Universitetet
i Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study program/specialization: *Computer Science*	Spring semester, 2016 Open / ~~Confidential~~
Author: *Andreas Bjerga*	Andreas Bjerga …………………………………… (signature author)
Instructor: *Hein Meling* Supervisor(s): *Hein Meling*	
Title of Master's Thesis: *AutoTest: Automatic Test Case Generation for Go*	
ECTS: *30*	
Subject headings: *Test Case Generation · Software Testing · Symbolic Execution · Constraint Solving · Autograder · Go · Abstract Syntax Tree · AutoTest*	Pages: 89 + attachments/other: code (zip file) *Stavanger, 15.06.2016* Date/year

AutoTest

Automatic Test Case Generation for Go

Andreas Bjerga

June 2016

University of Stavanger

Department of Electrical Engineering and Computer Science
Faculty of Science and Technology
University of Stavanger

Abstract

Autograder is a tool developed at the University of Stavanger that provides immediate feedback on programming assignments. The assignments are correctly automatically based on test cases written manually by the teacher. Writing tests manually is a tedious and time-consuming task that typically accounts for around 50% in the development of a modern application.

In this thesis we present a simple prototype of a tool named AutoTest; a static test generation tool for Go. The long-term objective of the project is to be able to automatically generate tests for typical assignments given on Autograder. The prototype combines techniques from static symbolic execution with constraint solving and a path exploration algorithm in order to find as many execution paths as possible and generate test inputs.

AutoTest in it's current state is far from complete. However the results from testing the prototype looks very promising. Additional functionality must be implemented before the tool can be tested thoroughly and before it can be used along Autograder.

Acknowledgments

I would like to thank my supervisor, Professor Hein Meling, for his guidance and feedback throughout this thesis. The weekly meetings have been very helpful to make sure that I've been on the right path. In addition, he has always been available and quick to respond whenever I've run into any questions regarding my research or writing.

I would also like to thank my wife for her support and constant encouragement during all my years of studying, and especially through this thesis. Thank you for your attention during all my frustrating moments, despite not understanding half of what I'm talking about.

Contents

1 Introduction

Autograder is a tool developed at the University of Stavanger that provides instantaneous feedback on programming assignments with the goal of improving student learning. The assignments are corrected automatically based on test cases and the students get score on their submission according to how many test cases that passed.

1.1 Motivation

Although Autograder is a nice addition to educational tools for computer science students, it currently requires the teacher to write all the tests for an assignment manually. Writing tests manually is a tedious task that can be very time-consuming. As a matter of fact, it typically acounts for around 50% of the costs in a modern software development company [1]. It can also be difficult, or even impossible, to cover all of a program's functionality by writing tests manually. If this could be done automatically instead, it could save the teacher from a lot of work. In addition, with modern techniques it is possible to generate tests automatically that can provide a greater code coverage than writing tests manually.

In this thesis we present AutoTest; a static test case generation tool that use techniques from symbolic execution in order to find as many execution paths as possible and generate a unique test for each path. The tool is currently a prototype and so far only works for basic examples. More functionality needs to be added before the tool can be used for real life applications.

1.2 Objective

The long-term objective of the AutoTest-project is to make an extension to the Autograder-platform that can generate test cases for a general solution automatically. The test cases will be generated based on an assignment's solution which must be given as input. Based on the long-term objective we have defined some subgoals. Not all of them are subjects for this thesis.

Most of the topics related to automatic test generation are outside the scope of what has been thought at the University of Stavanger. As a consequence, it required an extensive research period into new advanced topics. In this thesis we will delve into the topic of test generation and give a presentation of the modern techniques used for generating test cases automatically. We will use those techniques in order to generate tests for a simple application and measure the code coverage achieved. At a later point, when more functionality has been added, we hope to be able to generate tests for basic functions from the Paxos algorithm. A fully-featured, general test generator is outside the scope of this thesis and would probably be more suitable as a PhD-thesis.

Considering that this thesis is more about exploring the capabilities of automatic test generation than actually creating a fully-featured tool, the tool created here will not be able to handle every kind of code. More precisely, it will only support basic conditional statements and assignments. In addition, loops in an application has been excluded from consideration since it increases the complexity considerably.

1.3 Structure

The thesis is organized as follows:

Chapter 2 provides background information that is necessary to understand before continuing through this thesis. The chapter includes an introduction to software testing, symbolic execution, concolic testing, constraint solving, Autograder and a small introduction of the Go programming language.

Chapter 3 presents previous work in the area of symbolic execution and constraint solving.

Chapter 4 gives an overview of the design and architecture of AutoTest.

Chapter 5 contains details of the implementation of the tool.

Chapter 6 presents the structure of the tests produced and the results achieved.

Chapter 7 discuss the results achieved, the challenges encountered and the limitations of AutoTest.

Chapter 8 provides suggestions on further work that can be done to improve the tool.

Chapter 9 contains the conclusion of the thesis.

2 | Background

2.1 Software Testing

Software testing is an important tool that is used to improve the quality of a software application by checking that it works as expected and that it meets the requirements that was determined during the design process. Testing software properly is a necessity as releasing an unreliable software application may lead to customers losing confidence in the software company that produced it. As mentioned in the introduction, it consumes almost half of development efforts in modern software products. Even after an application is released there is a growing need for continued development and maintenance which also requires testing to verify that the systems work after changes are made.

There are generally two forms for testing [2][3]: *static testing* and *dynamic testing. Static testing* is carried out without executing the application being tested. Typical examples of static testing are walkthroughs, code inspections, or the use of a static code analysis tool. Static testing is done continuously by the programmers and this is a cost-effective method of dealing with bugs, but it is often more effective when an individual, or a group of individuals, who didn't write the code examines the code. The goal is to find defects or to check for missing requirements at a low cost.

On the other hand, *dynamic testing* requires one or more executions of the application under test with real inputs. In other words, dynamic testing requires the application to be compiled and run. One of the strengths with dynamic testing is that it is capable of finding bugs that are too complicated for static analysis too reveal. However, dynamic testing will only find bugs in the parts of the code that are executed based on the test input.

There are generally four levels of dynamic testing which are used at different levels in the development process [4].

Unit Testing tests the functionality of the smallest piece of software, for example a function or a component. It is written by the developers and is used to make sure that the smallest "building blocks" in the application works as intended.

Integration Testing is used to make sure that the units can interact with each other when they are combined into a larger structure.

System Testing (aka end-to-end testing) tests the entire system to make sure that it meets it's requirements.

Acceptance Testing is carried out by the customers or users, after the system testing is finished. The goal is to check that the required functions and features are satisfied.

Another approach to classify testing is also discussed in [4]. When the tests are designed from the system requirements, without any knowledge of the internal structure of the program, we call it *black-box testing*. The tester does not need to have any programming knowledge, he only needs to know *what* a certain function should do, not *how* it works internally. This method of testing can be applied to all the levels discussed above. In the other method, called *white-box testing*, the tester needs both programming and implementation knowledge. White-box testing test the internal structures of an application and are mainly applicable to unit and integration testing, but can also be applied to the system level.

2.1.1 Test Automation Frameworks

Test automation is the use of a software application that executes tests and then compares the actual outcomes from the test with the predicted outcome to check if they are the same. Test automation gives us the ability to test a piece of software quickly and thoroughly, with an increased test coverage. It can save software companies a lot of money and time, and even discover defects that manual testing cannot expose. When changes to the system are introduced, the tests can be run again to check that the new changes did not break the code that worked before the modifications were made. A *test automation framework*[5][6] is an execution environment for automated tests defined by a set of assumptions, concepts and practices. It defines the format that is used to create the tests, executes the tests and reports the result. The framework is application dependent and it can also be modified to meet a specific requirement for testing an application.

There are different kinds of testing frameworks and we will give a short description for some of them below.

Test Script Modularity: Small independent scripts are needed to represent modules or functions of the application being tested. The small scripts will then be used in a hierarchical fashion in order to construct larger test cases. This framework provides an abstraction layer that hides a component from the rest of the application such that changes made to other parts of the application will not affect the component.

Data-driven: Sometimes it is necessary to test the same function multiple times but with different input data. Data-driven testing often stores input data and expected output data in a separate file and utilizes a test script that will run through all the tests with the input data for the given function.

Keyword-driven: In addition to keeping the input data in a separated file, it also keeps a certain part of code belonging to the test script in another data file. The code that is stored is given a keyword that explains the action that is performed on the application. This framework is also known as *Table-driven framework* since the keywords and input data for the tests are stored in a table.

Hybrid: A combination of the frameworks discussed above. It leverages all of the benefits from all of the associated frameworks but is more complex.

Model-based: Model-based testing[7] is a framework that can generate executable tests with input values based on a model of the expected behavior of the system. It can automate the complete test design, given a suitable model. Model-based testing is a form of black-box testing. You only need to know how the system is expected to work and create a model based on the expected behavior. Then, the model-based testing tools will automatically generate tests from that model.

All of the frameworks discussed above can be used to save money and to give an increase in the code coverage, but none of them can automatically generate tests cases and inputs based on nothing more than the source files. In the next section we will introduce an advanced technique that can help us achieve this goal.

2.2 Symbolic Execution

Symbolic execution is a static program analysis technique that was first introduced in the 70s [8]. Since the 70s there has been a major increase in computational power and new constraint solving techniques that has led to a renewed interest in symbolic execution.

Today it is used in various popular testing tools, both open-source and commercial tools.

Symbolic execution aims to find as many different program paths as possible, and to generate a set of concrete test inputs for each path [9]. During execution it will also try to find different kinds of errors, like assertion violations, uncaught exceptions, security vulnerabilities and memory corruption. In other words, it is a technique that can generate tests with high code coverage while it also provides concrete inputs that may trigger bugs.

Instead of using actual concrete values as input, symbolic execution uses symbolic expressions to represent the program variables. The program variables are mapped to symbolic expressions, that can represent any concrete value, and stores them in a symbolic state, σ.

Let's take a look at the code in Listing 2.1, which is borrowed from an example in [10]. The code shows a simple function that takes two variables, x and y, as input, and swaps the integer-values for the variables only if x is larger than y. First, the input parameters x and y are initialized to the symbolic values X and Y, respectively. At every assignment operation, the symbolic values of x and y are updated. After line 3, the state contains the mapping $\sigma = \{x \rightarrow X + Y, y \rightarrow Y\}$. Variable y is then updated in line 4 to $X - Y$. Since the map contains the variable x, which is defined in the state, we replace the variable with the symbolic expression. The new mapping then becomes $\sigma = \{x \rightarrow X + Y, y \rightarrow X\}$.

Symbolic execution also includes a *path condition*, PC, which stores the conditional expressions over the symbolic variables that is encountered during execution. As a program is symbolically executed, conditions are gathered and stored in the PC. When the execution is finished, you are left with a collection of conditions, that must be satisfied in order for the program to traverse the specific path. The collected constraints are then solved using a *constraint solver* (see Section 2.4) and the solution from the solver

```go
 func swap(x, y int) (int, int, error) {
   if x > y {
      x = x + y
      y = x - y
      x = x - y
      if x-y > 0 {
         return 0, 0, errors.New("Error during swap!")
      }
   }
return x, y, nil
 }
```

Listing 2.1: Code that swaps two integers

forms the test inputs. The program will follow the exact same path as the symbolic execution, and terminate in the same way, if the application is executed on the concrete inputs from the solution.

Figure 2.1 shows the corresponding symbolic execution tree of Listing 2.1. *PC* is first initialized to `true`. When symbolic execution meets a conditional statement, like an if-else statement, the *PC* is updated for the "then" branch, and a fresh path constraint PC' is created for the "else" branch. Symbolic execution will continue to execute along each branch if the path constraint is satisfiable for both branches; that is, if there exist concrete values that can satisfy the path constraints. If *PC* or PC' is not satisfiable, symbolic execution will terminate along the unsatisfiable branch and the path constraint will become `false`. An example of this can be seen in step 5 in Figure 2.1.

2.2.1 Limitations

For very large programs, the number of feasible paths grows exponentially with an increase in program size. For programs that contains loops there can even be an infinite amount of execution paths. This is known as the *path explosion problem* [9]. To deal with this issue, we could put a limit on the number of paths ex-

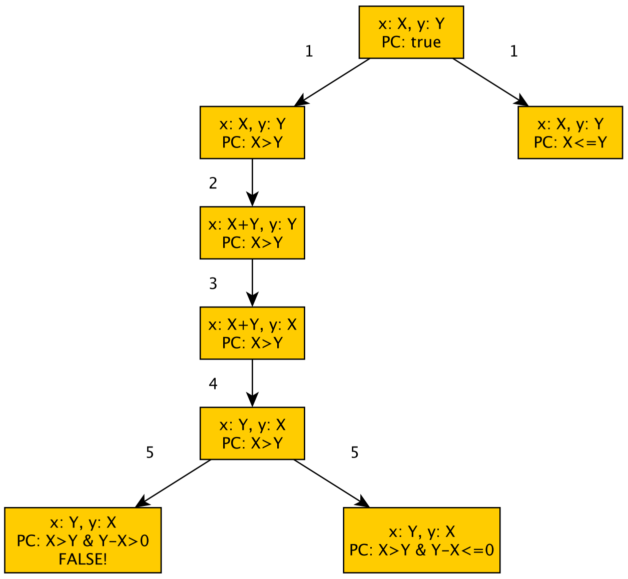


Figure 2.1: Symbolic execution tree for two integers that are swapped

plored, loop iterations or exploration depth. This will of course result in some execution paths not being explored, which is also a problem since we want to explore as many paths as possible. One way modern symbolic execution tools address this problem is by using search heuristics that prioritize paths that cover more code than others.

Another problem with symbolic execution is that if the symbolic path constraint along an execution path contains expressions that can't be solved by a constraint solver, then it cannot generate an input [11]. One such case could be if the path constraint contains a condition with a function call to operating-system or library functions. The code for these functions are unreachable for the solver, thus it is impossible for the solver to generate specific values that satisfies the constraint. *Concolic testing*, discussed in the next section, is a technique that address this problem.

2.3 Concolic Testing

Concolic testing, also called *dynamic symbolic execution*, maintains both concrete and symbolic state and performs symbolic execution dynamically, along a concrete execution path [10]. This is a technique that combines static and dynamic analysis techniques. The concept was first introduced in [12], but the term *concolic testing* was first introduced in [13].

Concolic testing starts with some given or random input and while it executes, it will collect the conditions encountered along the executed path and save them to the PC. The information stored in PC can then be used to find alternative paths. For example, when the execution hits an `if-else` statement, the concrete execution may lead to the `then`-statement to be executed. It is also possible that the program can execute through the `else`-statement. To guide the program along this alternative path we can negate the condition that caused the program to execute through the `then`-statement. A constraint solver will then be used to find the values (test inputs) that will guide the execution to this path and the program is then executed on those inputs. Conditions will be gathered along the new path, and the process will be repeated until all execution paths are found or until a certain limit is reached.

One of the main advantages of concolic testing [14] is that whenever symbolic execution fails to generate inputs for a path, typically because the PC contains an external function call, then dynamic symbolic execution can use concrete values as input to the function in order to see what values it returns. This way, the conditions can be simplified whenever symbolic execution is unable to reason about a condition. Accordingly, concolic testing can be seen as an extension of symbolic execution that includes runtime information. This makes it more powerful than static symbolic execution and more applicable for realistic programs.

2.4 Constraint Solving

The term *constraint solving* is also known as the constraint satisfaction problem (CSP), a problem where the goal is to check if there exist an assignment to a set of variables that satisfies a set of constraints [15]. CSPs appears in a number of fields related to computer science: software and hardware verification, type inference, static program analysis, test-case generation, scheduling, planning, graph problems, and more [16]. CSPs is a large topic that relies on a lot of heavy mathematics. In this section we will give a short introduction to constraint solvers and CSPs. We will only explain the information that is necessary to understand this thesis and not go into detail about the mathematics it relies upon.

So what does satisfiability mean? In [15] it is defined as follows:

Definition 2.1. *A formula is satisfiable if there exists an assignment of its variables under which the formula evaluates to* `true`*.*

Boolean satisfiability problem, better known as *propositional satisfiability* or SAT, is the most common form of CSP. The goal of SAT is to determine if a formula consisting of boolean variables and logical connectives ("$\wedge$", "$\neg$", "$\vee$"), has an assignment of `true` or `false` values that evaluates the formula to `true`. Not all problems can be defined by boolean variables and logical connectives. They need a more expressive logic, such as *first-order logic*. First-order logic[15] is a formula that consists of:

1. *Variables*
2. *Logical connectives*: e.g., "$\wedge$", "$\neg$", "$\vee$"
3. *Quantifiers*: "$\exists$" and "$\forall$"
4. *Nonlogical symbols*: function, predicate, and constant symbols
5. *Syntax*: rules for composing formulas

First-order logic is like a framework that provides a common syntax. On top of this framework there are *theories* (see Section 2.4.1), defining restrictions on the nonlogical symbols and the interpretation of the symbols. Solvers for these theories are called *Satisfiability Modulo Theories* (SMT). Nearly all SMT problems are very complex and requires a lot of computing power, but the number of problems that can be solved are always increasing, due to technical innovations and the annual competitions SMT-COMP [17] and the international SAT competitions [18]. Newer SAT solvers are able to solve formulas with more than hundreds of thousands of variables. The more common theories for SMT solvers have developed in a similar manner. For SMT solvers there are also a common input/output language called SMT-LIB [19] that makes it easier to run benchmarks across different solvers.

Most SMT solvers are built on top of effective SAT solvers. The atomic formulas (a single condition) in an SMT formula are mapped into boolean variables. The SAT solver can then check the new formula for satisfiability. The SMT formula will be unsatisfiable if the SAT solver finds the new formula to be unsatisfiable. However, if the SAT solver finds the formula to be satisfiable, a theory specific solver that is part of the SMT solver is used to check if the combination of predicates is satisfiable according to the theory restrictions. An SMT solver is usually a collection of theory solvers. In the next section we will take a look at some of the different theories and in Section 3.2 we will present some SMT solvers.

2.4.1 Theories

Below is a brief description of some of the theories. A deep understanding of them is not necessary for this thesis. The interested reader is referred to [15], [20] or [21] for more information.

Propositional logic: the variables are booleans and the operators allowed are OR ($\vee$), AND ($\wedge$) and NOT ($\neg$).

Linear arithmetic: variables can be either integers or reals and the predicate symbols allowed are $\{\leq\}$. The functions allowed are $\{+, -, \times\}$, but multiplication is only allowed with a numeric constant.

Difference arithmetic: is a fraction of linear arithmetic where each predicate is rewritten to be of the form $x - y \leq c$, where c is a constant, and x and y are variables. The variables can range over either integers or reals.

Bit-vectors: this theory is useful for modeling hardware or low-level software as information in computer systems are encoded as bit-vectors. The function allowed are $\{+, -, \times, /, \ll, \gg, \&, |, \oplus, \circ\}$, where "$\ll$" and "$\gg$" are left and right shifts, and "$\circ$" are concatenation of bit vectors.

Arrays: allows formulas that include arrays. The functions allowed on the arrays are $read(a, i)$ and $write(a, i, v)$. The first function reads the value at index i from the array a, and the second function updates the value at index i to v in the array a.

2.5 The Go Programming Language

AutoTest is implemented in the Go programming language. Go is a relatively new language that features some concepts that are not part in some of the more commonly used languages. For this reason a short introduction about Go is given here.

Google developed the language in 2007 and it was later released as an open-source project in 2009[22]. Go is often thought of as a systems programming language and it was designed to be an efficient, scalable and productive language. The development started as Google faced some challenges working with large hardware and software projects in existing programming languages, and the goal was to get rid of the slowness and clumsiness of developing large-scale software at Google. According to [23], Google faced the following challenges, which became the key objectives in the design of Go:

- slow builds
- uncontrolled dependencies
- many programmers used tweaked versions of the languages
- poor program understanding leading to code that is difficult to read and poor documentation of the code
- updates were costly
- challenging to write automatic tools
- cross-language builds

The result is a compact and modern language, with a clean syntax, that features a garbage-collector and that has built-in support for concurrent programming with the use of *goroutines* and *channels.* A goroutine is, as stated in [24], "a function that executes concurrently with other goroutines in the same address space". By writing `go` before a function or method call you run the call in a new goroutine. A channel is used such that two functions that are executed concurrently can communicate by sending values of a specified type on the channel. A channel can be either buffered or unbuffered and it can be used to either send a value on the channel, or as a receiver that will block the execution until values are received. This way of communicating amongst concurrent processes is based on A Theory of Communicating Sequential Processes [25] from 1984.

2.5.1 Go and the Abstract Syntax Tree

In order to find all execution paths in an application and create tests for each of them, we create an abstract syntax tree (AST) and analyze the tree structure. An *abstract syntax tree* (AST) takes some source code as input and creates a tree representation of the essential structure of the code [26]. It is abstract in the sense that it omits unnecessary syntactic details, such as grouping parentheses or semi-colons to terminate statements. It's not necessary to include them in the tree, because they are directly represented in the structure of the tree. As an example, Figure 2.2 shows a simple AST of the statement `x := y+(z*3)`.

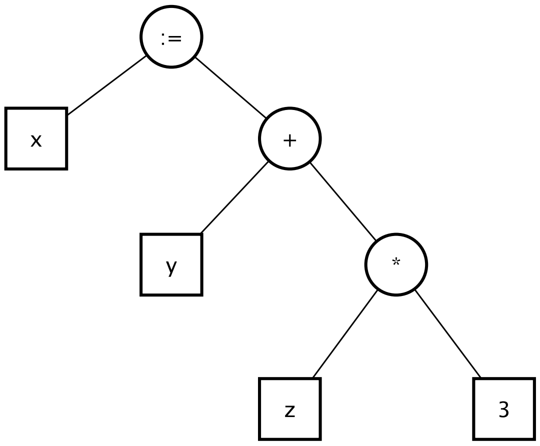


Figure 2.2: An example of a simple abstract syntax tree

Go provides a package called *ast* which declares the types used to represent ASTs for Go source code and it also contains functions to traverse the tree in a depth-first search fashion. Moreover, Go's *parser*-package contains functions that can take a directory, file, or a string of source code as input and then converts the source code into an AST. Listing 2.2 shows the output from parsing the same statement as mentioned above to an AST in Go. The tree contains a lot of information and is quite large for only a single-line statement. In Section 5.4 we will look into how a tree like this can be traversed to retrieve interesting information.

```
*ast.AssignStmt {
.  Lhs: []ast.Expr (len = 1) {
.  .  0: *ast.Ident {
.  .  .  NamePos: 8:2
.  .  .  Name: "x"
.  .  .  Obj: *ast.Object {
.  .  .  .  Kind: var
.  .  .  .  Name: "x"
.  .  .  .  Decl: *(obj @ 43)
.  .  .  }
.  .  }
.  }
.  TokPos: 8:4
.  Tok: :=
.  Rhs: []ast.Expr (len = 1) {
.  .  0: *ast.BinaryExpr {
.  .  .  X: *ast.Ident {
.  .  .  NamePos: 8:7
.  .  .  Name: "y"
.  .  .  }
.  .  .  OpPos: 8:9
.  .  .  Op: +
.  .  .  Y: *ast.ParenExpr {
.  .  .  .  Lparen: 8:11
.  .  .  .  X: *ast.BinaryExpr {
.  .  .  .  .  X: *ast.Ident {
.  .  .  .  .  .  NamePos: 8:12
.  .  .  .  .  .  Name: "z"
.  .  .  .  .  }
.  .  .  .  .  OpPos: 8:13
.  .  .  .  .  Op: *
.  .  .  .  .  Y: *ast.BasicLit {
.  .  .  .  .  .  ValuePos: 8:14
.  .  .  .  .  .  Kind: INT
.  .  .  .  .  .  Value: "3"
.  .  .  .  .  }
.  .  .  .  }
.  .  .  .  Rparen: 8:15
.  .  .  }
.  .  }
.  }
}
```

Listing 2.2: Output from parsing a simple statement to an AST

2.5.2 Go and C

In Section 3.2 we will present some of the constraint solvers currently used in the industry and for academic research. None of the constraint solvers has a Go API, but many of them has a C API. Go provides a package, *cgo*, which makes it possible to call C code in Go applications [27][28]. The first step in order to use C-code inside Go code is to import the pseudo-package "C". Function and variable declarations, include-statements and any other kind of C-code can be put in the *preamble*, which is a comment section located directly above the import statement. The preamble is used as a header when the C-part of the package is compiled. The names declared in the preamble may be used in Go code as if they actually were defined in the package C. Listing 2.3 shows an example where we define a C integer `i` in the preamble and in the Go code we convert it into a Go integer, and then print it out.

Flags for the compiler and linker for the C code can also be set in the preamble by using the `#cgo` directive. When the application is built, Go will concatenate all the `CFLAGS` directives in a package and use them to compile C files in that package. The same concatenation is done for the `LDFLAGS` directives which will be used at link time.

It is also possible to write Go code which can be used in C code, but this part of `cgo` is not used in this thesis so the details on how this is done will not be discussed here.

2.6 Autograder

Computer science courses at universities and colleges often consists of practical programming assignments. There might be times when a student struggles to grasp the concepts of a subject, and this in turn will make it difficult for the student to implement an assignment correctly. As the teachers usually have a pretty busy schedule, it is not always easy to receive the required feedback. This

```go
package main

/*
int i = 5;
*/
import "C"

import "fmt"

func main() {
  intc := int(C.i)
  fmt.Println("C int:", intc)
}

//Outputs: "C int: 5"
```

Listing 2.3: Cgo-example

may lead to students being uncertain and hesitating to continue programming before they have received feedback.

Autograder is the result of a master thesis given at the University of Stavanger where the goal was to improve student learning through rapid feedback and by encouraging self-learning [29]. Autograder does so by automatically correcting and grading programming assignments submitted by students based on test cases written by the teacher.

A simplified overview of Autograder and the components it interacts with is presented in Figure 2.3. When a student wants to get feedback on an assignment, he pushes the code to an online git repository at GitHub [30] that has been assigned to the student by the teacher. Autograder has a continuous integration service that is notified when new code has been transferred to GitHub. When it receives a notification that a repository has been updated, it immediately clones the students repository and the test repository. The two repositories are then merged and the test script is executed in Docker [31], a virtual machine environment. The result from the test script is then sent back to Autograder, which in turn passes the information on to a web service. Both students and teachers can then check the results and a detailed build log from the web service.

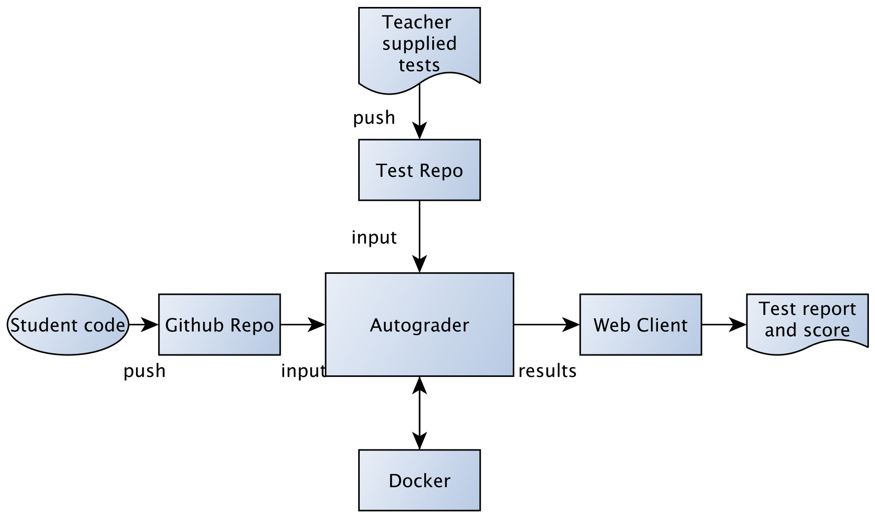


Figure 2.3: Overview of Autograder

Autograder was used in a master-level course at the time the thesis was written, and it is still used today in some courses. Autograder has proven to be a valuable tool, both for students and the teachers. Students now has the ability to get rapid feedback on their code, from anywhere in the world, at any time. The teachers now has access to an overview of the students progress that helps the teacher to recognize students that struggle. In addition, they no longer have to grade assignments manually.

3 | Related Work

This section will present some of the work that has been done related to symbolic execution and constraint solving. Parts of this work has been used as an inspiration for designing and implementing AutoTest. This thesis contributes by introducing a new symbolic execution tool, AutoTest, which is probably one of the first symbolic execution tool for Go.

3.1 Symbolic Execution Tools

This section introduces some of the symbolic execution tools that exists as of today. We will take a look at the architecture of the different tools and the techniques they use.

3.1.1 KLEE

KLEE [32] is an open-source, symbolic execution tool that can automatically generate test cases. In [33], the authors tested the tool on more than 450 applications, in total more than 430K lines of code. The tool was designed to work on a broad range of applications without alteration and with two goals in mind:

1. Hit every line of executable code in the program
2. At each dangerous operation, detect if any input value exists that could cause an error.

Pointer dereference and assertion operations are two examples of what are considered as dangerous operations. In order to realize the goals, KLEE uses symbolic execution to gather constraints along an execution path. When the path reaches an exit call, or when an error is found, KLEE solves the path's constraints with the constraint solver STP [34], and then produces a test case that will execute along the same path.

KLEE is run directly on bytecode generated by compiling the source code using the LLVM-compiler. This means that KLEE can be used to generate test cases for languages that are supported by the LLVM-compiler. These include C, C++, Objective-C amongst others [35].

Constraint solving is the foremost time-consuming task in a KLEE-run, but the authors have been able to decrease the time spent on constraint solving by simplifying expressions and eliminating queries before the constraint solving process. Without any optimizations, constraint solving accounts for on average over 90% of KLEE's running time. With the optimizations enabled, constraint solving accounts for around 40% of the execution time, which is a significant improvement! The query optimizations performed in KLEE are as follows:

Expression rewriting such that they are optimized for the compiler.

Constraint set simplification every time a new equality constraint is added to the constraint set, KLEE simplifies the constraint set.

Implied Value Concretization when a constraint is added to the set that suggests that a variable is concrete, KLEE will write the concrete value to memory. For example, if a constraint $x + 2 = 5$ is added to the set, KLEE will determine that $x = 3$.

Constraint independence by dividing constraint sets into disjoint independent subsets, KLEE can ignore irrelevant constraints before the query is sent to the constraint solver.

Counter-example cache maps constraint sets to counter-examples (i.e. variable assignments) and stores the mapping in a tree. This mapping can be searched efficiently for entries for subsets and supersets of a constraint set. This way of storing the cache introduces three new ways of eliminating queries.

The test cases KLEE produces achieve an impressive code coverage, on average over 90% on the applications evaluated in the paper. It is able to achieve high coverage with few test cases and it beats manual testing, often considerably. As an example, the authors ran KLEE on the GNU COREUTILS library, and the tests generated by KLEE surpassed the COREUTILS developers test suite by over 15%. This is a test suite that has been built up over a period of fifteen years, which tells something about how powerful symbolic execution can be!

3.1.2 SAGE

SAGE [36][11], short for Scalable Automated Guided Execution, is a tool that has been developed by Microsoft Research. The goal of the project was to address some of the limitations of *blackbox fuzz testing*, which is a form of blackbox random testing. Fuzz testing has shown to be a fast and easy approach for finding bugs in an application. The problem with blackbox fuzz testing is that it provides low code coverage. Say for instance that a simple application contains the statement `if x == 0`. The probability for the `then`-branch to be executed is one in 2^{32} when `x` is a randomly chosen 32-bit integer. This means that serious bugs could be easily overlooked by blackbox fuzz testing, and this is the problem that the developers of SAGE wanted to address.

SAGE uses a technique called whitebox fuzz testing. It starts with some initial input and symbolically executes a program while gathering constraints from conditional statements encountered along the execution. The constraints are then negated methodically and solved using a constraint solver (Z3). The solutions from the solver form inputs that will take different paths if executed.

One of SAGE's key components is its *generational search*, a directed search algorithm that is designed to maximize the code coverage by finding bugs as fast as possible. The basic idea of the algorithm is that it negates all the constraints in a path, one by one, and checks if the new path constraint is satisfiable with the constraint solver. The generational search algorithm makes it possible to generate thousands of new tests in a single symbolic execution, as opposed to the standard depth-first search where only the last constraints in the path is negated, generating only one new test per symbolic execution. Each new input generated by the generational search is also given a score based on their code coverage. The input with the highest score is the one that is symbolically executed next.

SAGE is not available to the public, but it has discovered many bugs in many large Microsoft applications. It found roughly one-third of all the bugs detected by file fuzzing throughout the development of Windows 7, and since 2008 it has been running non-stop on more than 100 machines, automatically fuzzing hundreds of applications. As with KLEE it applies a number of optimizations for the constraints. According to [36], SAGE was also the first tool that performs symbolic execution at the x86 binary level. This means that SAGE can be used on any kind of application no matter the programming language was used to make it.

3.1.3 Other Tools

In addition to KLEE and SAGE there are also a bunch of other symbolic execution tools. We will give a short presentation of some of them in this section.

CUTE (Concolic Unit Testing Engine) and jCUTE[13][37] are two other symbolic execution tools developed at the University of Illinois for C and Java applications. The tools are built up of mainly two modules: one that instruments the application and a library for symbolic execution, solving constraints and race detection. Instrumenting an application basically means that extra code is injected into the program. The extra code makes sure to call the library that performs symbolic execution during runtime. The solvers used in CUTE and jCUTE can handle arithmetic and pointer constraints.

Symbolic PathFinder (SPF)[38] is a tool built on top of Java Pathfinder[39]. Java PathFinder is an open-source tool developed by NASA that is used for verifying Java bytecode. It started as a model-checker for finding for example unhandled exceptions or deadlocks, or for collecting runtime information. Today it has many extensions, and Symbolic PathFinder is one of them. SPF extends the model-checking capabilities of Java PathFinder by combining it with symbolic execution in order to automatically create test cases. jFuzz[40], a concolic white-box fuzzer developed at MIT, is also built on top of Java PathFinder.

3.2 Constraint Solvers

Symbolic execution tools relies upon constraint solvers in order to generate inputs for each execution path. This section introduces some SMT solvers that was still under development in 2015, according to [41].

3.2.1 Z3

Z3 [42][43] is a free SMT solver developed at Microsoft Research and it was intended for dealing with problems related to software verification and software analysis through extended static testing and test case generation. It combines several of the theories discussed in Section 2.4. In 2007, Z3 won four categories in SMT-COMP [17], and got second place in seven other categories. It is used in a number of program analysis, verification and test-generation projects at Microsoft, including SAGE, which we discussed in Section 3.1.2. A user can communicate with the solver either using a textual format or through an API. The solver provides APIs for C, C++, .NET, Java and Python [44]. Among the textual formats Z3 supports are the SMT-LIB2 standard and Simplify. An overview of Z3's components are shown in Figure 3.1.

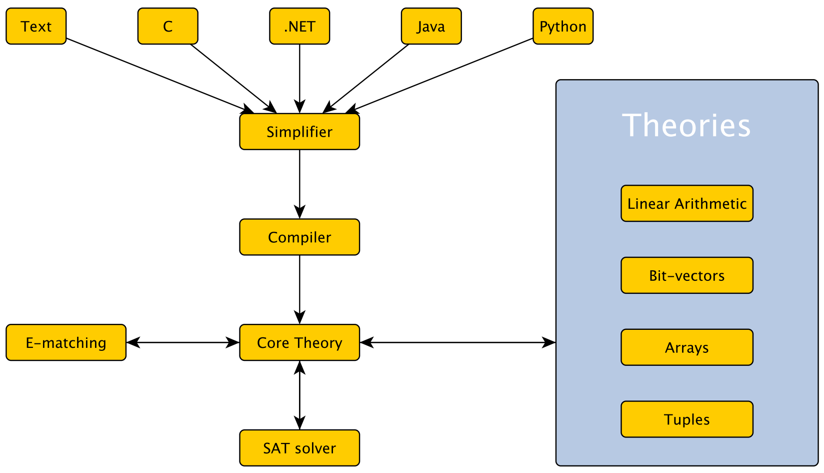


Figure 3.1: Overview of Z3

The simplifier reduces the input by using algebraic reduction rules. The compiler takes the simplified expression and converts it into a data structure that the core theory can work with. The core theory receives a satisfiable assignment for the variables from the SAT solver. Equality formulas are processed by the E-matching engine, which returns an *equality graph* (E-graph). Given an equality formula φ, the equality graph corresponding to φ is an undirected graph $(V, E_{=}, E_{\neq})$ where V represents the variables in φ, the edges $E_{=}$ represents the equality predicates, and the edges $E_{\neq}$ represents the inequality predicates [15]. As an example, say we receive a predicate $x = y$, the E-matching will search for an *equality path* between x and y, a path connected by $E_{=}$ edges. Nodes in the graph can point to more than one theory solver.

Z3 combines the theory solvers in order to solve a formula that cannot be expressed by a single theory.

3.2.2 Other Solvers

Simple Theorem Prover (STP) [45] [34] is an open-source theorem prover used by many companies and for academic purposes. It is based on a SAT solver called MiniSAT [46], which is also freely available, and it translates every formula into a propositional formula that is solved using MiniSAT.

STP was mainly designed for solving constraints of bit-vectors and arrays and it has won the bit-vector category in SMT-COMP two times, and received second place two times. It supports the textual formats SMT-LIB1 and SMT-LIB2 and provides APIs for C/C++ and Python.

Boolector is another SMT solver for the quantifier-free theories of bit-vectors and arrays [47][48]. It supports the SMT-LIB format and includes a *rewriting engine* that aims to reduce a given formula as much as possible. Since 2008, Boolector has received first place in SMT-COMP more than ten times, in the categories related to bit-vectors and arrays. This solver also features C and Python APIs.

Yices [49] is a theory solver that supports both versions of the SMT-LIB format, supplies a C API and also provides its own textual input language. Yices can decide satisfiability of formulas that contains atoms from the theory of uninterpreted function symbols with equality, linear arithmetic, bit-vectors, among others. Yices has also won several times in the categories it entered [50].

3.2.3 Combining SMT Solvers

Sometimes it can be difficult to select the SMT solver that works best for a specific application. There are formulas that specific solvers may be unable to solve and some solvers are also much faster than others to make a determination if a formula is satisfiable. *metaSMT* [51][52] is an open-source tool that address this problem by running multiple solvers in parallel. metaSMT can be thought of as an abstraction layer that allows different SMT solvers to be used in parallel, using one common API.

The architecture of metaSMT consists of three layers: a frontend, a middleend, and a backend layer. The frontend provides APIs for C++ and Python where the primitives of the SMT-LIB2 format can be defined. The middleend handles translations and optimizations on the formulas. The backend provides an interface that maps from the metaSMT API to APIs for SAT and STM solvers. In addition, there is also the option to send SMT-LIB2 commands directly to the solvers that supports this feature.

metaSMT has support for SMT solvers like Z3, Boolector, STP and SWORD [53] and SAT solvers like MiniSAT, PicoSAT [54] and AIGER [55]. Every command that is sent to the backend is either translated to SMT-LIB2 format and then sent to every available solver, or an API-call is made for each solver that correspond to the SMT-LIB2 command. Every solver runs the command in parallel until the fastest solver returns a result.

Since satisfiability is a NP-complete problem, the time it takes to check is based on each solvers heuristics and optimization techniques. In many situations it is impossible to say in advance which solver that will perform better. metaSMT provides an interface that allows use of multiple constraint solvers. KLEE (see Section

3.1.1) is an example of a dynamic symbolic execution tool that implements multi-solver supports through the metaSMT framework [56].

4 | Designing AutoTest

This section describes the architecture and the design choices we have made for AutoTest. As this tool is developed first and foremost for use in Autograder, we will first give an overview of how it works when a new assignment is given. We will then look further into the modules of the tool.

4.1 Overview

In order to use AutoTest to automatically create test cases for a programming assignment, the teacher first of all needs to create a solution for the assignment. The solution is assumed to be correct. The source code from the solution is used as input to AutoTest, as is shown in Figure 4.1. AutoTest will then run locally to produce test cases for the assignment. The teacher can then use the generated tests on Autograder in evaluating the students' submitted code.

AutoTest has a simple command-line interface that takes two parameters: an option parameter and a path. The two options are as follows: it can create tests for a single source file, or it can create tests for all source files in a directory. If option `-f` is used, the second argument should be the path to a file and AutoTest will try to parse the file into an AST. If option `-d` is used, the second argument is treated as the path to a directory.

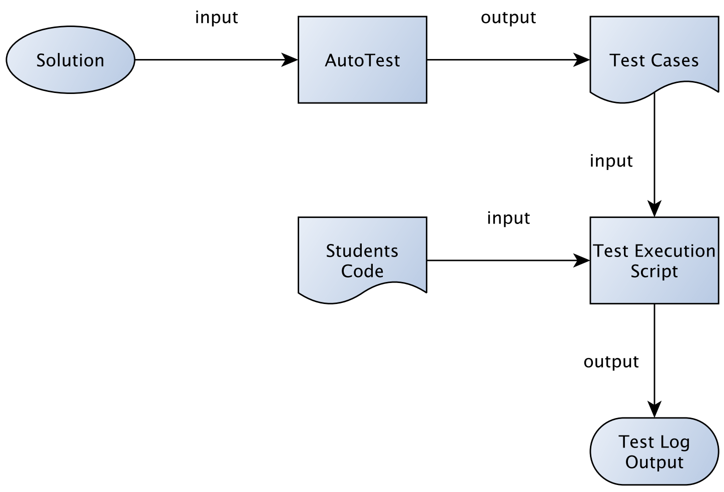


Figure 4.1: High-level overview of AutoTest

Figure 4.2 shows the different modules of AutoTest. The rest of this chapter will give a short presentation of each module and their purpose.

4.2 Parser

The Parser-module is the simplest of the modules. It is responsible of parsing the source code given as input into an AST. The input can either be a single file, or a directory. If the input is a directory, the parser will search the directory for all Go source files and parse each file. The PathFinder is then called for each file's AST.

4.3 PathFinder

The PathFinder will use the AST created from the Parser to retrieve information that we will need in order to create tests for a function. The information we are interested in are:

- Function name
- Input parameters

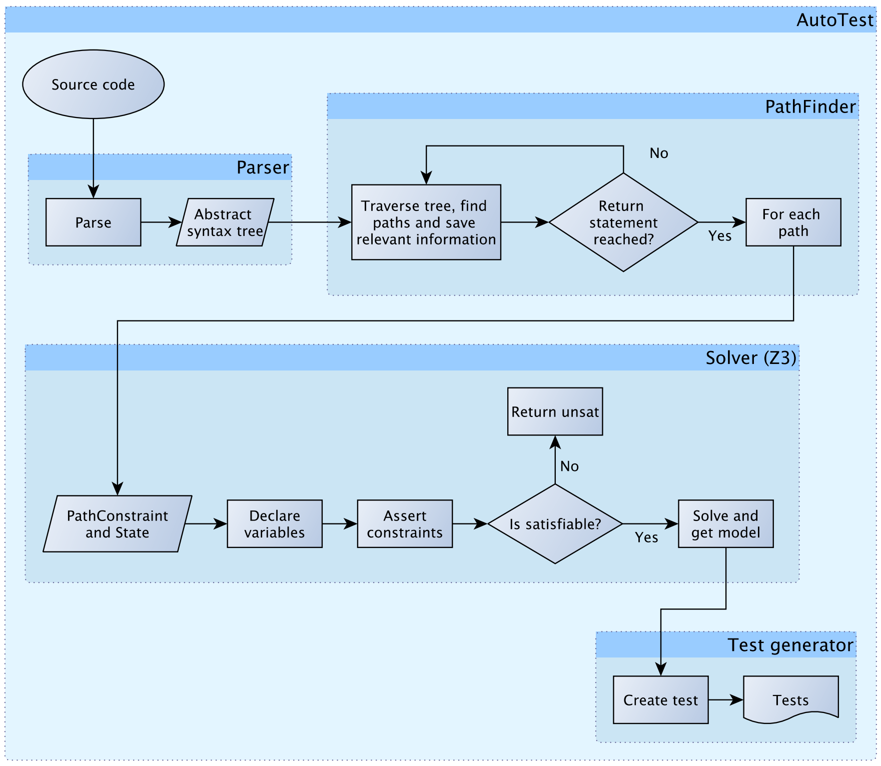


Figure 4.2: The modules of AutoTest

- Return types
- Conditions
- Variables

Function name, return types and input parameters are necessary in order to create the actual tests. In addition, constraints and variables are provided to the solver that returns input data for the tests.

The PathFinder performs symbolic execution on the AST using depth-first search in order to find as many paths as possible. Each execution path is represented by a state (a map of the variables encountered) and a path constraint (a collection of the conditions). As we move down a certain path in the tree, constraints are collected when conditional statements are encountered and variables and assignments are saved in the symbolic state. During traversal, information about the function name, input parameters and return types are also collected. When the function terminates, for example when a return statement is reached, the Solver will be called for each execution path found. The path constraint and state is then given as input to the Solver.

4.4 Solver

The Solver-module is responsible for checking if there exists a satisfiable solution for the given execution path. In this thesis, we use the constraint solver Z3 (see Section 3.2.1) to check for satisfiability. Since Z3 is an external library that does not support Go directly, we will use the C API provided by Z3 in combination with the *cgo*-package provided by Go. The Solver-module will loop through all the variables in the state, check what type they are, and declare them as symbolic variables through Z3's API. We will then loop through all the conditions in the path constraint and declare each constraint.

After both conditions and variables have been declared, the solver will try to find a feasible solution. If the path constraint is satisfiable, it returns a model with variable assignments that satisfies the constraint. The model forms the test inputs for each path, and if the function is executed on this input, it is guaranteed to take the exact same path as it did during symbolic execution. The Test Generator-module is called for each model returned from the Solver.

4.5 Test Generator

The Test Generator-module is responsible for creating test-files for each Go source file that has been analyzed by AutoTest. It creates a test-function for each function in a source file, where each test-function is like a small test-execution script. Each test-function is accompanied by a table that contains input values and the expected output values. Each row in the table corresponds to a test.

Every model received by the Solver forms the input of a test. The expected output can then be retrieved by calling the appropriate function on the generated input. The test-function loops through all the tests defined in the accompanied table. If the test returns an unexpected output, it will return an error. This kind of testing is known as *table-driven testing* (see Section 2.1.1.

4.6 Design Choices

We have now given a small introduction to each module of AutoTest and explained their purpose. In this section we will point out some of the design choices that has formed AutoTest.

4.6.1 Choice of Constraint Solver

As is mentioned in Section 3.2.2, it can be hard to decide what kind of constraint solver to use, even if you know the applications it is meant to be used for. In this thesis, the constraint solver Z3 is used. While researching and reading papers, Z3 was mentioned a lot of times. Since it is a tool developed at Microsoft Research and used daily by Microsoft themselves, it seemed like this solver would be a good choice. It supports many theories and provides a fully-featured, well-documented C API [44]. Go also provides a package *cgo*, which made it easier to create an API for Go that calls the C API instead of using a lot of time to learn the SMT-LIB textual input format.

4.6.2 Using the Abstract Syntax Tree

As we have seen from Chapter 3, a number of tools mentioned are actually dynamic test generation tools, mixing static and dynamic program analysis techniques. Many of these tools are working directly on compiled bytecode. In this thesis, the main focus has been on symbolic execution, which is a static program analysis technique, and not on dynamic testing.

Implementing a dynamic symbolic execution tool would require a runtime interpreter in addition to implementing symbolic execution on top. Even though dynamic symbolic execution is more powerful than static symbolic execution, it was decided that static symbolic execution on ASTs would be a good place to start, and probably more than enough in order to use the tool to test the assignments given on Autograder.

Go provides an easy way to parse source code into an abstract syntax tree, and the AST contains all the information needed to create tests for a function. In addition, Go's *ast*-package provides a function that will traverse the tree in a depth-first search fashion, which is one of the classic search heuristics for symbolic execution.

4.6.3 Unit Tests

Symbolic execution can be used for full-system testing, and many companies use symbolic execution for testing large applications this way. In this thesis, the focus has been mainly on creating unit tests for each function in an application, with the possibility of expanding to integration and system testing at a later point.

5 | Implementation

In this chapter we will first present a simple program and take a look at its AST. We will use the AST in Section 5.1 to explain how AutoTest has been implemented and some of the challenges encountered. The following sections will explain the most important parts of each module. We limit the implementation to some core features, first of all to gain experience in the field of symbolic execution. In Section 5.7 we will look at some of the problems with the first prototype and what we have done so far to improve AutoTest.

5.1 A Simple Example

The simple program in Listing 5.4 was used as a starting point for implementing the test generator. With the help of the *parser*-package we can parse the source code into an AST. The corresponding AST is shown in a simplified form in Figure 5.1.

All of the information we are interested in can be found in the AST. The next step is to traverse the tree in order to save the information we need in order to create test cases. Go's *ast*-package provides a function that traverses the tree in a depth-first fashion:

```
func Walk(v Visitor, node Node)
```

`Visitor` is an `interface`-type which containts one method:

```go
func ExampleOne(x int) int {
   if x == 0 {
      return 0
   }
   if x < 0 {
      return -1
   } else {
      return 1
   }
}
```

Listing 5.4: A simple example program

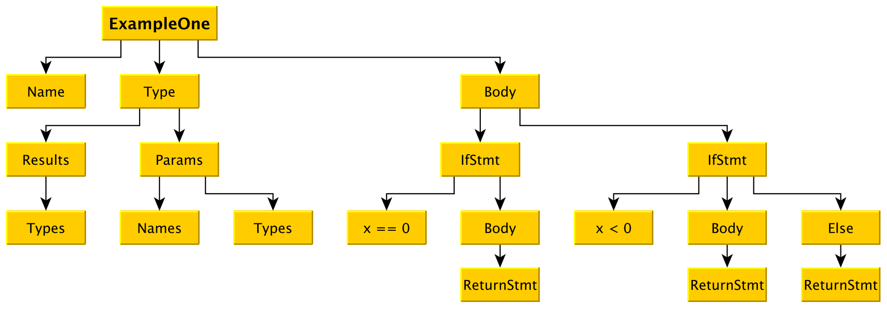


Figure 5.1: Abstract syntax tree of ExampleOne

`Visit(node Node) (w Visitor)`

The function takes a `Node` as input and returns a `Visitor`. `Node` is also an `interface` that all nodes in the AST implements. `Walk` starts by calling `v.Visit(node)` which will return a visitor `w` if node is not `nil`. `Walk` will be called recursively with visitor `w` until a node with value `nil` is hit. In order to use the `Walk`-function, a custom visitor is created that implements the `Visit` method of the `Visitor` interface . Before we move on to the traversal of the AST we will take a look at the parser in the following section.

5.2 Parser

The objective of the Parser-module is to parse source code into an AST. The most important functions in the parser are `ParseFile` and `ParseDir`, which are shown in Listing 5.5. Recall from Section 4.1, that if the first argument is `-f`, then the second argument `path` should point to a file. In this scenario `ParseFile` will be called. `ParseFile` calls a function with the same name from the Go *parser*-package and it takes four arguments:

fset is of the type `token.FileSet` and represents a set of source files.

path is a string of the filepath to the file.

src is an interface where source code can be entered directly. If `src != nil`, `src` will be parsed to an AST. Otherwise an AST will be produced from the `path`.

mode is a flag that can be set in order to control the amount of source code that will be parsed.

If the first argument is `-d`, the second argument should point to a directory, and `ParseDir` is called. If the `path` is indeed a directory, a list of all its subdirectories will be created. The function `allDirsToAST`, shown in Listing 5.6, loops through the list and returns a map that maps filepaths to ASTs. `parser.ParseDir` calls `parser.ParseFile` for all files in the given directory ending in ".go". The arguments in `parser.ParseDir` are the same as in `parser.ParseFile`, except the third argument. The third argument is a filtration function that can be used to exclude files that you do not want to deal with. In our case, we exclude files that contains "_test.go", since files ending in "_test.go" contains test cases and are thus not source code for an assignment.

```go
//ParseFile returns the AST of the given file
func ParseFile(path string) *ast.File {
   fset := token.NewFileSet()
   file, err := parser.ParseFile(fset, path, nil, 0)
   if err != nil {
      log.Fatal("error: argument is not a file!")
   }
   return file
}

//ParseDir calls ParseFile for all .go files in the directory
func ParseDir(path string) map[string]*ast.File {
   info, err := os.Stat(path)
   if err != nil {
      log.Fatal("error: not a valid path!")
   }
   if !info.IsDir() {
      log.Fatal("error: argument is not a directory!")
   }

   dirlist := createListOfDirs(path)
   filesMap := allDirsToAST(dirlist)
   return filesMap
}
```

Listing 5.5: Functions in Parser-module

```go
func allDirsToAST(dirlist []string) map[string]*ast.File {
   fset := token.NewFileSet()
   filesMap := make(map[string]*ast.File)
   filter := func(info os.FileInfo) bool {
      name := info.Name()
      return !info.IsDir() &&
         !strings.HasSuffix(name, "_test.go")
   }

   for _, val := range dirlist {
      pkgs, err := parser.ParseDir(fset, val, filter, 0)
      if err != nil {
         log.Fatalf("error: allDirsToAST: %v", err)
      }

      for _, v := range pkgs {
         for filename, file := range v.Files {
            filesMap[filename] = file
         }
      }
   }
   return filesMap
}
```

Listing 5.6: Parser-module: allDirsToAST

5.3 PathConstraint and Symbolic State

As mentioned earlier, `PathConstraint` is basically a collection of the conditions encountered during traversal of the AST. The solver (see Section 5.5) needs the conditions in order to find an input that corresponds to a satisfiable path. The code for `PathConstraint` is shown in Listing 5.7.

```go
type PathConstraint []Constraint

type Constraint struct {
  Lvl  int
  Cond Condition
}

type Condition struct {
  Left  string
  Op    string
  Right string
}

func (s PathConstraint) Empty() bool {
  return len(s) == 0
}

func (s PathConstraint) Peek() Constraint {
  return s[len(s)-1]
}

func (s *PathConstraint) Push(c Constraint) {
  (*s) = append((*s), c)
}

func (s *PathConstraint) Pop() Constraint {
  c := (*s)[len(*s)-1]
  (*s) = (*s)[:len(*s)-1]
  return c
}
```

Listing 5.7: `PathConstraint`-class and its methods

`PathConstraint` is implemented as a stack of constraints and it contains all the methods that are expected of a stack. `Empty` is a boolean that checks if there are any elements in the stack, `Peek` returns the element at the top of the stack, `Push` inserts the given `Constraint` at the top of the stack, and `Pop` removes and returns the top element from the stack.

A stack seemed like a natural choice to represent a `PathConstraint`, since during AST-traversal we will first move down a certain path and gather conditional statements from that path. When we reach the bottom of the tree and start moving up again, we need to remove conditional statements from the stack, as we pass them on the way up again. This is also why `Lvl` is part of the `Constraint` type. It is just an integer that logs at which level a constraint is added. When this level is passed on the way up the tree, it will be popped off the stack. This is seen in Listing 5.10 in the type switch, under `case nil`.

```go
type State map[string]SymbolicVariable

type SymbolicVariable struct {
  Name       string
  LvlCreated int
  VarType    string
  AssignStmt string
}
```

Listing 5.8: State and SymbolicVariable

Listing 5.8 contains the declarations of `SymbolicVariable` and `State`. The `SymbolicVariable` stores information about the name of a variable, the level in the tree where it is first encountered, the type of the variable and optionally an assignment. `AssignStmt` can be `nil` when a variable is a parameter to a function. Then it will not have any assignment, just the variable name and type. If we later encounter an assignment for this variable, we can update it at that time.

`State` is a map that stores all variables encountered during traversal. We use a map where the name is key, since it is easy to check if a key already exist. If a variable already exists in the map during traversal we only have to update the `AssignStmt`.

5.4 PathFinder and Symbolic Execution

This section presents the implementation of the visitor and the implemented method from the interface that is necessary in order to explore as many execution paths as possible.

```go
type FuncVisitor struct {
   Pkg        string
   FuncName   string
   FuncNumber int
   FilePath   string
   Params     *ast.FieldList //input parameters for the function
   Results    *ast.FieldList //return values
   PC         symbexec.PathConstraint
   State      symbexec.State
   Lvl        int
   TestInputs []string
}
```

Listing 5.9: FuncVisitor

Listing 5.9 shows the custom visitor named `FuncVisitor`. New test cases will be stored in the same package as the source code that the test cases belong to, and thus we have to save the name of the package. The name of the function is stored since the tests produced will be of the form `TestFuncName(t *testing.T)`, where `FuncName` is the name of the function that the test belongs to. Both `Params` and `Results` from Figure 5.1 are of type `*ast.FieldList`. Both values are needed so we can define what types are expected as input and output when we create tests for a function. We will explain this further in Section 5.6 when we discuss the test generator. The `PathConstraint` and `State` were explained in the previous section. The `Lvl` is used to keep track of what level a constraint was added to the `PathConstraint` and at which levels variables are declared.

To implement the `Visitor` interface we need to implement the method in the interface:

```go
func (v *FuncVisitor) Visit(node ast.Node) ast.Visitor {
   switch x := node.(type) {
   case nil:
      v.Lvl--
      if !v.PC.Empty() {
         c := v.PC.Peek()
         if c.Lvl == v.Lvl {
            v.PC.Pop()
         }
      }
   case *ast.IfStmt:
      v.Lvl++
      handleIfStmt(x, v)
      v.Lvl++
      return nil
   case *ast.AssignStmt:
      v.Lvl++
      handleAssignStmt(x, v)
   case *ast.ReturnStmt:
      v.Lvl++
      sat, res := solver.Solve(v.State, v.PC)
      if sat {
         v.TestInputs = append(v.TestInputs, res)
      }
   default:
      v.Lvl++
   }
   return ast.Visitor(v)
}
```

Listing 5.10: FuncVisitor's Visit-function

This function will be used by `Walk` to traverse the tree when we pass `FuncVisitor` as a parameter. The `Node` parameter is an interface, so we need to use a type switch which basically discovers the dynamic type of an interface variable. As you can see from Listing 5.10, we need to make a case for each node in the tree where we want something to happen. We have only included a few of the nodes in this listing to show how the method works. When the node is of type `nil`, then there are no more nodes at the current level and `Walk` will move up one level to the parent node. As we move up the tree, if we pass the same level as where the last condition that was saved to the PC, it is removed from the PC. We also need to include a default case in order for the level to always increment no matter which node it passes, not only the ones that are actually handled.

5.4.1 Handling If-statements

When an if statement is reached, we call a function `handleIfStmt` (see Listing 5.11). The condition of an `*ast.IfStmt` is an interface, so we need a type switch in order to retrieve the type of the condition. Once we have the type, we create a condition for the "then"-branch and one for the "else"-branch. For the `elseCond`, we use a map that finds the inverse of the operation from the "then"-branch. We can then traverse down one branch at a time by calling `Walk` on the next node for each branch, but only after we check if the path constraint along with the state are satisfiable. The traversal will only continue down a certain path if it is satisfiable. After one or both branches has been traversed `handleIfStmt` returns, the level for the visitor is incremented and the `Visit`-function returns `nil`. This stops the "original" traversal to continue down the children of the `IfStmt`, which we have already explored.

The reason why if statements are handled this way is because Go's *ast*-package don't provide us with a way to check a nodes parents or siblings. So if a conditional was handled directly, we would not know whether the condition is the child of an if statement or a loop.

When the traversal reaches a return statement, a call to the solver is made, which will return a model if the path constraint is satisfiable. The model is stored in the visitor in order to create test inputs after the traversal is complete. In the next section, we will take a look at how the solver works.

```go
func handleIfStmt(x *ast.IfStmt, v *FuncVisitor) {
   switch c := x.Cond.(type) {
   case *ast.BinaryExpr:
      left := exprToString(c.X)
      op := c.Op.String()
      right := exprToString(c.Y)
      cond := symbexec.Constraint{
         Lvl:   v.Lvl,
         Left:  left,
         Op:    op,
         Right: right}
      elseCond := symbexec.Constraint{
         Lvl:   v.Lvl,
         Left:  left,
         Op:    invOpMap[cond.Op],
         Right: right}

      v.PC.Push(cond)
      // check if sat before walk
      res := solver.CheckSat(v.State, v.PC)
      if res {
         ast.Walk(v, x.Body)
      } else {
         v.PC.Pop()
      }

      if x.Else != nil {
         fvElse := v
         fvElse.PC.Push(elseCond)
         //check if sat before Walk
         res := solver.CheckSat(v.State, fvElse.PC)
         if res {
            ast.Walk(fvElse, x.Else)
         } else {
            fvElse.PC.Pop()
         }
      }
   default:
      log.Fatalf("Unhandled type %s in handleIfStmt", c)
   }
}
```

Listing 5.11: handleIfStmt

5.5 Solver

This section describes the implementation of the constraint solver, which is called when symbolic execution hits a `ReturnStmt`.

Listing 5.12 shows some of the code for the constraint solver. At the top of the code is the preamble where we include the header files for C's standard library and Z3. `LDFLAGS` tells the linker in which folder it can find the Z3 library and then the name of the library. In the `init()`-function we set up the necessary variables before the solver can be used. The function `Solve` is called when the AST-traversal reaches a `ReturnStmt`. We will study some of the functions here in more detail later, but first we will take a quick look at what happens in the `Solve`-function.

First, we declare all the variables in the `state` through the Z3 API. In the next step we declare all constraints on the stack `pc` through the API. We then call `solverCheck`, which sends a call to the solver to check if the declarations has a satisfiable solution or not. If the `pc` is unsatisfiable, then we will return an empty string. If the check returns `true`, meaning that it is satisfiable, we get the model and return it as a string. The model is used to create inputs for the test cases. The function `solverReset` removes all declarations from the solver and prepares the solver for the next path constraint that needs to be solved.

Listing 5.13 shows the function that declares all the variables in the state into the solver and the function that declare every condition into the solver. For every variable stored in the state, we need to check what type the variable is, and then declare the correct variable type in the solver. As you can see from the listing, there is currently only support for integer values. When declaring constraints we must also check if the types or variables in the conditional statements are already declared in the solver context, and if they are not, we need to declare them. After both sides of a conditional has been checked, a constraint is created in the solver

```go
package solver

/*
#cgo LDFLAGS: -L/usr/local/Cellar/z3/4.4.1/lib -lz3
#include <stdlib.h>
#include <z3.h>
*/
import "C"

import (
   "fmt"
   "log"
   "strconv"
   "unsafe"

   "github.com/bjerga91/test-generator/symbexec"
)

var svr C.Z3_solver
var ctx C.Z3_context

func init() {
   ctx = MkContext()
   svr = mkSolver(ctx)
}

func Solve(state symbexec.State, pc symbexec.PathConstraint) (bool
    , string) {
   var res string
   vars := declVarsInState(state)
   assertConstraints(pc, vars)

   sat := solverCheck(ctx, svr)
   if sat {
      m := solverGetModel(ctx, svr)
      res = modelToString(ctx, m)
   } else {
      res = ""
   }
   solverReset(ctx, svr)
   return sat, res
}
```

Listing 5.12: Constraint Solver, Z3

context and then asserted. It is not before all of the constraints have been asserted that we can check for satisfiability and possibly get a model.

5.6 Test Generator

The test generator module handles the generation of tests, and it is also where the traversal of the AST is initiated. The module contains two main functions: `GenerateTestsForFile` and `GenerateTestsForDir`, which are called after the parser functions `ParseFile` and `ParseDir` respectively (see Section 5.2). As you can see from Listing 5.14, `GenerateTestsForDir` loops through the files in the map, and calls `GenerateTestsForFile` for each file.

In `GenerateTestsForFile`, we call `ast.Inspect`, which traverses the tree in depth-first order. It starts by calling the input function with `file` as input and continues traversal as long as the function returns true. In this case, we only want to find every `*ast.FuncDecl` node that is in the file's AST. For every `FuncDecl`, we create a `FuncVisitor` (see Section 5.4), that is used to store information about each function, and we call `ast.Walk` on the visitor with the node `FuncDecl` as the starting point for the traversal.

After the tree has been traversed, and the necessary information is stored in the visitor, we call `createTests`. `createTests` add test code to the string `text`: if `text` is empty, the code will include the package and import statements at the start of every Go file. If it already contains some text, then it will only include text for the given function. `createTests` and `codeAppendToFile` can be seen in Listing 5.15. After text has been created for all the functions in a file, a call to `writeTestsToFile` is made. This function extracts the file name from the source, say for example ”code.go” and creates a test file based on that name: ”code_test.go”. The file will be replaced if it already exist. Listing 6.25 shows an example of the output from the test-generator.

```go
func declVarsInState(state symbexec.State) map[string]C.Z3_ast {
    vars := make(map[string]C.Z3_ast)
    for _, val := range state {
        if val.VarType == "int" {
            res := declIntVar(ctx, val.Name)
            if val.AssignStmt != "" {
                if i, err := strconv.Atoi(val.AssignStmt); err == nil
                 {
                    data := declInt(ctx, i)
                    res = mkEq(ctx, res, data)
                }
            }
            vars[val.Name] = res
        } else {
            log.Fatal("declVarsInState: Vartype is not int!")
        }
    }
    return vars
}

func assertConstraints(pc symbexec.PathConstraint, vars map[string
    ]C.Z3_ast) {
    var left, right, constraint C.Z3_ast
    for _, c := range pc {
        left = checkTypeAndDeclVar(ctx, vars, c.Left)
        right = checkTypeAndDeclVar(ctx, vars, c.Right)
        if c.Op == "==" {
            constraint = mkEq(ctx, left, right)
        } else if c.Op == "!=" {
            constraint = mkNotEq(ctx, left, right)
        } else if c.Op == "<" {
            constraint = mkLt(ctx, left, right)
        } else if c.Op == "<=" {
            constraint = mkLe(ctx, left, right)
        } else if c.Op == ">" {
            constraint = mkGt(ctx, left, right)
        } else if c.Op == ">=" {
            constraint = mkGe(ctx, left, right)
        }
        solverAssert(ctx, svr, constraint)
    }
}
```

Listing 5.13: Z3: `declVarsInState` and `assertConstraints`

```go
//GenerateTestsForFile generates tests for the given file
func GenerateTestsForFile(file *ast.File, path string) {
   pkg := file.Name.Name
   funcNumber := 0
   text := ""
   ast.Inspect(file, func(n ast.Node) bool {
      switch x := n.(type) {
      case *ast.FuncDecl:
         var stack symbexec.PathConstraint
         visitor := pf.FuncVisitor{
            Pkg:        pkg,
            FuncNumber: funcNumber,
            FilePath:   path,
            Lvl:        0,
            PC:         stack,
            State:      make(symbexec.State),
         }
         ast.Walk(&visitor, x)
         funcNumber++
         text = createTests(visitor, text)
         return false
      }
      return true
   })
   writeTestsToFile(text, path)
}

//GenerateTestsForDir generates tests for the given directory
func GenerateTestsForDir(files map[string]*ast.File) {
   for path, file := range files {
      GenerateTestsForFile(file, path)
   }
}
```

Listing 5.14: Main functions in test generator module

```go
func createTests(f pf.FuncVisitor, text string) string {
   if text == "" {
      text += codeNewFile(f)
   } else {
      text += codeAppendToFile(f)
   }
   return text
}

func codeAppendToFile(f pf.FuncVisitor) string {
   inputString := getInputTypes(f)
   returnString := getReturnTypes(f)
   testcases := getTestCases(f)

   text := fmt.Sprintf(`
   type testpair%s struct {
      input %s
      expected %s
   }

   var tests = []testpair%s{
      %s
   }

   func Test%s(t *testing.T){
      for _, pair := range tests {
         out := %s(pair.input)
         if out != pair.expected {
            t.Error(
               "For", pair.input,
               "expected", pair.expected,
               "got", out,
            )
         }
      }
   }
   `, f.FuncName,
      inputString,
      returnString,
      f.FuncName,
      testcases,
      f.FuncName,
      f.FuncName)

   return text
}
```

Listing 5.15: `createTests` and `codeAppendToFile`

5.7 AutoTest - Version 2.0

After testing the tool and checking the code coverage on the tests generated by the tool (see Section 6), it is clear that the algorithm implemented for finding paths is not good enough. We will take a look back at the example presented in Section 5.1 and explain the problems with the tool. We will then highlight some of the changes made from version 1.0 to version 2.0.

5.7.1 The Flaws

Most of the related work on symbolic execution tools are working directly on the bytecode and not with the AST. In addition, none of them are working in Go. The algorithm for finding paths in ASTs for Go had to be designed from scratch. The first algorithm was designed with ExampleOne in mind. The code for ExampleOne is repeated in Listing 5.16 for the reader's convenience.

```go
func ExampleOne(x int) int {
   if x == 0 {
      return 0
   }
   if x < 0 {
      return -1
   } else {
      return 1
   }
}
```

Listing 5.16: ExampleOne

After implementing and testing the tool, it became clear that the PathFinder had to be improved. Conditions are popped off the stack when the DFS traversal moves upwards and reaches a node that has the same level as the condition at the top of the stack. As an example, the condition for the first if-statement, $x == 0$, will be pushed to the stack when going down the tree, until it hits the return statement. When DFS moves up the tree, the condition will be popped of the stack when it moves past the if statement. When the second if statement is reached, the stack will be empty.

The main problem with this approach is that the PC only saves one path at a time and thus does not support branching at if statements. Pushing and popping conditions for a path is not a good idea either.

```go
func ExampleTwo(x int) int {
    var res int
    if x == 0 {
        res = 0
    }
    if x < 0 {
        res = -1
    } else {
        res = 1
    }
    return res
}
```

Listing 5.17: ExampleTwo

Let's take a look at a similar example, shown in Listing 5.17, where the return statements are replaced by assignments and we only have one return statement at the end. At the return statement, the PC will be empty, when actually we should have found the three satisfiable paths $\{x == 0, x >= 0\}$, $\{x! = 0, x < 0\}$ and $\{x! = 0, x >= 0\}$. We need a new structure for storing paths that allows us to store multiple conditions at the same level.

5.7.2 PathFinder 2.0

Most of the new functionality in version 2.0 is related to the PathFinder and the path exploration algorithm. We have introduced a new structure and made some changes to the `FuncVisitor`. The changes can be seen in Listing 5.18.

The `FuncVisitor` now has a list of `Paths`, and each `Path` has its own `PC` and `State`. The visitor still has a `PC`, which is used during

```go
//Path is a struct that stores the PC and State for each path
    found
type Path struct {
   PC    symbexec.PathConstraint
   State symbexec.State
}

//FuncVisitor is used to store variables when traversing the AST
type FuncVisitor struct {
   Pkg        string
   FuncName   string
   FuncNumber int
   FilePath   string
   Params     *ast.FieldList //input parameters for the function
   Results    *ast.FieldList //return values
   PC         symbexec.PathConstraint
   VarHistory symbexec.VarHistory
   Paths      []Path
   Lvl        int
   TestInputs []string
}
```

Listing 5.18: `Path` and `FuncVisitor` v2.0

traversal. Instead of having a `State` to keep track of variables during traversal, we now use a `VarHistory` instead, which is a stack of symbolic variables. The reason why we use a stack instead of a map to store variables during traversal is because we need to keep a history of the variables in order to backtrack when we move up the tree.

As you can see from Listing 5.19, we have made some changes to the `Visit`-function, compared to how it was in the first version. First of all, recall that `case nil` corresponds to the DFS moving up one level. `v.popVariables` will remove all variables from `v.VarHistory` that has a higher `Lvl` than the current one. `v.popCondition` is performing the same operations that was done in the last version, popping a condition from the `PC` if the condition's level is the same as the current. If we reach level 0, which is basically the root of a funtion, we will solve all the paths that has been added to `v.Paths`.

When we reach a `ReturnStmt` we call `solvePaths()`, shown in Listing 5.20. The difference between `solvePaths` and `solveAllPaths` is that `solvePaths` will only try to find a solution for the paths that contains the condition at the top of the stack `PC`. This is done based on the assumption that all paths that contain the last condition on the stack will also reach the `ReturnStmt`. The paths that are solved will be removed from the list of paths. If the `PC` is empty when we reach a `ReturnStmt` it is probably because the statement is at the top-level, at the end of a function. In this case we will call `solveAllPaths` since the return statement will be reachable for all paths. `ExampleTwo` in Listing 5.17 is an example of such a case.

Listing 5.21 presents the changes made in the function `handleIfStmt`. The code is mostly the same as in the first version (Listing 5.11) with one important difference: the function call `addConditionsToPaths`. If the list `Paths` is empty, we create two new paths; one for the then-branch and one for the else-branch.

If we already have paths in our list we will loop through the

```go
func (v *FuncVisitor) Visit(node ast.Node) ast.Visitor {
   switch x := node.(type) {
   case nil:
      v.Lvl--
      fmt.Println("Going up. Lvl:", v.Lvl)

      //Removes variables from the variable history that has Lvl
          higher than current
      v.popVariables()

      //Removing last condition from PC if current lvl is same as
          the lvl the condition was saved to PC
      v.popCondition()

      if v.Lvl == 0 {
         v.solveAllPaths()
      }
   case *ast.IfStmt:
      v.Lvl++
      handleIfStmt(x, v)
      v.Lvl--
      return nil
   case *ast.AssignStmt:
      v.Lvl++
      handleAssignStmt(x, v)
   case *ast.ReturnStmt:
      v.Lvl++
      v.solvePaths()
   default:
      v.Lvl++
   }
   return ast.Visitor(v)
}
```

Listing 5.19: FuncVisitor's Visit-function v2.0

```go
func (v *FuncVisitor) solvePaths() {
    if v.PC.Empty() {
        v.solveAllPaths()
        return
    }
    var newPaths []Path
    for i := range v.Paths {
        if v.Paths[i].PC.Contains(v.PC.Peek()) {
            sat, res := solver.Solve(v.Paths[i].State, v.Paths[i].PC)
            if sat {
                fmt.Print("Path: ", v.Paths[i].PC.ToString(), "\t|\
                    tmodel:", res)
                v.TestInputs = append(v.TestInputs, res)
            }
        } else {
            newPaths = append(newPaths, v.Paths[i])
        }
    }
    v.Paths = newPaths
}
```

Listing 5.20: `solvePaths`

paths, and for each path we will create a `thenpath` and an `elsepath`. If `PC` is empty, we will replace the original path with the two new paths, given that they are satisfiable. If there are conditions in the stack, we will first check if the original path contains the condition at the top of the stack. If it does, the original path will be removed from the list and the `elsepath` and `thenpath` are added, given that they are satisfiable. If the original path does not contain the last condition on the stack, we will leave the path in the list `PC` unchanged. The code for the function `addConditionsToPaths` is displayed in Listing 5.22.

```go
func handleIfStmt(x *ast.IfStmt, v *FuncVisitor) {
   switch c := x.Cond.(type) {
   case *ast.BinaryExpr:
      left := exprToString(c.X)
      op := c.Op.String()
      right := exprToString(c.Y)
      cond := symbexec.Constraint{
         Lvl:   v.Lvl,
         Left:  left,
         Op:    op,
         Right: right}
      elseCond := symbexec.Constraint{
         Lvl:   v.Lvl,
         Left:  left,
         Op:    invOpMap[cond.Op],
         Right: right}

      //each path branches into PC(then) and PC'(else)
      v.addConditionsToPaths(cond, elseCond)

      v.PC.Push(cond)
      //check if sat before walk
      res := solver.CheckSatVarHist(v.VarHistory, v.PC)
      if res {
         ast.Walk(v, x.Body)
      } else {
         v.PC.Pop()
      }

      if x.Else != nil {
         v.PC.Push(elseCond)
         //check if sat before walk
         res := solver.CheckSatVarHist(v.VarHistory, v.PC)
         if res {
            ast.Walk(v, x.Else)
         } else {
            v.PC.Pop()
         }
      }
   //TODO condition could be other than BinaryExpr.
   default:
      log.Fatalf("Unhandled type %s in handleIfStmt", c)
   }
}
```

Listing 5.21: `handleIfStmt`

```go
func (v *FuncVisitor) addConditionsToPaths(cond symbexec.Constraint,
   elseCond symbexec.Constraint) {
   if len(v.Paths) > 0 {
      var newPaths []Path
      for _, path := range v.Paths {
         elsepath := append(path.PC, elseCond)
         elsesat := solver.CheckSat(path.State, elsepath)
         thenpath := append(path.PC, cond)
         thensat := solver.CheckSat(path.State, thenpath)
         state := path.State

         //If PC is not empty, we will only create new paths for paths that
         //contain the last Condition. If PC is empty, we will create new paths for all
         if !v.PC.Empty() {
            if path.PC.Contains(v.PC.Peek()) {
               if elsesat {
                  newPaths = append(newPaths, Path{
                     PC:    elsepath,
                     State: state})
               }
               if thensat {
                  newPaths = append(newPaths, Path{
                     PC:    thenpath,
                     State: state})
               }
            } else {
               newPaths = append(newPaths, path)
            }
         } else {
            if elsesat {
               newPaths = append(newPaths, Path{
                  PC:    elsepath,
                  State: state})
            }
            if thensat {
               newPaths = append(newPaths, Path{
                  PC:    thenpath,
                  State: state})
            }
         }
      }
      v.Paths = newPaths
      //If there are no paths in list already, we create a new path for each branch.
   } else {
      v.Paths = append(v.Paths, Path{PC: symbexec.PathConstraint{
          cond}})
        v.Paths = append(v.Paths, Path{PC: symbexec.PathConstraint{
          elseCond}})
    }
}
```

Listing 5.22: `addConditionsToPaths`

5.7.3 Test Generator 2.0

Version 2.0 also includes a small but important change to the test generator-module. It includes a function that retrieves the output of a function when it is executed on the generated input. This output is what forms the ”expected output” for the tests. The code for this function is shown in listing 5.23.

The function creates a temporary directory with a temporary go-file which calls the function with the generated input. We create a command to execute this file with `exec.Command()`. When we call `Output()`, the command runs and returns the standard output and error. After we have retrieved the output, we delete the temporary file and directory and then returns the output.

In addition, improvements has been made for the function `codeAppendToFile`, (version 1.0 is shown in Listing 5.15). The last version of the test generator only had support for one input and one output value, as you can see from the definition of the testpair-structure. The new version is made more general, to be able to support a number of input- and output variables.

```
func getExpectedOut(f pf.FuncVisitor, input string) string {
   end := strings.LastIndex(f.FilePath, "/")
   pkgString := f.FilePath[:end]
   text := fmt.Sprintf(`
      package main

      import (
         "fmt"
         "%s"
      )

      func main(){
         fmt.Print(%s.%s(%s))
      }
   `, pkgString,
      f.Pkg,
      f.FuncName,
      input,
   )

   createDir("../temp")
   path := fmt.Sprintf("../temp/temp.go")
   createFile(path, text)
   out, err := exec.Command("go", "run", path).Output()
   if err != nil {
      log.Fatal("getExpectedOut: Failed executing command!", err)
   }
   delDir("../temp")
   return string(out[:])
}
```

Listing 5.23: `getExpectedOutput`

6 | Results

In this chapter we will discuss the results achieved by generating tests with AutoTest. The tool is far from complete, and it has some limitations regarding the type of source code that it can handle. We will present the basic test scenarios used and show the results from testing the code coverage on the tests produced from AutoTest.

6.1 Version 1.0

We first show the output from running AutoTest on the file containing `ExampleOne` (see Listing 5.4, on Page 37) in Listing 6.24. We will then look at the output from running AutoTest on `ExampleTwo` (see Listing 5.17 on Page 54) where we can clearly see why version 1.0 does not have a good path exploration algorithm.

```
andreasbjerga$ ./AutoTestv1 -f ../seExamples/seExamples.go
Parse file
Traversing tree of function: ExampleOne
Path: x==0  |  model: x -> 0
Path: x<0   |  model: x -> (- 1)
Path: x>=0  |  model: x -> 0
Writing tests to file...
```

Listing 6.24: Output from AutoTest 1.0

6.1.1 ExampleOne

Listing 6.25 shows what a typical test file created from AutoTest looks like. In this case, it shows the test generated from `ExampleOne`. The source code is not this nicely formatted after it has been generated, but passing the generated code through Go's formatter, *gofmt*, fixes the formatting.

For every function that we test, we create a test pair structure that contains the input type(s) of the function, and the type(s) that the function returns. We will then create a table of the test pair with input and expected output values. This table basically contains all the tests for a function, and the function `TestExampleOne` is kind of like a test script that runs through all the tests specified in the table. The test script will match the `expected` output against what the function actually returns (`out`) for the provided `input`. If they are not the same, the current test case will fail. This type of testing is called table-driven testing.

You might have noticed that the expected output is defined as `nil` for each test case. Symbolic execution only generates inputs for the paths found during AST traversal, it does not find the expected output. Given that the tests are generated from a solution that the teacher has made, and since we assume that the solution is correct, we can easily find the expected output by simply calling the function with the generated inputs. Whatever value it returns would then be "expected output" when the tests are executed on students source code on Autograder. Version 2.0 of AutoTest included a workaround that finds this automatically (see Section 5.7.3). As of this version, the expected output has to be entered manually.

Finding the expected output for each case in `ExampleOne` is an easy task. After entering the values into the test pair slice, the tests will look like this:

It is quite clear from Listing 6.26 that we have a duplicate test. This is related to the issue discussed in Section 5.7.1. After hitting

```
type testpairExampleOne struct {
   input    int
   expected int
}

var tests = []testpairExampleOne{
   {0, nil},
   {-1, nil},
   {0, nil},
}

func TestExampleOne(t *testing.T) {
   for _, pair := range tests {
      out := ExampleOne(pair.input)
      if out != pair.expected {
         t.Error(
            "For", pair.input,
            "expected", pair.expected,
            "got", out,
         )
      }
   }
}
```

Listing 6.25: TestExampleOne

```
    var  tests = [] testpairExampleOne{
        {0, 0},
        {-1, -1},
        {0, 0},
    }
```

Listing 6.26: Tests for ExampleOne

the last if-statement in `ExampleOne` (see Listing 5.4), satisfiable paths should be $\{x = 0\}$, $\{x \neq 0, x < 0\}$ and $\{x \neq 0, x \geq 0\}$. As we can see from Listing 6.24, the paths found are $\{x = 0\}$, $\{x < 0\}$ and $\{x \geq 0\}$. The solver returns the model $\{x \rightarrow 0\}$ for both the first and last path, since it is a satisfiable model for both these paths. The model generated for the last path will never fire the last path, since it satisfies the if-statement with the condition $\{x = 0\}$. If the last path constraint also contained the condition $\{x \neq 0\}$, the solver would have returned a model that could only lead to the third path, $\{x \rightarrow 1\}$. Version 2.0 of our AutoTest tool has addressed this problem.

6.1.2 ExampleTwo

Listing 6.27 presents the tests produced when running AutoTest on `ExampleTwo` (see Listing 5.17 on 54). Since the test structure and the test-script function is pretty much the same in every case, we have only included the actual test cases. As you can see, version 1.0 of AutoTest did not generate a single test for this source code. This is not very surprising given the flaws we discussed in Section 5.7.1. In short, when the return statement is encountered during traversal the path constraint will be empty since the return statement is outside any if-sentences and since we pop conditionals of the stack when we move up the tree.

```
var testsExTwo = []testpairExampleTwo{}
```

Listing 6.27: TestExampleTwo

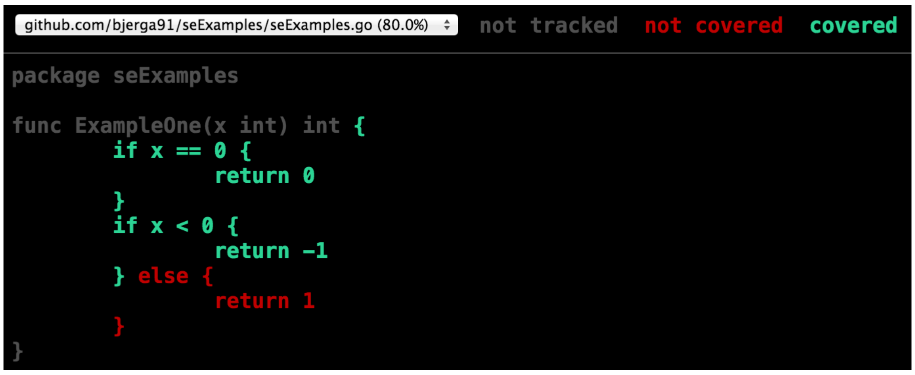
6.1.3 Code Coverage

After the tests has been created and expected output entered for each test, we can run the test and check the code coverage. The easiest way to do this is to move into the directory for which you have created tests and then run the command `go test -coverprofile cover.out`. This command will run all the tests it finds inside the current directory and then compute the test coverage; the amount of the package's source code that has been executed by running the tests. The output from running the command on `ExampleOne` is shown in Listing 6.28.

```
andreasbjerga$ go test -coverprofile cover.out
PASS
coverage: 80.0% of statements
ok      github.com/bjerga91/seExamples    0.037s
```

Listing 6.28: Code coverage on `ExampleOne`

Not surprisingly, the tests does not cover all of the code. The command mentioned above also creates a coverage profile for us, a file with detailed information about the code coverage. Go provides a nice HTML representation of the coverage information. If we run the command `go tool cover -html=cover.out`, the representation shown in Figure 6.1 opens up in the browser. This view provides an easy way to check what parts of the code that are actually covered and which are not.



```
package seExamples

func ExampleOne(x int) int {
	if x == 0 {
		return 0
	}
	if x < 0 {
		return -1
	} else {
		return 1
	}
}
```

Figure 6.1: AutoTest 1.0: HTML Coverage Profile

We also include the results from the code coverage on `ExampleTwo` in Listing 6.29. Since AutoTest 1.0 was unable to create a single test case for us, the results are as bad as expected.

```
andreasbjerga$ go test -coverprofile cover.out
PASS
coverage: 0.0% of statements
ok      github.com/bjerga91/seExamples   0.011s
```

Listing 6.29: Code coverage on `ExampleTwo`

6.2 Version 2.0

As mentioned in Section 5.7.2 and 5.7.3, version 2.0 includes an improved path exploration algorithm in addition to automatically finding the expected output. We will in this section take a look at the results of running AutoTest version 2.0 on `ExampleOne` and `ExampleTwo`, and then compare the code coverage achieved.

```
andreasbjerga$ ./AutoTest -f ../seExamples/seExamples.go
Parsing file...

Traversing tree of function: ExampleOne
Path: x==0      |  model: x -> 0
Path: x!=0, x<0       |  model: x -> (- 1)
Path: x!=0, x>=0      |  model: x -> 1

Traversing tree of function: ExampleTwo
Path: x!=0, x>=0    |    model: x -> 1
Path: x!=0, x<0    |    model: x -> (- 1)
Path: x==0, x>=0    |    model: x -> 0
Writing tests to file...
```

Listing 6.30: Output from AutoTest 2.0

The output from running the tests on the file `seExamples.go`, which contains both of the functions `ExampleOne` and `ExampleTwo`, is shown in Listing 6.30. If we compare the output from version 2.0 with the output from version 1.0 in Listing 6.24 it is easy to see that the path exploration algorithm has been improved. In version 1.0, each path in `ExampleOne` was only constrained by one condition. In version two, the two last paths have two conditions in the `PC`, which is as expected if we look at the source code (Listing 5.4 on

Page 37).

The tests produced for `ExampleOne` and `ExampleTwo` by AutoTest are displayed in Listing 6.31 and 6.32, respectively. We can see that AutoTest now produce three tests with unique inputs for each test. In contrast, version 1.0 produced two unique inputs for `ExampleOne` and zero inputs for `Exampletwo`. The important question now is: has the code coverage improved?

```
var testsExampleOne = []testpairExampleOne{
    {0, 0},
    {-1, -1},
    {1, 1},
}
```

Listing 6.31: Tests produced for `ExampleOne`

```
var testsExampleTwo = []testpairExampleTwo{
    {1, 1},
    {-1, -1},
    {0, 1},
}
```

Listing 6.32: Tests produced for `ExampleTwo`

6.2.1 Code Coverage

The tests for `ExampleOne` and `ExampleTwo` is now stored together in one file. We can then check the combined code coverage by running the same command as before: `go test -coverprofile cover.out`. The result is shown in Listing 6.33. With the new path exploration algorithm we are able to get 100% code coverage on `ExampleOne` and `ExampleTwo`!

```
andreasbjerga$ go test -coverprofile cover.out
PASS
coverage: 100.0% of statements
ok      github.com/bjerga91/seExamples    0.012s
```

Listing 6.33: Code coverage version 2.0

7 | Discussion

AutoTest in its current state is only a very simple prototype. There is a lot of work that remains before the tool can be used along with Autograder. A key objective in this thesis has been to develop an algorithm that would find as many execution paths as possible based on an AST of an application. This proved to be a difficult task, and a lot of ideas that seemed good at first, turned out to not work as perfectly as first anticipated after closer consideration. However, considering the results from the last chapter, it is fair to say that the path exploration algorithm in version 2.0 looks promising. Nevertheless, we need to test AutoTest on more source code before we can be sure that it performs well.

In the rest of this chapter we will discuss some of the limitations of AutoTest in its current version and also talk a bit about the challenges encountered throughout the work on this thesis. In the next chapter we will talk about the work that needs to be done to further improve the tool.

7.1 Limitations

As mentioned in the introduction, one of the subgoals of the AutoTest-project was to be able to automatically generate test cases for a simple application. This subgoal has now been accomplished. As we have mainly focused on this subgoal in this thesis, we have tried to limit the functionality to what was necessary in order to be able to generate tests for the example. In this section we will highlight some of the features that are currently supported and what are not.

7.1.1 Types and Conditionals

AutoTest in its current state only supports integers. Before other types can be supported, the data structure for storing conditionals has to be improved. When designing AutoTest, we made the assumption that a conditional statement was of the form x *op* y, where *op* is an operator like $\{=, \neq, <, \leq, >, \geq\}$. While this is true for the examples used in this thesis, this is not always the case. A conditional statement can also be for example a single boolean variable x.

In addition, we have only looked at if statements thus far. There are other conditional types that also must be handled in the function `Visit`, like for example switch-case statements and later for-loops. The handling of if statements must also be improved. We have only considered if statements with a single condition so far, when in fact, a typical condition in an if statement can be of the form x && y || z. There is also not support for `else if` in the if statements for the time being, only `if-else`.

7.1.2 Variables and Assignments

AutoTest's handling of variables and assignments in it's current state is also very weak. We have so far only considered assignments with a single variable or constant on each side of the assignment operator. In reality, an assignment can often be of the form $x = x + y$. Currently, we have used a string to store assignments of a variable. In order to handle algebraic operations in assignments we must find a better data structure to store assignments of a variable.

In addition, variables should be updated if they already exist in the symbolic state and not overwritten. Let's say that we hit an assignment statement of the form $x = x + y$ during traversal. We must then check if x and y already exist in the state, and replace each variable in the statement with their value from the state before we update the symbolic state with the new assignment for x. This functionality has not implemented yet.

7.1.3 Solver

The Z3 C API is quite large, with more than 200 functions. In this thesis we have created a smaller API for Go that contains the most relevant functions. It contains functions for declaring types, functions for initializing the solver itself, and functions related to the theories of algebraic numbers, propositional logic, equality and arithmetic. A fully-featured tool would also require to implement functions for the theories of bit-vectors and arrays.

7.2 Challenges

In this section we will explain some of the challenges encountered while designing and implementing the tool.

7.2.1 Working with the AST

Go's ast-package provides no way of checking the parent or siblings of a node in an AST. It is only possible to access the child of a node through the structure fields. This made it difficult to figure out how to use symbolic execution on ASTs and especially difficult to design the algorithm for finding paths. In addition, a lot of the functions and methods that the *ast*-package provides are working with interfaces. This requires a lot of type switching, in order to discover the dynamic type of an interface variable. Since an interface can represent any node, it can be difficult to predict what dynamic types that a function may return if the return type is an interface. This in turn makes it hard to know if all interesting nodes has been handled.

Take for instance an `IfStmt`, shown in Listing 7.34, as it is declared in the *ast*-package. The field `Cond` is of the type `Expr`, which is an interface. When we save a condition to the path constraint we need a type-switch to convert the condition to its dynamic type. In the case of an expression like $x >= 0$ the dynamic type will be an `*ast.BinaryExpr`. If the condition is a boolean variable x the type will be an `*ast.Ident`-node, and if the condition is $\neg x$ the type will be an `*ast.UnaryExpr`. We have currently only treated `*ast.BinaryExpr` for conditionals in this thesis, but in order to create a tool that will support any kind of code, each node most be handled individually.

```
type IfStmt struct {
        If   token.Pos // position of "if" keyword
        Init Stmt      // initialization statement; or nil
        Cond Expr      // condition
        Body *BlockStmt
        Else Stmt // else branch; or nil
}
```

Listing 7.34: `IfStmt` struct

8 Further Work

Even though the results from AutoTest version 2.0 looks promising, there is a lot of remaining work that needs to be done before it can be used in combination with Autograder. This chapter presents suggestions for further work that can be done to further improve AutoTest and make it work with a broader range of applications.

8.1 Address Limitations

We discussed some limitations in the last chapter (see Section 7.1). The tool can be improved a lot by addressing the limitations discussed here. We will repeat some of them here for the reader's convenience. To further improve the tool there must be implemented support for multiple conditions in if statements, other conditional structures than if sentences (switch-case, for-loops) and for allowing `else-if` in if statements.

Finding the negated condition in an `if-else` statement is quite easy, but finding the negated condition in an `if-elseif-else` statement is considerably harder. We need to get the condition that triggers the else-statement in order to add it to a path constraint. In the case of an `if-elseif-else` statement, the conditions from all `if-elseif` statements must be collected, and then we would need to find the condition for the else-statement that is the inverse of the collection; the condition that is not covered by any of the other conditions.

8.2 New Data Structures

As also mentioned in the section about limitations (Section 7.1), we need new data structures for storing variables and conditions. One approach could be to change the type of variables and conditions to interface, which we can use to store the nodes from the *ast*-package we encounter during traversal. This way, we do not need to create a type-switch to find for example the type of a condition, but can instead save every condition encountered as an interface. This approach however, requires type-switching before variables and conditions can be declared in the solver.

Say we got a list of conditions, where one of the conditions is of the form `*ast.BinaryExpr`, then we would have to check what type the left operand, X, is and declare it in the solver if it is not already declared. Then we would do the same for the right operand, Y. Lastly, we would check what the operator Op is and handle it accordingly.

Let's look at another example with variables. Say you got the two assignments $x = x + y$ and $y = x - y$. First we store x in the state with key x and value $x + y$, which is a `*ast.BinaryExpr`. In the second assignment, $y = x - y$, we can check if any of the variables on the right side of the assignment are declared in the state. Since x exist in the state we can replace x in the binary expression with the value that is stored in the state for the key x, and then store y in the state with the new expression $(x + y) - y$. Storing variables this way makes it easier to update variables that already exists in the state, and it can make it easier to declare the variables in the solver.

8.3 Control Flow Graph

Currently, a `Path`-structure is used to represent an execution path. Each `Path` has a path constraint and a state (map of symbolic variables). We then use a list of `Path`-objects to represent all the execution paths of a function. The problem with this is that there will be many duplicate conditions and variables across all the paths.

A Control Flow Graph (CFG) will probably be a better data structure for representing all execution paths of a function. The root of the graph is the entry point of the function and nodes in our graph will be either assignments or conditional statements. Every if statement would create a branch-point that would result in a then-branch and a else-branch.

After a function has been traversed, we would have a complete CFG. An execution path in the CFG is then represented by a leaf node and the path leading to it.

8.4 Constraint Solver

Currently, when we have checked a path for satisfiability, we reset the solver Z3, meaning that all variables and conditions declared are removed from the solver. It is possible to create a backtracking point with the function `solverPush`. Instead of resetting the entire solver, one can instead backtrack a number of backtracking points by using the function `solverPop`. So instead of redeclaring a lot of the conditions and variables for the next path, it is possible to backtrack and then continue with the conditions and variables that has not been declared. This is not easy to do with the current data structure that is used to represent paths, but it could be a good optimization for the solver after for example a CFG (see last section) has been implemented.

Another possible optimization related to the solver is to run multiple constraint solvers in parallel through metaSMT (see Section

3.2.3). With running multiple solvers in parallel, AutoTest could be able to create tests for a large application faster, than by only using Z3. However, AutoTest is probably not going to be executed on large enough applications that it will matter considerably.

8.5 Beyond Unit Testing

One of static symbolic execution's biggest drawbacks is that constraint solvers cannot reason about external function calls and library functions, where the code is not accessible. What happens if we perfrom symbolic execution on one function and the function contains a call to another function, where the other function is part of the file parsed by AutoTest? In other words, the code is accessible, but it is outside of the AST that is currently being traversed. At the time being there are no functionality implemented for such a case, but it is possible to continue symbolic execution down the AST of the function that was called and then return to the original function and continue traversal. Since a constraint solver cannot understand the function call, this feature will improve the code coverage in such cases. This feature also makes it possible to extend testing from not only considering unit testing, but also integration and system testing.

8.6 Dynamic Symbolic Execution

Even though static symbolic execution is expected to achieve a satisfying code coverage on the assignments that will be given on Autograder, it is also possible to further improve the code coverage by combining the static test generation with dynamic symbolic execution. This is no small task however, as it requires a runtime interpreter that analyzes the bytecode while the application subject to test generation is executed.

9 | Conclusion

This thesis presents a simple prototype of AutoTest; a static test generation tool for Go. The prototype make use of modern test generation techniques and utilizes a constraint solver to generate inputs for a function.

Two versions of AutoTest has been presented, where the main difference is the algorithm for finding exeution paths. The results show that version 2.0 definitely is an improvement over version 1.0 in terms of finding execution paths. The techniques presented works, and the tool could prove to be very useful. AutoTest needs to be tested more thoroughly on a larger collection of source code. However, more functionality must be added first, to eliminate some of the limitations of the tool in it's current state.

The topic of constraint solving is an advanced and enormous field. We've only touched the surface of the subject in this thesis. A deeper understanding of constraint solving, and SMT theories in particular, is probably necessary in order to design a fully-featured tool.

References

[1] Lu Luo and Carnegie Mellon Universityˆ dInstitute for Software Research International. “Software Testing Techniques-Technology Maturation and Research Strategies”. In: *Institute for Software Research International-Carnegie Mellon University, Pittsburgh, Technical Report* (2010).

[2] Aditya P. Mathur. *Foundations of Software Testing: Fundamental Algorithms and Techniques.* Pearson India, 2007.

[3] Software Testing Help Team. *Static Testing and Dynamic Testing - Difference Between These Two Important Testing Techniques.* 2015. URL: `http://www.softwaretestinghelp.com/static-testing-and-dynamic-testing-difference/`.

[4] Roger S. Pressman. *Software Engineering: A Practitioner’s Approach.* Seventh Edition. McGraw-Hill, 2007.

[5] Michael Kelly. *Choosing a test automation framework.* 2003. URL: `http://www.ibm.com/developerworks/rational/library/591.html`.

[6] Gayatri Ghanakota. *Testing Frameworks.* URL: `https://www.cs.colorado.edu/~kena/classes/5828/s12/presentation-materials/ghanakotagayatri.pdf`.

[7] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann, 2010.

[8] Cristian Cadar et al. “Symbolic Execution for Software Testing in Practice: Preliminary Assessment”. In: *Proceedings of the 33rd International Conference on Software Engineering*. ACM. 2011, pp. 1066–1071.

[9] Cristian Cadar and Koushik Sen. “Symbolic Execution for Software Testing: Three Decades Later”. In: *Communications of the ACM* 56.2 (2013), pp. 82–90.

[10] Corina S Păsăreanu and Willem Visser. “A survey of new trends in symbolic execution for software testing and analysis”. In: *International journal on software tools for technology transfer* 11.4 (2009), pp. 339–353.

[11] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. “Automated Whitebox Fuzz Testing”. In: *NDSS*. Vol. 8. 2008, pp. 151–166.

[12] Patrice Godefroid, Nils Klarlund, and Koushik Sen. “DART: Directed Automated Random Testing”. In: *ACM Sigplan Notices*. Vol. 40. 6. ACM. 2005, pp. 213–223.

[13] Koushik Sen, Darko Marinov, and Gul Agha. “CUTE: A Concolic Unit Testing Engine for C”. In: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2005, pp. 263–272.

[14] Patrice Godefroid. “Test Generation Using Symbolic Execution”. In: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2012)*. Vol. 18. Leibniz International Proceedings in Informatics (LIPIcs). 2012, pp. 24–33.

[15] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer-Verlag Berlin Heidelberg, 2008.

[16] Leonardo de Moura and Nikolaj Bjørner. “Satisfiability Modulo Theories: Introduction and Applications”. In: *Communications of the ACM* 54 (2011), pp. 69–77.

[17] *SMT-COMP*. URL: `http://smtcomp.sourceforge.net/2016/` (visited on 05/30/2016).

[18] *The International SAT Competitions Web Page*. URL: `http://www.satcompetition.org/` (visited on 05/30/2016).

[19] *SMT-LIB*. URL: `http://smtlib.cs.uiowa.edu/` (visited on 06/01/2016).

[20] Leonardo de Moura and Nikolaj Bjørner. “Satisfiability Modulo Theories: An Appetizer”. In: *Formal Methods: Foundations and Applications* (2009), pp. 23–36.

[21] Leonardo de Moura. *SMT Solvers: Theory and Implementation*. URL: `https://leodemoura.github.io/files/oregon08.pdf` (visited on 06/03/2016).

[22] Mark Summerfield. *Programming in Go: Creating Applications for the 21st Century*. Addison-Wesley, 2012.

[23] Google Inc. Rob Pike. *Go at Google: Language Design in the Service of Software Engineering*. 2012. URL: `https://talks.golang.org/2012/splash.article`.

[24] The Go Authors. *Effective Go*. 2016. URL: `https://golang.org/doc/effective_go.html`.

[25] Stephen D Brookes, Charles AR Hoare, and Andrew W Roscoe. “A Theory of Communicating Sequential Processes”. In: *Journal of the ACM (JACM)* 31.3 (1984), pp. 560–599.

[26] Joel Jones. “Abstract Syntax Tree Implementation Idioms”. In: *Proceedings of the 10th conference on pattern languages of programs (plop2003)*. 2003, pp. 1–10.

[27] The Go Authors. *Command cgo*. URL: `https://golang.org/cmd/cgo/` (visited on 05/01/2016).

[28] The Go Authors. *C? Go? Cgo.* Mar. 17, 2011. URL: `https://blog.golang.org/c-go-cgo` (visited on 05/01/2016).

[29] Heine Furubotten. "The Autograder Project: Improving software engineering skills through automated feedback on programming exercises". MA thesis. University of Stavanger, June 2015.

[30] *GitHub.* URL: `https://github.com/` (visited on 06/11/2016).

[31] *Docker.* URL: `https://www.docker.com/` (visited on 06/11/2016).

[32] The KLEE Team. *KLEE LLVM Execution Engine.* URL: `http://klee.github.io/` (visited on 06/10/2016).

[33] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *OSDI.* Vol. 8. 2008, pp. 209–224.

[34] Vijay Ganesh and Trevor Hansen. *STP constraint solver: Simple Theroem Prover SMT solver.* URL: `http://stp.github.io/` (visited on 06/10/2016).

[35] Chris Lattner. *The LLVM Compiler Infrastructure.* URL: `http://llvm.org/Features.html` (visited on 05/04/2016).

[36] Patrice Godefroid, Michael Y Levin, and David Molnar. "SAGE: Whitebox Fuzzing for Security Testing". In: *Queue* 10.1 (2012), p. 20.

[37] Koushik Sen and Gul Agha. "CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools". In: *Computer Aided Verification.* Springer. 2006, pp. 419–423.

[38] Corina S Păsăreanu et al. "Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software". In: *Proceedings of the 2008 international symposium on Software testing and analysis.* ACM. 2008, pp. 15–26.

[39] *NASA Java PathFinder.* URL: `http://babelfish.arc.nasa.gov/trac/jpf/` (visited on 05/05/2016).

[40] Karthick Jayaraman et al. "jFuzz: A Concolic Whitebox Fuzzer for Java." In: *NASA Formal Methods.* 2009, pp. 121–125.

[41] *SMT Solvers.* URL: `http://smtlib.cs.uiowa.edu/solvers.shtml` (visited on 06/04/2016).

[42] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 2008, pp. 337–340.

[43] *Z3 Wiki.* URL: `https://github.com/Z3Prover/z3/wiki` (visited on 06/10/2016).

[44] *Z3: An Efficient Theorem Prover.* URL: `http://z3prover.github.io/api/html/index.html` (visited on 05/05/2016).

[45] Vijay Ganesh and David L Dill. "A Decision Procedure for Bit-Vectors and Arrays". In: *Computer Aided Verification.* Springer. 2007, pp. 519–531.

[46] *The MiniSat Page.* URL: `http://minisat.se/` (visited on 06/04/2016).

[47] Robert Brummayer and Armin Biere. "Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays". In: *Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 2009, pp. 174–177.

[48] Aina Niemetz, Mathias Preiner, and Armin Biere. "Boolector 2.0". In: *JSAT* 9 (2015), pp. 53–58. URL: `https://satassociation.org/jsat/index.php/jsat/article/view/120`.

[49] Bruno Dutertre. "Yices 2.2". In: *Computer Aided Verification.* Springer. 2014, pp. 737–744.

[50] *Yices Wiki: SMT Competitions.* URL: `http://yices-wiki.csl.sri.com/index.php/Main_Page#SMT_Competitions` (visited on 06/04/2016).

[51] Finn Haedicke et al. "metaSMT: Focus on Your Application not on Solver Integration." In: *DIFTS@ FMCAD.* 2011.

[52] Heinz Riener et al. "MetaSMT: a unified interface to SMT-LIB2". In: *Specification and Design Languages (FDL), 2014 Forum on.* Vol. 978. IEEE. 2014, pp. 1–6.

[53] *SWORD - A Module-based SMT Solver.* URL: `http://www.informatik.uni-bremen.de/agra/eng/sword.php` (visited on 06/10/2016).

[54] *PicoSAT.* URL: `http://fmv.jku.at/picosat/` (visited on 06/10/2016).

[55] *AIGER.* URL: `http://fmv.jku.at/aiger/` (visited on 06/10/2016).

[56] Hristina Palikareva and Cristian Cadar. "Multi-solver support in symbolic execution". In: *Computer Aided Verification.* Springer. 2013, pp. 53–68.

List of Figures

List of Listings

A | Attachments

A.1 Source Code

Source code of AutoTest version 2.0:

A.2 Test Examples

Source code for the examples used for test generation: