) (IMPLIES (> x 10) (EQ (F1 x) (- x 10)))))

(BG_PUSH (FORALL (x) (IMPLIES (> x 10) (EQ (F2 x) (- x 10)))))

(BG_PUSH (FORALL (x) (IMPLIES (<= x 10) (EQ (F1 x) (F1 (F1 (+ x 13)))))))

(BG_PUSH (FORALL (x) (IMPLIES (<= x 10) (EQ (F2 x) (F2 (+ x 3))))))
```

This evaluates to valid for $x > 10$ as per base case:

```
(EQ (F1 11) (F2 11))
```

however when x is less than 0, the nested recursion is not being evaluated as per the counter example.

```
Counterexample:
  context:
    (AND
      (NEQ (F1 1) 3)
    )

1: Invalid.
```

6.2.3 Induction proofs and recursion

The best way of verifying the correctness of recursive programs is to use induction proofs which allow verification of the program for any value of the variable.

For example the factorial program defined as;

$$F[n] = \text{If } n=0 \text{ then } 1 \text{ else } n*F[n-1]$$

Has a base case that:

$$F[0] = 1$$

Then we can assume the following

$$\forall\ n>0\ (F[n-1] = \text{factorial of } n-1)$$

so that the function to prove is as follows:

$$\forall\ n>0\ (F[n] = n*F[n-1])$$

In Z3 this can be defined as follows

The base case

```
(BG_PUSH (FORALL (n) (IMPLIES (EQ n 0) (EQ (F n) 1))))
```

The assumption

```
(BG_PUSH (FORALL (n) (IMPLIES (> n 0) (EQ (F (- n 1)) (factorial (- n 1))))))
```

The inductive case

```
(FORALL (n) (IMPLIES (> n 0) (EQ (F n) (* n (F (- n 1))))))
```

In order to validate the program above, the definition of factorial has to be defined as shown in the previous chapters.

```
(BG_PUSH (EQ (factorial 1) 1))
```

```
(BG_PUSH (FORALL (n ) (IMPLIES (> n 1) (EQ (factorial n) (* (factorial (- n 1)
```

The definition of Fn can be defined as follows:

```
(BG_PUSH (EQ (F 1) 1))

(BG_PUSH (FORALL (n ) (IMPLIES (> n 1) (EQ (F n) (* (F (- n 1)) n)))))
```

Z3 is unable to validate the program and the following counterexample is returned:

```
(AND
  (NEQ (* n 1) n)
  (EQ (* 1 n) 2)
  (EQ (F n) 2)
)
```

The counterexample does not appear to be showing any issue with the program representation. The counterexample does not show that the inductive assumption statement has been used, and use of the assumption statement would be required to complete the proof. It is unclear why Z3 is unable to successfully use the assumption statement.

This chapter shows that Z3 is not able to successfully validate recursive programs. When using the method described by Nikolaj Popov and Tudor Jebelean [40], [26], the termination cannot be calculated successfully for any number of recursive calls. It also has a problem with calculating termination for large number of recursive calls and prove had to settle for only 50 to be able to receive valid from Z3.

Even though the definition of a recursive program can be easily expressed with Z3 syntax, it does not help with the proofs. It again proves correct only for specific values for the number of recursion calls and it is unable to deal with multiple recursion calls such as in the Fibonacci sequence program. Attempts to prove a nested recursion also did not go well as even a simple program was not able to be validated by Z3.

It was also disappointing that Z3 was unable to deal with a simple induction proof, which might have been very helpful with different programs, especially when dealing with recursion. This was a result of Z3 not being able to use the provided assumptions which is crucial for induction proofs.

Chapter 7

In Closing

7.1 Z3 and Automatic program verification

Program Verification using a SMT solver like Z3 and symbolic calculation allows the creation of a model of the program in order to check its validity. The model provides two predicates that require validation in an SMT solver. The first predicate can validate that for any set of input parameters x (x in precondition) the program returns a value, i.e. $\forall x.prog(x)$. The second predicate can validate that for any set of input parameters x (x in precondition) the postcondition should be attainable, i.e. $\forall x.goal(x)$. The ultimate goal of program verification is to check if the program returns the correct value, which can be expressed by the following: $\forall x.prog(x) = goal(x)$.

Z3 is unable to validate such predicates. It requires the x value to be given in order to conclude if the program is correct. In formal terms, we have to replace a general predicate expression with a more specific propositional expression. When validating any of the examples shown in this thesis, a specific input parameter had to be specified.

If the precondition allows only a small set of x values, then it is possible to validate the program by executing the statements for each x, however in case of big or infinite sets this is not possible. In such cases we can only sample specific data that might be a good representation of the set. The model cannot establish if the program works correctly for every x, but only for specific conditions. The samples of x can create test cases for the model, but will not be able to definitely prove the correctness of the program. The model appears to be a program checker as described in [7].

Other limitations of Z3 are the ability to work only on integers due to the Simplify format [14] which makes it unable to represent fractions or strings. The universal quantifiers are often not calculated properly if the equation is too complicated making it often invalid with no meaningful counterexample.

In some cases if the specific number x was too large, Z3 was unable to calculate it due to insufficient resources so that, for example the Sum of Array Elements had to be run for array of 50 items only. Larger arrays caused an error. There is also no known documentation on how to increase this items limit for Z3 to be able to show correctness of the program for higher numbers.

Z3 also does not have adequate ability to work with mathematical equations. These limitations include trouble with negative numbers which need to be specified explicitly and with simple calculations such as division, as no such notion exists in Z3. Similarly factorial needed to be defined in a mathematical way in order to be able to compare it with the factorial output of the program to ensure post conditions were met.

Working with arrays in Z3 was also very challenging. Besides the inability to work with an arbitrary length array, Z3 cannot, for example, sort an array if the items of the array are unknown.

Concurrent programs cannot be properly defined when we take into account that any program can be interrupted between arbitrary statements. When this was taken into account for concurrent programs, Z3 was returning invalid for programs that were valid under certain conditions.

It has been also discovered that Z3 does not take into account the assumptions about elements, for example when sorting array with elements a and b, such that $a > b$. The inability too work with the assumptions was also shown in the example of induction, when the inductive assumption was not properly used while proving.

It can be noticed that the verification of programs using Z3 with Simplify format has a very recursive structure. Z3 calls recursively each state in the calculation and reaches the initial state of the program which indicate the program termination. The tool was able to prove simple recursive programs, however more complicated programs (that use multiple or nested recursive calls) came back to be invalid due to the complexity of calculations Z3 had to perform.

One of the main requirements of a prover is to show the program is correct, and if not, provide guidance to where the program fails. The error messages provided by Z3 are hard to understand regarding where the syntax problem appeared and it is

time consuming to find it in more complicated programs. The prover rarely shows a counter example, which is ambiguous to the user and hard to understand so that fixing the code is impossible based on the lack of counter examples.

Z3 is not the easiest prover to use for its lisp-like syntax which does not resemble a program in a way that most programmers understand. It is useful to use a bracket matching text editor to avoid syntax errors caused by missing brackets.

7.2 Future Work

It is important for a prover to prove a variety of different programs. It is still an issue that provers are not powerful enough to be able to prove complex programs [21]. Z3 still requires a lot of work in order to be able to accomplish the task of program verification. The complexity of programs is often not understood by the prover and it is assumed that the user is knowledgable about the program and the problem the program solves [23] in order to aid the prover.

Many programs that are developed in the industry are based on math, structures and different data types. One limitation of Z3 is that it can only work with integers and it is unable to do simple mathematical equations which can be easily handled by Mathematica or Maple. For example, a definition of factorial had to be constructed, which makes this verification tool dependent on external sources to produce a successful validation. Z3 can be combined with programs such as Maple or Mathematica to help with proving [18].

It has been noted that computer programs are inductive definitions of programs and can be proved by induction [41] [10]. Inductive structure occurs naturally in the structure of the program and yields the conditions for verifications of the properties. Inductive assertions reflect the way we understand the program. Large complex programs must be constructed in ways that do not violate the limits of human mind to manage complexity [27]. This observation might be very helpful in proving programs. Improvements to the Z3 induction would need to be added in order to be able to prove a variety of different programs.

Program verification is not inexpensive [27]. Formal testing is still viewed as less costly [10]. Program verification can be a substitution for testing. As in program testing, Z3 runs the program on test inputs and verifies if the expected output is reached [7]. It might be beneficial to use Z3 as a testing tool for programs when the program is represented symbolically and then run on various inputs as the program

specifications dictate.

In order to think of Z3 as a prover it is important to understand what kinds of relations are able to be validated in Z3, especially, to know exactly when this technique is closer to proving then testing or checking. More research examples are required to understand the possibilities and limits of Z3 when verifying programs by calculating relations. Secondly, since manual translations of programs into the Simplify syntax is prone to human errors, an automatic compiler and solver might be introduced similar to the program developed in [46]. Finally, using both Z3 and Maple tools would definitely lead to interesting results, however building a proper interface would be difficult and problematic.

Bibliography

[1] I. Adler and N. Megiddo, "A simplex algorithm whose average number of steps is bounded between two quadratic functions of the smaller dimension," *J. ACM*, vol. 32, no. 4, pp. 871–895, 1985.

[2] R. Alur and T. A. Henzinger, "Finitary Fairness," *ACM Transactions on Programming Languages*, vol. 20, no. 6, pp. 1171–1194, 1998.

[3] B. Anton and Z. Stachniak, "Substitutional definition of satisfiability in classical propositional logic," *Theory of applications and satisfiability testing*, vol. 3569, pp. 31–45, 2005.

[4] A. Bauer, M. Pister, and M. Tautschnig, "Tool-support for the analysis of hybrid systems and models," in *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, (San Jose, CA, USA), pp. 924–929, EDA Consortium, 2007.

[5] H. Bekic, "Definable operations in general algebras, and the theory of automata and flowcharts.," in *Technical Report*, IBM Laboratory, 1969.

[6] A. Blikle, "An analysis of programs by algebraic means," in *Mathematical Foundation of Computer Science*, pp. 167–213, Banach Center Publications, 1997.

[7] M. Blum and S. Kannan, "Designing programs that check their work," *J. ACM*, vol. 42, no. 1, pp. 269–291, 1995.

[8] J. Carette and R. Janicki, "Computing Properties of Numerical Imperative Programs by Symbolic Computation," *Fundam. Inf.*, vol. 80, no. 1-3, pp. 125–146, 2008.

[9] J. Carette, R. Janicki, and Y. Zhai, "Program verification by calculating relations."

[10] R. Cartwright, "Formal program testing," in *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, (New York, NY, USA), pp. 125–132, ACM, 1981.

[11] G. B. Dantzig, *Linear programming and extensions.* Princeton University Press, 1963.

[12] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[13] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," *J. ACM*, vol. 7, no. 3, pp. 201–215, 1960.

[14] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: a theorem prover for program checking," *J. ACM*, vol. 52, no. 3, pp. 365–473, 2005.

[15] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: A Theorem Prover for Program Verification," *ACM Transactions on Programming Languagegs and Systems*, vol. 52, no. 3, pp. 365–473, 2005.

[16] E. W. Dijkstra, "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," *Communicatons of the ACM*, vol. 18, no. 8, pp. 453–457, 1975.

[17] E. W. Dijkstra and W. H. J. Feijen, *A Method of Programming.* Adison Wesley, 1988.

[18] D. K. Enric Rodriguez-Carbonell, "Program Verification Using Automatic Generation of Invariants," *Book Series Lecture Notes in Computer Science*, vol. 3407, pp. 325–340, 2005.

[19] W. H. J. Feijen and A. J. M. van Gasteren, *On a method of multiprogramming.* New York, NY, USA: Springer-Verlag New York, Inc., 1999.

[20] R. W. Floyd, "Assigning meaning to programs," in *Mathematical aspects of computer science: Proc. American Mathematics Soc. symposia* (J. T. Schwartz, ed.), vol. 19, (Providence RI), pp. 19–31, American Mathematical Society, 1967.

[21] M. Foley and C. A. R. Hoare, "Proof of a Recursive Program: Quicksort," *Comput. J.*, vol. 14, no. 4, pp. 391–395, 1971.

[22] J. Franco, "Some interesting research directions in satisfiability," *Annals of Mathematics and Artificial Intelligence*, vol. 28, no. 1-4, pp. 7–15, 2000.

[23] D. I. Good, R. L. London, and W. W. Bledsoe, "An interactive program verification system," in *Proceedings of the international conference on Reliable software*, (New York, NY, USA), pp. 482–492, ACM, 1975.

[24] C. A. R. Hoare, "An Aximoatic Basis for Computer Programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–583, 1969.

[25] R. Janicki and E. Romanowicz, "Proving properties of programs with theorem provers. Experiments with Z3 and Simplify," *Proceedings of The 2009 International Conference on Software Engineering Research and Practice*, vol. 1, pp. 10–16, 2009.

[26] T. Jebelean, L. Kovacs, and N. Popov, "Experimental Program Verification in the Theorema System," *Int. Journal on Software Tools for Technology Transfer (STTT)*, 2006. in press.

[27] R. L. London, "A view of program verification," in *Proceedings of the international conference on Reliable software*, (New York, NY, USA), pp. 534–545, ACM, 1975.

[28] Z. Manna, *Methematical Theory of Computation.* 1974.

[29] Z. Manna and J. Vuillemin, "Fixpoint Approach to the Theory of Computation," in *ICALP*, pp. 273–291, 1972.

[30] A. W. Mazurkiewicz, "Proving Algorithms by Tail Functions," *Information and Control*, vol. 18, no. 3, pp. 220–226, 1971.

[31] A. Mili and J. Desharnais, "A system for classifying program verification methods: Assigning meanings to program verification methods," in *ICSE '84: Proceedings of the 7th international conference on Software engineering*, (Piscataway, NJ, USA), pp. 499–509, IEEE Press, 1984.

[32] M. B. Monagan, K. O. Geddes, K. M. Heal, G. Labahn, and S. M. Vorkoetter, *Maple Programming Guide.* 1998.

[33] L. Moura and N. Bjorner, "Efficient E-Matching for SMT Solvers," in *CADE-21: Proceedings of the 21st international conference on Automated Deduction*, (Berlin, Heidelberg), pp. 183–198, Springer-Verlag, 2007.

[34] L. D. Moura and N. Bjrner, "Z3: An Efficient SMT Solver," in *In Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS*, 2008.

[35] G. J. Myers, *The Art of Software Testing.* J. Wiley, 2004.

[36] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)," *J. ACM*, vol. 53, no. 6, pp. 937–977, 2006.

[37] T. Nipkow, "Hoare Logics for Recursive Procedures and Unbounded Nondeterminism," in *Computer Science Logic (CSL 2002)* (J. Bradfield, ed.), vol. 2471 of *LNCS*, pp. 103–119, Springer, 2002.

[38] S. Owicki and D. Gries, "Verifying properties of parallel programs: an axiomatic approach," *Commun. ACM*, vol. 19, no. 5, pp. 279–285, 1976.

[39] N. Popov and T. Jebelean, "Proving Termination of Recursive Programs by Matching Against Simplified Program Versions and Construction of Specialized Libraries in Theorema," in *Proceedings of 9-th International Workshop on Termination* (D. Hofbauer and A. Serebrenik, eds.), (Paris, France), pp. 48–52, June 2007.

[40] N. Popov, "A Practical Approach to Verification of Recursive Programs in Theorema," February 2004. Technical report 04-06, Institute e-Austria Timisoara (www.ieat.ro). Contributed talk at Computer Aided Verification of Information Systems (CAVIS-04), Timisoara, Romania.

[41] C. Reynolds and R. Yeh, "Induction as the basis for program verification," in *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, (Los Alamitos, CA, USA), p. 389, IEEE Computer Society Press, 1976.

[42] H. Wasserman and M. Blum, "Software reliability via run-time result-checking," *J. ACM*, vol. 44, no. 6, pp. 826–849, 1997.

[43] S. Wolfram, *The Mathematica Book*. Cambridge University Press, 1999.

[44] R. T. Yeh, "An approach to program verification," in *DAC '76: Proceedings of the 13th Design Automation Conference*, (New York, NY, USA), pp. 295–300, ACM, 1976.

[45] E. L. Yushchenko and I. V. Kasatkina, "Current methods for proving program correctness," *Journal Cybernetics and Systems Analysis*, vol. 16, no. 6, pp. 832–861, 1980.

[46] Y. Zhai, "Symbolic Execution Tool for Program Verification," (http://www.cas.mcmaster.ca/cas724/2007/tool/index.html).

[47] Y. Zhai, "An Analysis of Programs by Symbolic Computations," in *Master Thesis*, Dept. of Computing and Software, 2006.