Static Testing:
Using the Weakest Pre-condition Calculus

Elmar Keij
ekeij@cs.uu.nl

Master Thesis
INF/SCR-07-37

Daily Supervisor: dr. S.W.B. Prasetya
Supervisor: prof. dr. S.D. Swierstra

Center for Software Technology,
Institute of Information and Computing Sciences,
Utrecht University,
Utrecht, The Netherlands

May 31, 2009
Abstract

An important aspect of the software process is the validation and verification of software. In practice the most widely used verification method is software testing. However, proper software testing is expensive and labour intensive. One of the reasons for this is that it is often necessary to mimic the runtime behaviour of (parts of) the software. In this thesis an alternative method of testing, called static testing, is proposed. This method prevents the execution of software by incorporating techniques from the field of static verification, such as the weakest pre-condition calculus. Because the software is no longer executed, there is no need to mimic its runtime behaviour. This thesis explores the basic principles and design of this approach. While static testing has potential, it is still far from a practical application.
Before you lies the thesis of Elmar Keij, submitted in partial fulfillment of the requirements for the degree of Master of Science in Software Technology. This thesis presents the work I have done during my thesis project, which focused on the construction of a prototype static testing tool.

I would like to take this opportunity to thank some people who have helped and supported me during the time I worked on my thesis project. Firstly and foremost I would like to thank my daily supervisor, Wishnu Prasetya, for his support, patience, and guidance. His door has always been open for me and I have enjoyed our, occasionally lengthy, conversations. Secondly, I would like to thank my family for their unwavering support and encouragement. Especially, my parents who made it possible for me to study, without their guidance I would not be where I am today. Also I would like to thank my brother-in-law for his advice regarding my thesis. Finally, I would like to thank my fellow students with whom I have shared the “ST-Lab” for all the distractions and advice.
5 Conclusion

5.1 Concluding Remarks ................................................. 73
  5.1.1 Soundness and Completeness ................................. 73
  5.1.2 Code Coverage .................................................. 74
  5.1.3 Dynamic versus Static ........................................ 74

5.2 Future Work .......................................................... 75
  5.2.1 Enhancing the Prototype ....................................... 75
  5.2.2 Improving Test-conditions ..................................... 75
  5.2.3 Automatic Testing .............................................. 75
  5.2.4 Control and Feedback .......................................... 76

Appendices

A Installation and Configuration ................................... 79
  A.1 Code Availability ................................................ 79
  A.2 Usage .............................................................. 79

Bibliography ............................................................. 81
Chapter 1

Introduction

As with any engineering discipline, in software engineering the quality of the product (the software) is of grave importance. The IEEE Computer Society[25] defines software engineering as:

1. The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software.
2. The study of approaches as in (1).

It is generally accepted that validation and verification are important aspects of software engineering, and have a big influence on the quality of software. Therefore, they play a large role in most software processes. Validation and verification are commonly confused, but are not the same. In this thesis we will adhere to the description from Boehm[7], he eloquently describes the difference between them as:

- “Validation: Are we building the right product?”
- “Verification: Are we building the product right?”

Manual testing is the most popular validation and verification method that is used today. The term testing is not strictly defined and can involve a lot of different activities. Ian Sommerville identifies two main categories of testing in his book Software Engineering 8[28]. These categories are validation testing and defect testing. The goal of validation testing is “To demonstrate to the developer and the customer that the software meets its requirements.” The goal of defect testing is “To discover faults or defects in the software where the behaviour of the software is incorrect, undesirable or does not conform to its specification.” Hence, for validation testing, a successful test is a test where a system performs correctly. Whereas for defect testing, a test is successful when the system performs incorrectly. Following Boehm’s definition, defect testing falls under the verification activity. It has to be noted that testing cannot demonstrate that a system is without faults. As Edsger Dijkstra [28] stated “Testing can only show the presence of errors, not their absence.” The overall goal of testing is to convince developers and customers that the software is good enough for operational use, it inspires confidence in the software. In this thesis the term testing refers to the defect testing category, unless stated otherwise.

An illustrative example of the popularity of testing can be seen in the fairly new and “hyped” software development methodology, eXtreme Programming (XP) in
which testing plays an essential role. In XP developers are encouraged to write unit tests before writing the actual code. When a bug is found, a unit test must be written to prevent it from returning, known as regression testing. Furthermore, most production code is pair programmed, also known as code inspection which essentially is “human” testing. This systematic and disciplined approach pays off and software quality increases. However, software testing is a labour intensive and expensive process. Software testing usually consumes about 50% of the total cost involved in developing software [6, 21], whilst adding no direct functionality to the product. Yet, testing is frequently not done well, e.g. how often do you have to update the software on your computer? In a fairly recent study by the National Institute of Standards and Technology it was found that software errors alone cost the U.S. Economy about $59.5 billion annually[22]. That same study states that one-third of this cost might be saved by improving the software testing infrastructure. Enough incentive to explore different approaches to testing compared to current industry standards and practises.

A logical approach is the (partial) automation of verification via CASE tools. The popularity of CASE tools peaked in the early 1990s, but the hype could not live up to the expectations. Recently, a more realistic comeback of CASE tools can be observed. For example, almost every programming language has a unit-test framework — a framework for testing individual units of software. Also, other popular tools like version management, build management and bug tracking systems, are part of most software projects. Static software verification and formal specification of source code is getting more popular, as can be seen in the growing popularity of the Java Modeling Language[16]. Static verification can be used as a complement to testing or can even (partially) replace testing. In either case the use of static verification tools can notably increase the quality and efficiency of the verification process.

A distinction can be made between two different categories of static verification tools, namely specification-based verification and specification-less verification. As opposed to specification-less verification, specification-based verification requires not only the source code of the software, but also a specification. The specification-based category consists of theorem provers like, ESC/Java and HOL, and automatic testing tools like Korat and TestEra. These tools determine if a program is correct with respect to its specification. The specification-less category consists of tools that perform an automatic program analysis searching for typical errors and mistakes, some example tools are PMD, JLint and FindBugs. Because of the heuristic nature of these tools they are often less accurate than the specification-based category. However, on occasion they find errors impossible to find via specification based verification.

Despite the existence of some very nice CASE tools, static verification is an area that still holds a lot of challenges. This thesis project focuses on specification-based testing. We combine techniques used in static verification with techniques used in dynamic testing, resulting in a novel technique we call static testing.

This thesis is divided into the following chapters: chapter 2 provides an more elaborate introduction to static testing. Introducing some basic concepts and the thesis project itself. Chapter 3, describes the approach we take to achieve a static testing enviroment. In favour of readability, it does not contain any code of the actual implementation. One might even view it as an elaborate specification. However, most of the functions do have a direct equivalent in the implementation. Chapter 4 presents parts of the implementation of the prototype we deemed most interesting.

1Computer-Aided Software Engineering
Introduction

So it does not present the full implementation. Finally, in chapter 5 we wrap up and reflect upon our approach and possible future work.
Chapter 2

Background and Project

In this chapter the thesis project is presented. Also background information concerning software testing is provided.

SWEBOK[2] defines software testing as:

“Software testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior.”

Verification is somewhat of a loaded term in software engineering and exactly which activities are classified as such differs for some people. In this thesis the term verification applies to any activity which verifies (a part of) the actual behaviour of software against its expected behaviour. However, a distinction is made between dynamic and static verification. All verification activities which require the execution of the software are classified as dynamic verification while all activities that do not are classified as static verification.

Ammann and Offutt[3] define testing as:

“Evaluating software by observing its execution.”

As the definitions illustrate, the typical view on software testing is that in order to evaluate the behaviour of software it must be executed. In this project this is labelled dynamic testing. In the work presented in this thesis an alternative is proposed, static testing. Instead of executing the program, techniques from static analysis are used and a static test environment is created. Clearly, dynamic testing is a form of dynamic verification. Whereas static testing is a form of static verification, i.e. the program is not executed. One of the goals of this thesis project is the creation of a proof of concept static test tool. Two main challenges are identified for this tool: automatic generation of test-data and automatic generation of test-conditions. Due to time restrictions only the second challenge is pursued.

2.1 Dynamic Testing

An important facet of modern software testing is unit testing. It is the most commonly used technique for functional testing. Unit testing is at the bottom of the testing hierarchy and is a test-method that evaluates the smallest testable unit of
Chapter 2

an application. What exactly constitutes a unit can vary between languages, but usually it is a procedure, function or method. Unit testing was popularised by the SUnit and JUnit testing frameworks. Respectively, the frameworks are intended for the programming languages Smalltalk and Java. Nowadays almost every programming language has its own unit testing framework. The frameworks help automate testing and provide a “standard” testing paradigm. The level of automation is usually quite limited — a developer defines test-cases, test-data and the expected results, for a particular unit and the framework can automatically execute these test-cases and observe the test results. A big advantage is that the tests can easily be repeated.

One of the key features of unit testing is isolation. Ideally a unit must be tested in total isolation from other units, thus reducing complexity and making “bug hunting” a lot easier. While this is great in theory, in practise it is not as easy. Consider the object oriented language Java for example. A method in Java usually is not a pure function, meaning that it may read or write to the program state. Therefore, if proper isolation is needed, control of that part of the program state the unit interacts with is necessary. Typically, in an object oriented language the program state can be controlled using test doubles. A test double is a generic term coined by Meszaros [20] assigned to any kind of pretend object used in place of a real object. There are four distinct test doubles:

- **Dummy:** Sometimes a unit only passes an object around without actually using it itself, in this case a dummy object can be used. Any object can be used as long as its type correct.

- **Fake:** A fake object implements the same functionality as the object it replaces, but in a much simpler way. Therefore, it is not suitable for production, but is less likely to contain errors. Fake objects are often used to replace objects that are not finished yet.

- **Stub:** A stub does not implement the same functionality as the object it replaces, instead it gives specific answers to specific questions. It is specially designed with a specific test (or series of test) in mind. The stub explicitly controls what the unit “reads” from the program state and thus can force the unit down a specific path.

- **Mock:** A mock object is very much like a stub and they are often confused, but are not the same[12]. The emphasis of a mock object is on what the unit “writes” to the program state. It verifies the side effects the unit has on the program state.

2.1.1 Issues

While dynamically testing a program is a very natural way of software testing, it is not without its share of difficulties. Of course there are many existing tools that can help a developer. Often these tools use a form of static analysis, but in order to illustrate the problem “pure” dynamic testing is assumed.

**Isolation**

Notice that test doubles can only “stand in” for objects and thus operate on a inter-class level, not on a intra-class level. This means that they can isolate classes from each other, but not the methods within a class itself.
Also constructing a desired program state often poses a considerable challenge. An illustrative anecdote; a friend, who works for a secondment company, told a story about the new company where he was posted. The company had a problem, the execution of their test-suite was taking too long. Upon investigation it became evident that their test doubles, or the lack thereof, were poorly designed. There was an unit test which opened a database connection, ran a query to obtain a desired result and closed the connection for every single value of a set of test-data. Imagine doing this for a set of a 1000 test values! Of course the developer should not have opened a database connection at all. A better solution would have been to build a stub for the result set. However, this incident does illustrate that designing and implementing test doubles often distracts the programmer from the actual task, the test-cases themselves. Also because side effects are not easily identified and imitated, the manual construction of test doubles can be very complex. As a result, in practise this often results in poorly designed and minimalistic test doubles. Furthermore, the operational semantics of a language can stand in the way of properly testing a unit in isolation. Consider Java for example, the only method of testing a private method is through a public method which clearly compromises isolation. A possible solution is to include the test-cases in the class itself, but this is a typical example of bad separation of concerns. A last resort is to circumvent the limitations of Java all-together and call the method via reflection. While this last option is the least of two evils, it is far from ideal.

**Improving Test-adequacy**

The goal of testing is to build confidence in the software that is tested. Because software testing can only evaluate a finite number of test-cases, it raises the question: how do we determine if the software is adequately tested? This is where code coverage comes into play. Code coverage is a measure that describes the degree to which a piece of source code is tested. To measure code coverage one or more coverage criteria can be used. Some common criteria are:

- **Statement coverage**: Are all statements of a program executed at least once?
- **Branch coverage**: Are all branches of a decision structure (if statements etc.) executed at least once?
- **Path coverage**: Are all possible paths through a program executed at least once?

Measuring the code coverage of a test-suite for dynamic testing can be done via instrumentation. However, improving the code coverage when it is inadequate is problematic. For example, assume a software project that has set their mark of path coverage on 90%, but measurements indicate that the actual path coverage is 75%. Identifying inadequate test-cases and improving them relies on the wit of the testers and their understanding of the programs. Without the use of static program analysis tools this task is like finding a needle in a haystack instead of a systematic and disciplined approach.

**Efficiency**

Test-suites are often inefficient, because the cost of executing the test framework is often higher needed. This is due to the overlap of executions, different test-cases performing the same (partial) executions.
It is quite common that a test-suite evolves simultaneously with the development of the software. Different developers test the same piece of software with different requirements in mind. Often they do not take into account that an execution (or partial execution) can be shared between test-cases. As a result a test-suite contains a lot of test-cases that can be merged. Even when different test-cases do not perform the same execution, the test-cases can often be adapted and merged so they do — without compromising the test-adequacy. With a more disciplined testing approach it is possible to construct a test-suite that “shares” more test-executions, but it is complex to identify which test-cases can be shared and which can not. Also it would make a already complex and expensive process even more complex and expensive.

Portability

In order to prevent portability issues, a lot of companies build their software using the Java or .Net platform. But there is a lot of software that needs to work on different platforms and is build using C, C++ or even Javascript. The term platform is used loosely and it can include CPUs, operating systems, compilers, libraries or even browsers. Variations in platform can wreak havoc on a dynamic test-suite, because the results of executing that same test-suite can differ from platform to platform. Therefore more test-cases are required to properly test portable code, increasing the already high costs of testing.

2.2 Static Testing

While dynamic testing is the conventional approach to software testing, static analysis has been used in this field earlier. There are several research projects that have, to some degree, explored the use of static analysis techniques to benefit dynamic testing, e.g. DART[1] and Korat[18]. In this thesis, however, the focus is on a purely static testing framework.

The term static testing is not very common, in the past it has been used to refer to static checking and sometimes even software inspection. See for example, the work of Korel [15] or by Marshall [19]. Therefore, in the interest of clarity, the definition of static testing used in this project is defined using the definition of software testing from SWEBOK with some small modifications:

"Static testing consists of the static verification of the behaviour of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behaviour."

The only difference between dynamic and static testing is how they verify the actual behaviour of a program against its expected behaviour. Testing can be divided into two distinct activities, first observing the actual behaviour, and second comparing it to the expected behaviour. In the case of dynamic testing both activities are straight forward, especially if the program is written in a programming language that is supported by an unit test framework. In any case, it usually boils down to executing a test-case written in the same programming language, capturing the results and comparing them to the expected results, which are also written in the same language. In the case of static testing, however, it is not as easy. Since by definition the program can not be executed, an alternate means to infer the actual behaviour is necessary. Also an alternate method for specifying the expected behaviour and comparing it to the actual behaviour is needed.
By the above definition of static testing, any testing approach which does not execute the software falls under static testing. In this paragraph a high level overview is presented of the static testing approach taken in this thesis project. It shall be explained in more detail in section 3.1. The approach adheres to the unit testing methodology, meaning that the framework must be able to test units in isolation. In order to be able to define the expected behaviour a formal specification language is used. The specification is in the form of a Hoare triple [13]. Let us assume an arbitrary piece of software \( s \) which must be tested. Its expected behaviour is represented by a pre-condition \( P \) and a post-condition \( Q \). These three variables form the Hoare triple \( \{P\} s \{Q\} \). In order to statically determine the actual behaviour a third predicate is constructed, called the verification-condition, which is build using the weakest pre-condition calculus. It is inferred from the program and the post-condition. The testing of the program its behaviour is achieved by transforming the verification-condition into another predicate, called a test-condition. The verification-condition is a logical implication of the following form, \( P \Rightarrow wp(s, Q) \).

The left and right hand side of the implication can both contain variables. In order to get a test-condition some or all of these variables are instantiated, note that the values used take on the role of test-data. It is often possible to instantiate multiple test-conditions from a single verification-condition. Finally, each test-condition is solved, meaning that the validity of the predicate is computed. Since no execution of code occurs this process statically tests the actual behaviour against the expected behaviour. Note that the test-conditions only verify a part of the verification-condition. While it would be better to completely verify the verification condition, verifying an arbitrary predicate is undecidable. Therefore, as with dynamic testing, static testing can only show the presence of errors not their absence.

An important part of this thesis project is the construction of a proof of concept tool for our static testing approach. The main goal of this tool is the generation of the above mentioned test-conditions. If a test-condition is invalid, a violation and a set of test-data which uncovered the violation is found. The data can then be used to also produce the violation dynamically. However, because a specification can be flawed, a violation is not a bug or error in the program per se. In this thesis a violation is defined as:

A violation is an inconsistency between the behaviour of the program, and its specification.

With the data that led to the violation, a developer should investigate the program and determine if the program contains an error or if the specification was flawed.

There are numerous projects that identify the same issues of dynamic testing as those in section 2.1.1. Most of them try to find solutions while retaining the dynamic testing environment. However, we believe that the most of the problems originate from the fact that the program under test is executed. More specifically, that the functional testing of a unit via executing is the cause of most of the problems of dynamic testing, because the unit is executed in a foreign runtime environment. For example, in order to test a unit in isolation there is often the need to create an artificial runtime environment using dummy objects. This can be very complex and often takes up a lot of time.

Furthermore, the semantics of the binary program can not be changed. The binary program is explicitly mentioned, because in order to execute the program a binary version is required via either compilation or interpretation. Unfortunately, both are fixed and bound to the operational semantics of a language. So some rules which are
In theory, static testing does not necessarily have the same difficulties solving the issues in 2.1.1, because it is not bound to the operational semantics of a language. For static testing a custom semantics can be defined in terms of the weakest pre-condition semantics. A condition is that it must be consistent.

2.3 Motivation

For automatic generation of test-conditions, the traditional choices made for the weakest pre-condition generation need to be reconsidered. What do we do with loops? What do we do with a procedure call? What if the procedure call is recursive? If and how do the background predicates influence the test-condition generation? Is it possible to reuse an existing weakest pre-condition calculus? Is it possible to calculate a more precise weakest precondition than previous possible? If so, what kind of optimisations can we perform and how precise can we make the test-conditions?

Besides generating test-conditions we also want to know if our approach to static testing using the weakest pre-condition is any good. Is the weakest pre-condition a sufficiently strong formal representation of the behaviour of the program? Do we get false positives and if so, are these acceptable? Or does our approach miss some obvious programming errors? In practise it is possible that the program under test is partially specified, how do we cope with partial specifications? Can we gracefully degrade our results?

Because this is uncharted territory, the list of possible questions and issues we could state can go on and on. However, in this thesis we focus on one essential question:

To what extent is static testing a feasible testing technique?

Since at the beginning of this project there was no static testing tool, we could not perform an empirical study into the feasibility of static testing. Instead we build a proof of concept tool to get a “feel” for it.

2.4 Scope

The general principle of static testing allows for a broad range of possible research directions. However, a scope is required for this thesis project. As a rule of thumb the focus is on the primary techniques needed to construct our notion of a basic static testing environment.

As mentioned earlier, automatic generation of test data is not considered in this thesis. Nonetheless, it is a very interesting topic and would be a very nice feature for the tool. Also having a specified program as starting point, makes this a realistic option. However, automatic test generation focuses on a completely different, complex and elaborate area of research and was therefore dropped.

Also no precautions are taken to cope with programs that do not have a specification. Indeed better results can be expected for a program with a detailed specification, but there are no measures to enforce this. A program is simply accepted as it is and assumed that it is sufficiently specified to perform the test cases required by the developer.

Static testing uses an intermediate language named BoogiePL[10]. This is not an executable programming language, but an (imperative) intermediate language
Background and Project

intended for static verification. Obviously this means that in order to test a real program written in any language, it must first be translate from that language to BoogiePL. In this thesis the challenges and problems involving such translations is not considered. Luckily translators for C# [26] and Java [17] already exist.

Another very interesting topic is comparing our static testing tool against a dynamic testing tool in terms of performance, code coverage and other metrics. However this thesis does not contain any empirical research of that sort.

2.5 Contribution

The first and foremost contribution of this thesis is a notion of static testing. We combine concepts and techniques from the field of testing and the field of static verification. For example, our approach uses the ideology of unit testing from testing and the weakest pre-condition calculus from static verification. To our knowledge this is a novel combination.

The second contribution is actually an implication of the above. Static testing brings a more formal verification technique in range of agile software processes. Agile software development is currently very popular, and is dominated by dynamic testing. An important rule from the rules of eXtreme Programming is “Code the Unit Test First”. With our approach we could easy replace the rule by “Specify the Unit Test First”. Besides making the process more formal it also makes it even more “agile”, because constructing and adjusting a test-suite requires more effort than constructing and adjusting a specification.

The third contribution is a prototype tool (see chapter 4). This tool contributes a weakest pre-condition calculus for BoogiePL. Furthermore it contributes a control flow equation for BoogiePL and a technique to linearize a control flow graph. However the tool is intended as a proof of concept and should be regarded as such.

The final contribution of this thesis is the exploration of static testing. Since this is an unexplored area of research, it was unclear exactly what path to take. Therefore, different alternatives or related techniques were explored, some turned out to be useful, and some did not.

2.6 TestBPL

In this thesis project a prototype tool was constructed, named TestBPL. Figure 2.1 represents the basic flow of TestBPL. In this figure the rounded rectangles represent an activity and the rectangles with the turned corner represent intermediate results. Before the tool can begin its static testing process, the program under test must first be translated to an intermediate program. This is an equivalent program described in a simpler language than the source program. The first step in the static testing process is the linearization of the program, which is one of the major topics of this thesis. The result of the linearization process are traces, which are a symbolic representation of all the possible paths through the program under test. Next the wp calculator turns these traces into linear verification conditions, which are predicates that are consistent with their respective trace. The final step that TestBPL performs is the construction of test-conditions. A test-condition is created during the instantiation process, which substitutes some or all variables in the linear verification condition with its respective test-data. Note that the developer must
therefore provide a test-suite that is consistent with linear verification conditions. Finally, a separate solver is required in order to verify the test-conditions.

Figure 2.1: TestBPL
Chapter 3

Approach

Contrary to dynamic testing, static testing can not execute a program to verify its behaviour. Static testing requires an alternate approach to verify the behaviour of a program. In this chapter our approach to static testing is presented. Section 3.1 explains some basic concepts and terminology used in our static testing approach. In the next section 3.2 BoogiePL is introduced, the intermediate language on which TestBPL operates. Section 3.3 defines an weakest pre-condition calculus for BoogiePL and shows the problem of computing it. The subsequent section, section 3.4, introduces the notion of program instances. In section 3.5 a description of the linearization process is given. Then in section 3.6 it is shown how TestBPL calculates the weakest pre-condition for the test-conditions. Finally, in section 3.7 an example is presented, which shows the entire process. Int this chapter, however, the focus is not on the implementation details, those can be found in chapter 4. From this point on when “static testing” is mentioned, our specific approach to static testing is referred to, unless otherwise specified.

3.1 Background

Modern programming languages like Java or C# have a lot of features and building a static testing framework that copes with all these features is complex. The main advantage of translating a program written in a modern language into an intermediate language is that the intermediate language is much simpler. So building a static testing framework for the intermediate language is also much simpler. An additional advantage is that it is easier to support multiple programming languages. Inevitably there are also downsides, e.g. providing sensible feedback in the context of the modern language. The employment of an intermediate language is quite common among verification tools, for instance ESC/Java2 [9] and VeriJava [33] use an intermediate language based on the guarded command language by Dijkstra [11], whereas Spec# [5] uses BoogiePL.

As stated before, testing can be divided into two basic activities, namely observing the actual behaviour of a program and comparing that to the expected behaviour of that program. When testing dynamically the expected behaviour is defined by programming a test-case just like the program itself. The actual behaviour is obtained by execution and then compared to the pre-programmed expected behaviour. When testing statically a formal specification expressing the expected behaviour is needed. This specification is expressed in a specification language. In some cases the specification and programming language are integrated like BoogiePL, but usu-
ally they are distinct languages like JML and Java. Most specification languages
have their roots in a simple language invented by C.A.R. Hoare back in 1969 [13].

3.1.1 Hoare Logic

In the Hoare style a specification is written as follows:

\[
\{P\} \; s \; \{Q\} \tag{3.1}
\]

The parts between \{\} are called assertions and are formulae in predicate logic. Respectively, the \(P\) and \(Q\) are the pre- and post-condition of \(s\), which is a program or partial program. These three parts together form a Hoare triple. Intuitively a triple may be interpreted as “If the assertion \(P\) is true before initiation of a program \(s\), then the assertion \(Q\) will be true on its completion.” For a more formal interpretation some basic concepts are introduced.

The state of a program can be described as a set of variables and their values at a given time. An assignment in the program will transfer the program from one state to the other. The set of all possible states is denoted as \(\Sigma\). A predicate can be seen as a set of states, in which case a pre-condition specifies the set of allowed initial states and a post-condition specifies the set of all possible final states of a program. Note that \(P\) and \(Q\) are subsets of the \(\Sigma\) of \(s\). With respect to this “set interpretation” of predicates a machine that can execute a program can be abstractly modelled as:

\[
\text{exec} : \Sigma \rightarrow S \rightarrow \mathcal{P}(\Sigma) \tag{3.2}
\]

This function takes an initial state and a program, and returns a set of possible final states. It returns a set of final states, because a program can be non-deterministic and non-terminating. A deterministic program will return a single state, whereas a non-deterministic program will return a set of possible final states, and for a non-terminating program an empty set is returned.

Under the set interpretation of predicates the Hoare triple can be abstractly and semantically defined as follows, where \(\sigma \in \Sigma\):

\[
\{P\} \; s \; \{Q\} = (\forall \sigma \in P \cdot (\text{exec } \sigma \; s) \subseteq Q) \tag{3.3}
\]

If there exists a state \(\sigma\) in \(P\) which does not satisfy this predicate, then the specification and the program \(s\) are inconsistent, such a \(\sigma\) is denoted as \(\hat{\sigma}\). Assuming the Hoare triple as correctness criteria for a program, it should be clear that finding a \(\hat{\sigma}\) disproves the correctness of \(s\). It must be noted that finding such \(\hat{\sigma}\)'s is the goal of testing in general.

In program verification a distinction is made between total and partial correctness of a program. Partial correctness requires that \(s\) if a program terminates, it will terminate in a correct state. Total correctness, on the other hand, also requires the termination of the program. The above model of the Hoare logic is somewhat simplistic and can only prove partial correctness. This can also be seen in definition 3.3. Note that when \(s\) is a non-terminating program, the \((\text{exec})\) will return an empty set. Because the empty set is the subset of any \(Q\), the model effectively assumes all non-terminating programs to be true. Since software testing basically “tests” the partial correctness of a program, the model in 3.3 is sufficient.

While the Hoare triple gives us a means to specify the expected behaviour, definition 3.3 still induces the execution of a program to verify it. So another model for static testing is needed.

\(^1\)C.A.R. Hoare
3.1.2 Verification Conditions

Instead of verifying a Hoare specification dynamically through execution we could also try to verify it statically through theorem proving. A commonly used strategy in manual and automatic theorem proving is the weakest pre-condition reduction strategy [24]. The weakest pre-condition is the largest set of initial states that guarantees that a program terminates in a valid final state. So rather than proving the specification \( \{P\} s \{Q\} \), it states that it is sufficient to prove that \( P \) implies the weakest pre-condition of \( s \) with respect to \( Q \), formally:

\[
\{P\} s \{Q\} = \models P \Rightarrow \wp s Q \tag{3.4}
\]

It is based on the rules of consequence [13], the pre-condition strengthening rule in particular, see 3.5.

\[
\models P \Rightarrow P'
\]

\[
\{P'\} s \{Q\} \quad \models P \Rightarrow \wp s Q \tag{3.5}
\]

The pre-condition strengthening rule states that in order to prove \( \{P\} s \{Q\} \) it is sufficient to find a weaker pre-condition \( P' \) for which \( s \) can establish \( Q \). Of course the original \( P \) must also imply the \( P' \). Unfortunately, when the implication \( P \Rightarrow P' \) does not hold, its cause is unknown. Either the \( P' \) is too strong or the specification \( \{P\} s \{Q\} \) is invalid. This problem can be solved, by constructing the weakest possible pre-condition of \( s \). The weakest pre-condition of \( s \) with respect to \( Q \) is denoted as, \( \wp s Q \).

In automatic theorem proving the predicate \( P \Rightarrow \wp s Q \) is called a verification condition. The validity of the predicate \( P \Rightarrow \wp s Q \) means that every value “in the set” \( P \) is also a value “in the set” \( \wp s Q \). Figure 3.1 visualises the relation between \( P \) and \( \wp s Q \).

\[\text{Figure 3.1: } P \subseteq \wp s Q \]

\[\text{Figure 3.2: } P \nsubseteq \wp s Q \]

Because the weakest pre-condition guarantees \( s \) to end up in \( Q \), the figure also shows that the validity of the predicate \( P \Rightarrow \wp s Q \) is sufficient to prove the correctness \( \{P\} s \{Q\} \). On the other hand, if the predicate \( P \Rightarrow \wp s Q \) does not hold, it means that there are states in \( P \) which are not in \( \wp s Q \), see figure 3.2. Note that \( P \setminus \wp s Q \) represents the set of inputs to the program \( s \) that will lead to erroneous behaviour, these are the previously mentioned \( \hat{\sigma} \) states. If it can be prove that this set is not empty, the correctness of the \( \{P\} s \{Q\} \) specification is disproven.

**Weakest Pre-condition**

Definition 3.6 gives an abstract model of the weakest pre-condition. However, the weakest pre-condition strategy requires that it must be possible to calculate the weakest pre-condition of a program. Equations 3.7 and 3.8 present concrete examples of calculation rules. The \( \wp \) function abstracts from the semantics of the
Chapter 3

machine (exec) and introduces its own set of semantic rules. The wp and similar functions are known as predicate transformers, because they transform one predicate into the other.

\[
\begin{align*}
wp & : \mathcal{S} \rightarrow \mathcal{P}(\Sigma) \\
wp s Q & = \{ \sigma | \text{exec } \sigma s \subseteq Q \} 
\end{align*}
\]

(3.6)

The wp function expects two arguments, a program and a post-condition and it returns the weakest pre-condition. As opposed to the machine semantics, the wp function starts at the post-condition and works backwards through the program. This reverse behaviour is formally defined by the following calculation rule, where \( s_1 \) and \( s_2 \) are subsequent statements.

\[
wp (s_1; s_2) Q = wp s_1 (wp s_2 Q) 
\]

(3.7)

Consider listing 3.1, the program adds five to the variable \( x \) via two separate assignments. The weakest pre-condition of an assignment is subtle, but shall be explained in more detail later on. For now assume that an assignment \( v := e \); replaces all occurrences of the variable \( v \) by the expression \( e \) in \( Q \), see definition 3.8.

\[
w p (v := e;) Q = Q[v/e] 
\]

(3.8)

Example

With these two rules we can calculate the weakest pre-condition of listing 3.1:

\[
\begin{align*}
wp (x := x + 2; x := x + 3) (x > 10) \\
= \wp (x := x + 2) (wp (x := x + 3) (x > 10)) \\
= \wp (x := x + 2) ((x > 10)[x + 3/x]) \\
= \wp (x := x + 2) (x + 3 > 10) \\
= (x + 3 > 10)[x + 2/x] \\
= (x + 2 + 3 > 10)
\end{align*}
\]

This example shows that the wp function incrementally transforms one predicate

\[
\begin{align*}
\{ x > 5 \} \\
x := x + 2; \\
x := x + 3; \\
\{ x > 10 \}
\end{align*}
\]

from

\[
\begin{align*}
\{ x > 5 \} \\
x := x - 2; \\
x := x + 3; \\
\{ x > 10 \}
\end{align*}
\]

Listing 3.1: Correct add5

Listing 3.2: Incorrect add5

into another, so the weakest pre-condition of \( x := x + 3 \) is the post-condition of \( x := x + 2 \). Because the weakest pre-condition is calculated from the program statements themselves, it represents the actual behaviour of the program and can be used for static verification. Intuitively, it is easy to see that listing 3.1 is correct, since the weakest pre-condition \( x + 2 + 3 > 10 \) is equivalent to the pre-condition. Formally, a verification condition needs to be constructed, for listing 3.1 this is
$x > 5 \Rightarrow x + 2 + 3 > 10$. The verification condition is valid because the left- and right-hand sides of the implication are equivalent. Since the truth table for logical implication, table 3.1, shows that when both sides are equal the implication holds, the correctness of listing 3.1 is formally proven.

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>$P \Rightarrow Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>

Table 3.1: Truth table for logical implication

Calculating the weakest pre-condition of listing 3.2 results in the verification condition $x > 5 \Rightarrow x - 2 + 3 > 10$ or in short $x > 5 \Rightarrow x > 9$. This verification condition is invalid, because it is possible that the right-hand side is true while the left-hand side is not, so the implication fails. Or in the set interpretation, $P \setminus \wp s Q = \{6, \ldots\} \setminus \{10, \ldots\} = \{6, 7, 8, 9\}$ is not empty. This notation provides four counter examples for which the verification condition fails. So the correctness of listing 3.2 is formally disproven.

### 3.1.3 Test Conditions

A program $s$ can be viewed as a, possibly infinite, set of program instances. A program instance is an “incarnation” of $s$ containing no loops or branches, it is merely a, possibly infinite, sequence of statements. The goal of theorem proving is to prove the correctness of all program instances, while the goal of static testing is only to validate the correctness of a single program instance. So while a test condition “looks” the same as a verification condition, $P \Rightarrow \wp s Q$, the semantics of the $\wp$ function are different.

There is an important difference in the calculation of the weakest pre-condition of a loop. Because a verification condition must represent all possible program instances, it must represent all possible iterations of a loop. Unfortunately, there is no known algorithm that can calculate the weakest pre-condition of a loop in general. The solution for verification conditions usually involves a sound approximation of the weakest pre-condition of the loop and often requires an additional specification called a loop invariant. The danger of this approach is over- or under-approximation of the loop invariant, making the verification condition unsolvable. Test conditions, on the other hand, represent a single program instance; in such cases, every loop of the original program is assumed to be executing a predetermined number of iterations. Of course every loop may have a different number of iterations, but all are predetermined. This assumption effectively removes the loops, and the weakest pre-condition can be calculated precisely.

Figure 2.1 shows that the input to the $\wp$ calculator is not just the intermediate program, but also something called a trace. Traces will be discussed in further detail in section 3.5, for now assume that a trace together with the program form the program instances.

The $\wp$ calculator in static testing calculates a weakest pre-condition for every program instance given. Because a program instance does not contain any loops we call the resulting conditions, *linear verification conditions* as to distinguish them from ordinary verification conditions. To turn a linear verification condition into a
test condition we perform one final action. Some or all variables and quantifications are replaced by their respective test values. This happens in the “Instantiater” block in figure 2.1. The combination of linear verification conditions and instantiation makes a test condition more precise and easier (sometimes even trivial) to prove. A test condition is defined as:

A test condition is a predicate obtained by instantiating some or all variables in a linear verification condition.

3.2 Intermediate Language

As presented earlier, the first step in the process of static testing is the translation from a source language to the intermediate language. There are basically three possible intermediate languages to choose from, namely BoogiePL (BPL), the guarded command language (GCL) or a custom intermediate language. Designing a custom language from scratch is cumbersome, which leaves the GCL and BPL languages. The BoogiePL language was chosen as the intermediate language, because it is quite popular, clearly defined, and there are existing translators for JML (Java) and Spec# (C#) to BPL. The GCL language is not clearly defined, as there are many dialects that have been derived from the original paper by Dijkstra [11].

3.2.1 BoogiePL

BoogiePL was developed at Microsoft Research [26] as part of the Spec# system [5]. Architecturally, the Spec# programming system consists of three main parts, see figure 3.3. The Spec# Compiler compiles a Spec# program, an annotated C# program, into .Net CIL bytecode. It differs from an ordinary compiler, because it not only compiles an executable, but also preserves the specification. The next step is a translator that translates the bytecode to BoogiePL. Since the translator consumes bytecode instead of source code it is possible to add support for other languages, as long as they can be compiled to annotated CIL bytecode. The Boogie verifier consumes the BPL program and generates verification conditions that are fed to an automatic theorem prover, called Simplify. The theorem prover will attempt to automatically prove the conditions, but might not always succeed.

In listing 3.3 and listing 3.4 an example translation is presented. In listing 3.3 the source program, a class called Example, is presented. Listing 3.4 contains the translated version of that class. An object oriented programmer might miss the constructor method in the translated version, which is implicit in the Spec# source program. It is intentionally left out in favor of brevity, because even a small Spec# program is translated into a fairly lengthy BPL program. The translated program in listing 3.4 shows the main features of the BoogiePL language. BoogiePL is designed for program verification and program analysis and therefore has some oddities compared to a “normal” programming language. For example, in a normal programming language parameters are often passed by reference, because it is more
public class Example {
    int x;
    public int P (int y)
       requires x > 0
       ensures y > 0 ⇒ x = old(x) + y
       ensures (y > 0 ⇒ result = old(x) + y) ∨ (y ≤ 0 ⇒ result = old(x))
    {
        if (y > 0){
            x += y;
        }
        return x;
    }
}

Listing 3.3: Spec# Example

BoogiePL is a procedural language tailored for modular verification, meaning that a BPL program, like listing 3.4, is self-contained. Since BPL does not allow any form of inclusion of other BPL programs, all relevant information must be in the BPL program itself. Because Hoare logic alone is not expressive enough for this self-contained design, BoogiePL has a special mathematical part. This part allows background predicates to be coded directly in BPL, for example the axiomatic heap in listing 3.4. In the mathematical part of the language axioms, symbolic constants, and uninterpreted functions are found. The uninterpreted functions are fundamentally different from procedures as they do not allow any implementation. However, constraints can be added to a function or a symbolic constant through axioms, e.g. axiom Example <: System.Object. The <: operator in this example provides partial order on names, and this axiom represents the inheritance relation between Example and System.Object.

Besides the mathematical part, BoogiePL allows global variables (e.g. var $Heap : [ref, name]any,) and procedures. A procedure can have a specification, similar to the Hoare triple, but the syntax is a little different. As can be seen in listing 3.4, a procedure has a signature, an optional specification and an optional body. The returns part of the signature is also optional. The body of a procedure can be defined separately from the procedure using the implementation construct. However, the specification must be defined in the procedure and consists of three different parts: a pre-condition (the requires clause), a post-condition (the ensures clause), and a list of global variables (the modifies clause). The modifies clause lists the global variables that the implementation of that procedure may assign to. The body of a procedure is made up of local variables, labels, commands and transfer of control commands.
Chapter 3

var $Heap : [ref.name]any;
function $typeof(ref) returns (name);
function $Is(any.name) returns (bool);
function $NotNull(name) returns (name);

axiom(∀ o : ref, T : name • $Is(o, T) ≡ o = null ∨ $typeof(o) <: T);
axiom(∀ o : ref, T : name • $Is(o,$NotNull(T)) ≡ o ≠ null ∧ $Is(o,T));
const System.Int32 : name;
const System.Object : name;
const Example : name;

axiom Example <: System.Object;
const Example.x : name;

procedure Example.P$System.Int32(this : ref, $in : int) returns (Example.P.return : int);
  requires $Heap[this, Example.x] > 0;
  modifies $Heap;
  ensures $in > 0 ⇒ $Heap[this, Example.x] = old($Heap[this, Example.x]) + $in;
  ensures $in ≤ 0 ⇒ Example.P.return = old($Heap[this, Example.x]) + $in

implementation Example.P$System.Int32(this : ref, $in : int) returns (Example.P.return : int)
{
  var y : int, stack0i : int, stack0b : bool;
  entry :
  assume $Is(this, $NotNull(Example));
  y := $in;
  assume $Is(y, System.Int32);
  block1258:
    block1258 :
      stack0i := 0;
      stack0i := y ≤ stack0i;
      goto true1258to1292, false1258to1275;
    true1258to1292 :
      assume stack0b = true;
      goto block1292;
    false1258to1292 :
      assume stack0b = false;
      assert this ≠ null;
      stack0i := cast($Heap[this, Example.x], int);
    block1292:
      stack0i := stack0i + y;
      assert this ≠ null;
      $Heap[this, Example.x] := stack0i;
    goto block1292;
  return;
}
3.2.2 Setting the scene

In this thesis only a subset of the BoogiePL language is considered, which is referred to as BoogiePL$_S$. The most important compromises were made in the mathematical part of the language. More specifically BoogiePL$_S$ does not have the axioms, symbolic constants and functions of BoogiePL. It does, however, contain global variables and procedures. Basically, the specification part of BoogiePL$_S$ is restricted to Hoare logic. Another less significant compromise is the absence of the implementation structure, which separates the implementation and specification of a procedure, see listing 3.4.

Notations

Some notational conventions must be introduced before BoogiePL$_S$ can be defined. Two different notations for a sequence of $\alpha$ are used, where $\alpha$ can be any syntactic construct of BoogiePL$_S$. The first notation is quite conventional and is written as $[\alpha_1, \ldots, \alpha_n]$. However, a sequence may also be written using the bar notation $\overline{\alpha}_n$ (pronounced “alpha n bar”). Both define a sequence of $\alpha$ from 1 to $n$. A sequence may always be empty, thus optional, unless otherwise specified. The $\overline{\alpha}_n$ notation is used to explicitly denote an empty sequence. Two sequences $\overline{\alpha}_n$ and $\overline{\gamma}_n$ can be joined using the $\sqcup$ operator, which adds the $\gamma$ sequence behind the $\alpha$ sequence.

$$[x_1, \ldots, x_n] \sqcup [y_1, \ldots, y_n] = [x_1, \ldots, x_n, y_1, \ldots, y_n] \quad (3.9)$$

The $\langle \rangle$ brackets also have a special meaning in our notation. BoogiePL contains language constructs which allow white space. So in order to show where a language construct $\alpha$ begins and ends it is enclosed with $\langle \alpha \rangle$.

BoogiePL$_S$

BoogiePL$_S$ is defined by an abstract syntax in (3.11) — for a more complete definition of the BoogiePL language we refer to the DeLine and Leino [10]. The abstract syntax of BoogiePL$_S$ is divided into the following syntactic categories:

\[
\begin{align*}
\mathbb{P} & \in \text{Program} \\
\mathbb{P} & \in \text{Procedure} \\
\mathbb{S} & \in \text{Spec} \\
\mathbb{B} & \in \text{Body} \\
\mathbb{D} & \in \text{Decl} \\
\mathbb{B} & \in \text{Block} \\
\mathbb{C} & \in \text{Command} \\
\mathbb{T} & \in \text{Toc} \\
\mathbb{id} & \in \text{Name} \\
\mathbb{l} & \in \text{Label} \\
\mathbb{e} & \in \text{Expr} \\
\mathbb{a}, \mathbb{x}, \mathbb{y} & \in \text{Var} \\
\mathbb{\tau} & \in \text{Type}
\end{align*}
\]
The syntax of BoogiePL is given by the following abstract syntax:

\[
\begin{align*}
\mathbb{P} &::= (D_1, P_1, \ldots, D_n, P_m) \\
P &::= \langle \text{procedure } id \ (x_n : \tau) \ S_n \ \{b\} \rangle \\
&\quad \mid \langle \text{procedure } id \ (x_n : \tau) \ \text{returns} \ (y_n : \tau) \ S_n \ \{b\} \rangle \\
S &::= \langle \text{requires} \ e; \rangle \\
&\quad \mid \langle \text{modifies} \ x_n; \rangle \\
&\quad \mid \langle \text{ensures} \ e; \rangle \\
\beta &::= (D_1 \ldots D_n \mid B_1 \ldots B_m) \\
D &::= \langle \text{var} \ [x_1 : \tau, \ldots, x_n : \tau]; \rangle \\
B &::= \langle l : \ [C_1 \ldots C_n] \ T \rangle \\
C &::= \langle x := e; \rangle \\
&\quad \mid \langle a[e_1] := e_2; \rangle \\
&\quad \mid \langle \text{assert} \ e; \rangle \\
&\quad \mid \langle \text{assume} \ e; \rangle \\
&\quad \mid \langle \text{havoc} \ [x_1, \ldots, x_n]; \rangle \\
&\quad \mid \langle \text{call} \ \text{id}([e_1, \ldots, e_2]); \rangle \\
&\quad \mid \langle \text{call} \ [x_1, \ldots, x_n] := \text{id}([e_1, \ldots, e_2]); \rangle \\
T &::= \langle \text{goto} \ [l_1, \ldots, l_m]; \rangle \\
&\quad \mid \langle \text{return}; \rangle \\
\end{align*}
\]

A program \( P \) consists of a sequence of distinct global variable declarations \( D \) and procedures \( P \), in arbitrary order. The name of a procedure \( id \) must be distinct from any other procedure name in the program. Also the in- \( (x_n) \) and out-parameters \( (y_n) \) of a procedure must be disjoint, meaning that the variables of the sequences themselves must be distinct and that those sequences do not share any common variables.

A procedure can have a sequence of specifications. There are three kinds of specification clauses namely the requires, the modifies, and the ensures clause. A procedure can be called using the call command. The procedure that calls another procedure is known as the caller and the procedure that is being called is known as the callee. A requires clause specifies the state a caller must satisfy before calling the callee and the ensures clause specifies the state of the callee after the call. While a requires clause can only refer to the pre-state of a call, an ensures clause can refer to both the pre-state and post-state of a call via the old operator. In order to refer to the value of a variable \( x \) in the pre-state it is annotated with the old operator, like so old(\( x \)). Naturally the old operator may only occur in an ensures clause. Recall that BoogiePL passes its parameters by value, which implies that all parameters are local to a procedure. The out-parameters, however, are “special” because they are copied from the callee to the caller after a call. Before a call, the out-parameters are reset by BoogiePL, so they do not have a value in the pre-state and may not be referenced to in the requires clause. Furthermore, because BoogiePL passes its parameters by value, referring to the post-state of the in-parameters in the ensures clause is meaningless from the callers point of view. Therefore it is assumed that all occurrences of in-parameters in an ensures clause refer to the value of that parameter in the pre-state, its initial value. In order to keep the syntax consistent for all variables it is demanded that all occurrences of in-parameters in the ensures clause be annotated as old. Also when a call is executed BoogiePL assigns an arbitrary value to the out-parameters and the local variables of the callee. So it would be pointless to refer to the old value of an out-parameter. A procedure may always read from a global variable, but it may not always write to a global variable. A procedure may only write to global variables defined in the modifies clause.
The body of a procedure consists of a sequence of local variable declarations and a sequence of blocks. The local variables must differ not only from all other local variables, but also from the in- and out-parameters. There are four levels of scoping in BoogiePL S: the expression bound scope (e.g. \( \forall x: \text{int} \ldots x \)), the local variable scope, the parameter scope, and finally the global variable scope. During name resolution a variable is looked up in that particular order. If a variable is not found among these scopes, an “undeclared variable” error occurs. BoogiePL S allows a local variable to shadow a global variable, which can cause confusion. Even if a procedure specifies a variable \( x \) in its \texttt{modifies} clause, it can still be shadowed by a local \( x \) and all assignments are made to the local variable not the global variable.

The execution of a procedure starts at the first block in that procedure. All blocks start with a label that uniquely identifies a block and therefore must be distinct from all other labels. A sequence of commands follow the label, these commands can include:

- An assignment to a variable. The variable \( x \) must be in scope and the type of the expression \( e \) must be assignable to \( x \), meaning that they are of the same type or the type of \( x \) is \texttt{any}.

- An assignment to an array element. The variable \( x \) must be in scope and the type of \( e_1 \) must be assignable to the index type of \( x \) and the type of \( e_2 \) must be assignable to the type of the array element.

- An assert command. The expression \( e \) is evaluated and must be of type \texttt{bool}. If \( e \) evaluates to true the program continues and the control is transferred to the next command, but if \( e \) evaluates to false the execution “goes wrong” indicating a non-recoverable error.

- An assume command. Like the assert command, the expression \( e \) is evaluated and must be of type \texttt{bool}. If \( e \) evaluates to true the program continues and the control is transferred to the next command, but if \( e \) evaluates to false the program stalls forever preventing it from ever “going wrong”.

- A havoc command. The havoc command assigns an arbitrary value of the correct type to every variable in \( x_1 \).

- A call command. There are two kind of call command, one ignoring the return value of a procedure and another capturing it. The length of the sequence of expressions \( [e_1, \ldots, e_2] \) must be equal to the length of sequence of in-parameters of the callee, also the types of the expressions must be assignable to the types of the in-parameters. Once called, the command evaluates the expressions and passes these to the in-parameters and transfers control to the callee. Upon return from the callee, if the procedure has out-parameters, the value of the out-parameters are copied into the \( [x_1, \ldots, x_n] \) variables. Of course the length of the sequence of out-parameters must be equal to the length of \( x_n \) and also the types of the out-parameters must be assignable to the types of \( x_n \).

Finally a block must end with a special command that transfers control to another block via the \texttt{goto} command or back to the caller via the \texttt{return} command. These two commands are called the transfer-of-control (Toc) commands. The \texttt{goto} command is a little unusual, because it is non-deterministic. It arbitrarily chooses a label from the sequence and transfers control to the block identified by that label.
Expressions

The expression language of BoogiePLS differs somewhat from conventional expression languages, but it also contains a lot of similarities, like arithmetic, relational and logical binary operators:

<table>
<thead>
<tr>
<th>arithmetic</th>
<th>relational</th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>+ : int × int → int</td>
</tr>
<tr>
<td>subtraction</td>
<td>− : int × int → int</td>
</tr>
<tr>
<td>multiplication</td>
<td>× : int × int → int</td>
</tr>
<tr>
<td>division</td>
<td>/ : int × int → int</td>
</tr>
<tr>
<td>modulo</td>
<td>% : int × int → int</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>logical</td>
<td>∨ : bool × bool → bool</td>
</tr>
<tr>
<td>and</td>
<td>∧ : bool × bool → bool</td>
</tr>
</tbody>
</table>

the unary minus − : int → int and the logical negation ¬ : bool → bool. In addition to these standard features the expression language of BoogiePLS contains the logical binary operators for equivalence ≡ : bool × bool → bool and implication ⇒ : bool × bool → bool. Naturally it allows literals of its built-in types, like arrays $\text{Example}[\text{this},1]$, integers $1,2,3,\ldots$, booleans true, false. There is, however, a small difference between the expression syntax of BoogiePL and BoogiePLS. BoogiePL supports a shorthand for two dimensional arrays e.g. $\text{\$\text{Heap[this, Example.a]]}$ instead of $\text{\$\text{Heap[This][Example.a]}}$, BoogiePLS does not. BoogiePLS also allows the old operator, which already has been introduced. The most important difference of the expression language compared to an ordinary programming language is quantification. BoogiePLS allows universal and existential quantifications over expressions, see 3.12.

\[
e_1 ::= \vdots \\
| (\forall x_1 : \tau \cdot e_2) \\
| (\exists x_2 : \tau \cdot e_2) \\
\]

(3.12)

Note that any variable in old is not bound by a quantification and refers to the value of that variable outside that quantification. For example, in the following predicate $x > y \Rightarrow (\forall x : \text{int} \cdot \text{old}(x) < x)$, the old $(x)$ inside the quantification refers to the same $x$ as in the left-hand side of the implication.

### 3.3 Weakest Pre-condition

The second step in static testing is the linearization of the program, see section 3.5. However, in order to explain the necessity of linearization the weakest pre-condition calculation is presented first.

#### 3.3.1 Rules

**Preliminary**

Before the rules for the weakest pre-condition calculation can be defined, a representation for a predicate that is consistent with BoogiePLS is needed. Consider a function that returns the **Type** of an expression: *typeof : Expr → Type*. Using this
function a predicate can be defined as a subset of expressions ($\textbf{Predicate} \subseteq \textbf{Expr}$). Every member of the $\textbf{Predicate}$ set must be of type $\textbf{bool}$, so ($\forall e \in \textbf{Predicate} \bullet$ $\text{typeof} \; e = \textbf{bool}$) must hold. In favour of brevity we often write $e : \tau$ instead of $\text{typeof} \; e$, assuming that the $\text{typeof}$ function is applied to resolve the correct type of $\tau$.

While uncommon, it is syntactically possible to define multiple specifications for a procedure in BoogiePL, see the abstract syntax of BoogiePL in 3.11. Because it is awkward to work with multiple specifications, functions to join them into a single specification are defined, e.g. a the function $\text{pre}$ below that joins all pre-conditions into a single pre-condition (see definition 3.13). Note that the function $\text{mod}$ uses the join operator over sequences from (3.9). The functions $\text{pre}$ and $\text{post}$ replace unwanted specifications with the boolean expression $\text{true}$, but since $e \land \text{true} \equiv e$ the conjunctions are effectively ignored.

Furthermore, the function ($\text{spec}$) is assumed, which maps the name of a procedure to its respective sequence of specifications.

Procedure

Using the previously mentioned definitions the calculation of the weakest pre-condition of a procedure can be defined. It is modelled as a polymorphic function named $\text{wp}$. The body of a procedure returns an anonymous (lambda) function that transforms a post-condition into the weakest pre-condition for that body. This is also known as a predicate transformer. By applying the post-condition to the $\text{wp}$ of the body, see definition (3.15), a concrete pre-condition is obtained. However, this is not yet a correct weakest pre-condition, because it contains references to the old values of variables from the post-condition. Since the anonymous function has transformed the post-condition to a pre-condition, the old variables must also be transformed into “current” variables. This is done via a substitution which basically removes the old operator. There are, however, some caveats concerning substitutions, which shall be dealt with later.
Chapter 3

Using the \textit{wp} function for a procedure, a function that calculates the verification condition can be constructed, see (3.16).

\[ \text{vc : Procedure} \rightarrow \text{Predicate} \]
\[ \text{vc} \langle \text{procedure id} (x_n: \tau) \ S_n \{ \beta \} \rangle = \text{condition} \]

where
\[ \text{condition} = \text{pre} \ S_n \Rightarrow \text{wp} \langle \text{procedure id} ... \rangle \quad (3.16) \]

\section*{Body}

The \textit{wp} function for the body of a procedure is defined in definition 3.17. A body consists of a sequence of local variable declarations and a sequence of blocks. The variable declarations are optional, but at least one block is required. The responsibility of the \textit{wp} function for \textbf{Body} is to "direct" the calculation. Since the control flow between the blocks is dictated by the transfer of control commands, blocks can occur in arbitrary order without affecting it. A procedure always starts its execution at the first block \((B_1)\), the entry point.

Definition 3.17 contains two rules; one for a body without declarations, and one with declarations. The first rule only returns the weakest pre-condition of the first block, but the second rule must also handle the declarations. Recall that the weakest pre-condition must be calculated in reversed order. In order maintain this behaviour function composition is used. Consider that function composition is defined as, \((g \circ f) \ x = g \ (f \ x)\), which can be read as \(g\) following \(f\), or \(g\) composed with \(f\). By composing the declarations with \(B_1\), it is considered before the declarations. In principle the order in which the declarations themselves are composed is irrelevant, since the resulting predicates are logically equivalent for any order.

\[ \text{wp : Body} \rightarrow (\text{Predicate} \rightarrow \text{Predicate}) \]
\[ \text{wp} \langle [] [B_1 \ldots B_m] \rangle = \lambda q \rightarrow \text{wp} B_1 q \]
\[ \text{wp} \langle [D_1 \ldots D_n][B_1 \ldots B_m] \rangle = \lambda q \rightarrow (\text{wp} D_1 \circ \ldots \circ \text{wp} D_n \circ \text{wp} B_1) q \quad (3.17) \]

\section*{Local Variable Declaration}

The \textit{wp} rule for local variable declarations is presented in 3.18. Because BoogiePL does not assign a particular value to a variable when it is declared, it can have any value. Therefore, the declaration behaves much the same as a \texttt{havoc} command. This behaviour is represented in the weakest pre-condition with a universal quantification over the post-condition.

\[ \text{wp : Decl} \rightarrow (\text{Predicate} \rightarrow \text{Predicate}) \]
\[ \text{wp} \langle \text{var} [x_1: \tau, \ldots, x_n: \tau]; \rangle = \lambda q \rightarrow (\forall x_1: \tau, \ldots, x_n: \tau \bullet q) \quad (3.18) \]

\section*{Block}

In BoogiePL a block consists of at least one label and at least one transfer of control (toc) command, but normally also contains a sequence of command. A block without any commands returns the \textit{wp} function of its successors directly, while a block with commands presents a similar problem as the declarations in the previous paragraph. Again the \textit{wp} function must maintain the reverse order of the weakest pre-condition calculation. The solution is therefore similar to that of the body and
Approach

function composition is used to arrange the correct order.

\[
\begin{align*}
wp : \text{Block} & \to (\text{Predicate} \to \text{Predicate}) \\
wp (l : [\ ] T) &= \lambda q \to wp T q \\
wp (l : [C_1 \ldots C_n] T) &= \lambda q \to (wp C_1 \circ \ldots \circ wp C_n \circ wp T) q 
\end{align*}
\]

\[3.19\]

Command

Up until this point, apart from the \text{Decl} rule, most of the work done by the \text{wp} function has been administrative — directing the calculation of the weakest pre-condition. The \text{wp} rules for commands add more meaning to the predicate. They actually transform the post-condition of a command into the weakest pre-condition that is consistent with that command. Because the \text{call} and assignment command are complex, they are dealt with separately. Consider the rules for commands in \[3.20\], at the bottom the rule for the \text{havoc} command is defined. The \text{havoc} command basically “resets” the \text{xn} variables to an arbitrary value. Meaning that after this command the \text{xn} can have any value which is represented by an universal quantify over the \text{xn}. The \text{assume} and \text{assert} commands are pretty straight forward and respectively introduce an implication and a conjunction.

\[
\begin{align*}
wp : \text{Command} & \to (\text{Predicate} \to \text{Predicate}) \\
wp (x := e;) &= \lambda q \to q [e/x] \\
wp (a[e_1] := e_2;) &= \lambda q \to q [(\text{repby}(a, e_1, e_2))/a] \\
wp (\text{assert } e;) &= \lambda q \to e \land q \\
wp (\text{assume } e;) &= \lambda q \to e \Rightarrow q \\
wp (\text{havoc } [x_1, \ldots, x_n];) &= \lambda q \to (\forall [x_1 : \tau, \ldots, x_n : \tau] \bullet q)
\end{align*}
\]

\[3.20\]

Assignment

\text{BoogiePL}\_S has two kinds of assignment, one to a ordinary variable and one to an array element, see definition 3.20. The definition shows that both assignments induce a substitution on the post-condition. A substitution replaces all occurrences of the variable, which is being assigned to, with the expression that is assigned. The exact details of a substitution shall be discussed in the next section.

An assignment to an ordinary variable is relatively easy, e.g. an assignment \(x := 1\); replaces all occurrences of \(x\) in the post-condition with 1. Unfortunately, an assignment to an array element can not be solved with an ordinary substitution. Since the index of an array can be an expression it is possible to alias an array element. Given an array \(a\), a variable \(i = 1\), and another variable \(j = 1\), the expressions \(a[i]\) and \(a[j]\) refer to the same array element. Consider the following program:

\[
\{x > 0\} \ a[i] := x \ \{a[j] > 0\}
\]

and let us calculate the weakest pre-condition naively, with an ordinary substitution:

\[
\begin{align*}
wp(a[i] := x)(a[j] > 0) &= (a[j] > 0)[x/a[i]] \\
&= a[j] > 0
\end{align*}
\]

The resulting pre-condition is incorrect, because if \(i = j\) and \(x \leq 0\) hold, then \(a[j] > 0\) does not hold. Instead, the substitution should replace all array indices
that refer to the same array element. Unfortunately, in general, the value of the index expressions are unknown when encountering an assignment. A possible solution is to replace all occurrences of the array in the post-condition with a conditional expression which decides if the substitution affects that array element. This basically postpones the actual assignment and “pushes” it into the weakest pre-condition. Let us recalculate the weakest pre-condition using the conditional expression:

\[
\text{wp}(a[i] := x)(a[j] > 0) = (a[j] > 0)[(i = j \rightarrow x|a[j])/a[j]] = (i = j \rightarrow x|a[j]) > 0
\]

While the resulting pre-condition is correct for the example, it is not perfect. Firstly, the BoogiePL language does not support conditional expressions \((x \rightarrow y|z)\). Secondly, this substitution can only handle fully indexed arrays. For instance, if the example would have the following post-condition \(a = b\) instead of \(a[j] > 0\), the substitution would be incomplete: \([i =? \rightarrow x|a]/a\). Note that the equality and inequality operators are the only operators that can introduce an array without an index to the post-condition.

To solve these problems a similar approach as presented by Prasetya and Swierstra [24] is taken. They introduced a new array notation, \(a(i \text{ repby} e)\), which denotes a new array that has the exact same contents as the old array \(a\), except for the value at index \(i\), which is \(e\). However, the \text{repby} notation is not part of the BoogiePL expression language and it can not be added, because BoogiePLS must remain a pure subset of BoogiePL. Recall that BoogiePLS does not support the unimplemented functions of BoogiePL. In order to model the \text{repby} notation, an exception is made to this rule. The only allowed function application in BoogiePLS is that of the \text{repby} function. Consequently, the \text{repby} function becomes a reserved function name and may only be used by the \text{wp} function. The \text{repby}(a, i, e) function is equivalent to the \(a(i \text{ repby} e)\) notation, it denotes a function that returns a new array which has the exact same contents as the old array \(a\), except for the value at index \(i\) which is \(e\). Conceptually, the \text{repby} function postpones the assignment even longer and pushes it to the solver.

Using the \text{repby} function the \text{wp} rule for array assignments is defined, see definition 3.20. Conceptually, it replaces the assignment \(a[i] := x\) with the assignment \(a := \text{repby}(a, i, e)\). The \text{repby} function itself is not excluded from any substitutions, so nested \text{repby} functions in the weakest pre-condition are possible. Note that this rule also substitutes array variables without indexes, like in an equality relationship \((a = b)\).

The solver basically has two possibilities for solving the \text{repby} functions. Either the solver has an implementation of the \text{repby} function which symbolically manages the arrays. Or the \text{repby} functions are rewritten to the previously mentioned conditional expression. The latter is preferred, because it is more efficient. It implicitly demands that the expression language of the solver is richer compared to BoogiePL and supports the conditional expression. A \text{repby} function can be rewritten using definition 3.21.

\[
\text{repby}(a, i, e)[j] = (i = j) \rightarrow e|a[j]
\]

Fully indexed arrays can directly be rewritten, using definition 3.21. Array variables without an index, on the other hand, can not and need a special treatment. Only
equality relationships need to be considered, because they are the only expressions in a predicate that can introduce such variables. Taking into account that two arrays are equal if and only if all their elements are equal, definition 3.22 rewrites the equality relationships accordingly.

\[
\begin{align*}
(a = b) &= (\forall i \cdot a[i] = b[i]) \\
(a \neq b) &= (\forall i \cdot a[i] \neq b[i])
\end{align*}
\] (3.22)

By applying definition 3.22 and 3.21 subsequently, variables without an index can be solved:

\[
\begin{align*}
(repby(a, i, e) = b) &= \forall j \cdot repby(a, i, e)[j] = b[j]) \\
&= \forall j \cdot ((i = j) \rightarrow e[a[j]] = b[j])
\end{align*}
\]

Substitution

We distinguish between two kinds of substitutions, normal variable substitution (3.20) and old variable substitution (3.15). The substitutions have a similar notation, the \(e[x/\text{old}(x)]\) denotes a substitution of \(\text{old}(x)\) by \(x\) in expression \(e\), and \(e[y/x]\) denotes the substitution of variable \(x\) by \(y\) in expression \(e\). Both replace all occurrences of respectively \(\text{old}(x)\) and \(x\) with respectively \(x\) and \(y\). The difference is that \([/\] can only substitute \(\text{old}\) variables, while a \([/\] can substitute any variable but \(\text{old}\) variables. This behaviour defines the semantic of the \(\text{old}\) operator.

Take for example this small program \(x := y; \{x > \text{old}(x)\}\) for which the weakest pre-condition must be calculated. The assignment \(x := y\) will induce the following substitution \((x > \text{old}(x))[y/x]\). If the substitution were to replace all the occurrences of \(x\), the resulting weakest pre-condition would be \(y > \text{old}(y)\). Clearly this predicate is incorrect, because the assignment can only affect the current value of \(x\) not the old one. By excluding the \(\text{old}(x)\) from the replacements, as it should, the correct weakest pre-condition \(y > \text{old}(x)\) is obtained.

This is not the only caveat of variable substitution, both kinds of substitutions must prevent binding free variables. No variable of expression \(e_2\) must become bound in expression \(e_1\) in the substitution \(e_1[e_2/x]\). To illustrate this problem, consider the following \(wp\) calculation, which ignores this rule:

\[
\begin{align*}
wp(x := y; (\forall y : \text{int} \cdot y < x)) &= \forall y : \text{int} \cdot y < y \\
&= \forall y : \text{int} \cdot y < y
\end{align*}
\]

The original predicate states that \(x\) is maximal, but the resulting pre-condition does not state that \(y\) is maximal. The solution is to introduce fresh variables to \(e_1\) for the variables that are binding the free variables of \(e_2\). So in the above example variable \(y\) in \(e_1\) is replaced with the fresh variable \(y'\). Resulting in the correct weakest pre-condition, \((\forall y' : \text{int} \cdot y' < y)\), stating that \(y\) is maximal.
Chapter 3

Call

There are two possible strategies for calculating the weakest pre-condition of a call command, black box and white box reduction. For the first, the \( \text{wp} \) function considers the callee to be a black box and has no knowledge of the implementation of the callee. For the second, the \( \text{wp} \) function considers the callee to be a white box and has perfect knowledge of the implementation of the callee. The difference for the resulting weakest pre-condition is quite significant. The black box reduction basically replaces the call in the caller with the specification of the callee. An advantage is that, because the \( \text{wp} \) function does not use the body of the callee, it does not have to be complete. Also this solution strictly separates the procedures, inducing a strong isolation. A disadvantage is that every procedure must have a specification and that an incorrect specification propagates to the weakest pre-condition of the caller. The white box reduction, on the other hand, replaces the call by inlining the body of the callee. The advantage of this solution is that no specification is needed. Furthermore, it returns a precise and correct weakest pre-condition. A disadvantage is that the implementation of the callee must be finished. There is no separation of procedures and thus poor isolation. Considering that TestBPL follows the unit testing philosophy, which advocates strong isolation, the black box reduction strategy is chosen.

Consider the following procedure, which performs a multiplication of \( x \) and \( y \):

\[
\text{procedure} \quad \text{times} \ (x : \text{int}, \ y : \text{int}) \quad \text{returns} \ (z : \text{int})
\]

\[
\text{requires} \quad x \geq 0 \land y \geq 0; \\
\text{ensures} \quad z = x \times y;
\]

{ \ldots }

Now assume a call command, \( \text{call} \quad z := \text{times}(x, y) \). In favor of simplicity, the same formal and actual parameters for the call are used, but in general we need to take parameter passing into account. Conceptually, the black box reduction rewrites the call to:

\[
\begin{align*}
\text{assert} \quad x \geq 0 \land y \geq 0; \\
\text{call} \quad z := \text{times}(x, y) \\
\text{assume} \quad z = x \times y;
\end{align*}
\]

Which states that the pre-condition of the \text{times} procedure should hold before the call and that the post-condition must holds after the call. Formally, the same effect can be achieved via a single transformer, where \( r \) is the post-condition of the call:

\[
\lambda r \to \text{wp} (\text{assert} \quad x \geq 0 \land y \geq 0; \ \text{havoc} \ z; \ \text{assume} \ z = x \times y; )
\]

\[
= \lambda r \to x \geq 0 \land y \geq 0 \land (\forall z \bullet z = x \times y) \implies r)
\]

In the interest of readability the \( \text{wp} \) rule, for the call command, uses a slightly different notation. Consider definition 3.23, the procedure above the horizontal line represents the callee and below the line the \( \text{wp} \) function is defined. This notation is
used, because it allows to easily refer to the formal parameters of the callee.

\[
\text{procedure } id \ (x_n : \tau) \text{ returns } (y_n : \tau); \ldots
\]

\[
\text{wp :: Command } \rightarrow (\text{Predicate } \rightarrow \text{Predicate})
\]

\[
\text{wp}(\text{call } b_n := id(a_n); \} = \lambda r \rightarrow p' \land (\forall b_n : \tau, z_n : \tau \bullet q' \Rightarrow r)
\]

\[
\text{where } p = \text{pre (spec id)}
\]

\[
q = \text{post (spec id)}
\]

\[
p' = p \left[ a_n / x_n \right]
\]

\[
q' = q \left[ b_n / y_n \right]
\]

\[
\left[ a_n, z_n // \text{old}(x_n), \text{old}(z_n) \right]
\]

Keep in mind that the goal of the \text{wp} function for the call command is to transform the post-condition after the call to a weakest pre-condition before the call. To achieve this, it first needs to substitute the formal parameters in the specification of the callee to the actual parameters of the caller. The \( p' \) in definition 3.23 represents the pre-condition of the callee where this substitution was performed. Recall that the out-parameters of a procedure can not occur in its pre-condition, so only the in-parameters need to be substituted. For the post-condition, on the other hand, the \text{wp} function has to substitute both the in- and out-parameters. Furthermore, since the post-condition is transformed into a pre-condition, the \text{old} variables must become “current” variables. So the \text{old} variables are substituted by their respective “current” variables, e.g. for global variables \([z_n // \text{old}(z_n)]\). Because the in-parameters that occur in a post-condition are always \text{old} variables, the substitution of the formal and actual parameters can be combined with the substitution of \text{old} in-parameters, see the \([a_n / \text{old}(x_n)]\) substitution at the bottom. Note that this substitution is equivalent to the substitutions \([\text{old}(a_n)]//\text{old}(x_n)[/\text{old}(a_n)]\). Finally, the last variables that need to be transform are the “normal” variables. Just like the old variables their state changes, but instead of changing from past to present, they change from present to future. So they must be represented in the pre-condition as “new” variables, but since there is no \text{new} operator in Boogie\text{PL}_S fresh variable names are needed for them. However, a universal quantification over these variables has the same effect, with the added bonus that it is cheaper to perform automatically.

\text{Toc}

The final commands for which the \text{wp} function must be defined are the \text{Toc} commands, see definition 3.25. A \text{return} command signifies an end of a procedure and relinquishes control to its caller, but in the context of the \text{wp} function it is an entry point. Because the \text{return} command does not do anything else, the \text{wp} only needs to propagate the post-condition.

In Boogie\text{PL}_S every block has an unique label and the \text{goto} command requires a list of these labels. The \text{goto} command arbitrarily transfers control to one of the labels, or more precisely to its respective block. Therefore, before the \text{wp} function can be defined a function is needed to resolve a label to its block, see definition 3.24. The unimplemented \text{block} function is assumed to map a \text{Label} to its respective \text{Block}.

\[
\text{block :: Label } \rightarrow \text{Block}
\]

Given the fact that the \text{goto} command is non-deterministic, it is unknown which block is chosen. This means that it is the only command in Boogie\text{PL}_S to produce a set with multiple final states in the \text{exec} function in definition 3.2. Recall that
this simple machine model represents non-determinism by returning all possibilities, thus all possible program states. Since there is no way to know which member the machine will actually return, the only safe option is to assume that all members must be valid final states. Therefore, the $\text{wp}$ function is defined by the conjunction of the weakest pre-conditions of all the blocks, see 3.25.

\[
\text{wp} : \text{Toc} \to (\text{Predicate} \to \text{Predicate})
\]

\[
\begin{align*}
\text{wp} \langle \text{goto } [l_1, \ldots, l_n]; \rangle &= \lambda q \rightarrow \bigwedge_{i=1}^{n} \text{wp} (\text{block } l_i) q \\
\text{wp} \langle \text{return}; \rangle &= \lambda q \rightarrow q
\end{align*}
\] (3.25)

Unfortunately, the $\text{wp}$ rule for the $\text{goto}$ command, while theoretically correct, has a practical problem. It can be non-terminating, because its definition is circular. Take for example the following program:

\[
\text{Loop} : \\
\ldots; \\
\text{goto Loop, End;}
\]

It induces a infinite sequence of conjunctions of the $\text{Loop}$ block, so a weakest pre-condition can never be obtained. Non-termination in the $\text{wp}$ is unacceptable, because it must always generate the weakest-precondition regards of the termination or non-termination of a program.

### 3.3.2 Fixing Loops

Even when a programming language does not contain non-deterministic constructs, calculating the weakest pre-condition for a loop is a common problem in static verification. Let us assume, for the sake of argument, that the BoogiePL$_S$ language contains a structured $\text{while}$ loop, see definition 3.26.

\[
\{P\} \text{ while } g \text{ do } s \{Q\}
\] (3.26)

Despite the fact that the $\text{while}$ loop is deterministic, calculating its weakest pre-condition is still problematic, because it is unknown how many iterations the loop will make. Clearly this can not be deduce from its syntax, because the number of iterations depends on the semantics of the guard $g$ and the program state. Even when an initial program state is known, as with testing, it is still impossible to calculated the number of iteration of an arbitrary loop, because of the halting problem. The halting problem is a decision problem which states: given a program and a finite input, decide wheter or not the program terminates. If it were possible to determine the number of iterations of a loop, it would imply that it is possible to solve this problem. However, the halting problem is proven to be algorithmically undecidable by Alan Turing, which implies that calculating the number of iterations of a loop is also algorithmically undecidable.

One might get the idea that the weakest pre-condition does not exists in general. However, Sampaio [27] shows that it does exists. Consider a predicate as a complete lattice. In this context the $\text{wp}$ function induces a least fix point on this lattice, the weakest pre-condition. Unfortunately, the lattice can be of infinite size, so it might be impossible to reach the least fixed point.
**Approach**

**Lattice**

A partial order is a pair \((L, \sqsubseteq)\) where the set \(L\) is equipped with a partial ordering \(\sqsubseteq\), which is a reflexive, transitive, and antisymmetric relationship.

\[
\begin{align*}
(\forall x \in L \bullet x \sqsubseteq x) & \quad \text{reflexivity} \\
(\forall x, y, z \in L \bullet (x \sqsubseteq y) \land (y \sqsubseteq z) \Rightarrow (x \sqsubseteq z)) & \quad \text{transitivity} \\
(\forall x, y, z \in L \bullet (x \sqsubseteq y) \land (y \sqsubseteq x) \Rightarrow (x = y)) & \quad \text{antisymmetry}
\end{align*}
\]

A subset \(Y\) of \(L\) has \(x \in L\) as an upper bound if \((\forall y \in Y \bullet y \sqsubseteq x)\) and has \(x\) as a lower bound if \((\forall y \in Y \bullet x \sqsubseteq y)\). The \(x\) is an least upper bound of \(L\) if it is an upper bound and whenever \(x'\) is another upper bound of \(L\) and \(x \sqsubseteq x'\) holds. Similarly, \(x\) is the greatest lower bound of \(L\) whenever \(x'\) is another lower bound of \(L\) and \(x' \sqsubseteq x\) holds. An element \(\bot\) is the least element (or bottom) of \(L\) if \(\bot \sqsubseteq x\) for all \(x \in L\), and \(\top\) is the greatest element (or top) of \(L\) if \(x \sqsubseteq \top\) for all \(x \in L\). A complete lattice \(L\) (definition 3.27) is a partially ordered set \((L, \sqsubseteq)\) such that all subsets have least upper bounds \(\bigsqcup\) and greatest lower bounds \(\bigsqcap\) and \(L\) is equipped with a bottom and a top.

\[
L = (L, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top) \quad (3.27)
\]

An example of a complete lattice is the boolean set \{true, false\} when ordered by the implication operator and equipped with a disjunction for the least upper bound and conjunction for the greatest lower bound. Furthermore, the bottom element is \(false\) and the top element is \(true\). It is called the Bool lattice, see definition 3.28 where \(a, b \in \text{Bool}\).

\[
\begin{align*}
L &= \{true, false\} \\
a \sqsubseteq b &= a \Rightarrow b \\
a \sqcup b &= a \lor b \\
a \sqcap b &= a \land b \\
\bot &= false \\
\top &= true
\end{align*} \quad (3.28)
\]

The Bool lattice can be used to construct a lattice of predicates. In section 3.1.1 \(\Sigma\) was defined to represent the set of all possible states. Combined with the equality relationship the \(\Sigma\) induces a partial order \((\Sigma, =)\). This order is discrete, meaning that for all \(\sigma, \sigma' \in \Sigma\) if \(\sigma \sqsubseteq \sigma'\) then \(\sigma = \sigma'\). Using the partially ordered \(\Sigma\) and the complete lattice Bool a complete lattice of functions \(\{\Sigma \rightarrow \text{Bool}\}\) can be constructed. Since \(\Sigma\) is discrete, \(\{\Sigma \rightarrow \text{Bool}\}\) and \(\Sigma \rightarrow \text{Bool}\) are identical. The Predicate lattice can be defined by pointwise extension of the Bool lattice, see definition 3.29.

\[
L = \Sigma \rightarrow \text{Bool} \\
p \sqsubseteq q &= \forall \sigma \bullet p(\sigma) \Rightarrow q(\sigma) \\
p \sqcup q &= \lambda \sigma \bullet p(\sigma) \lor q(\sigma) \\
p \sqcap q &= \lambda \sigma \bullet p(\sigma) \land q(\sigma) \\
\bot &= \lambda \sigma \bullet false \\
\top &= \lambda \sigma \bullet true
\quad \text{where } \sigma \in \Sigma \text{ and } p, q \in \text{Predicate}
\]

(3.29)

Note that the Predicate lattice is a concrete representation of a predicate and that the previous Predicate representation is a symbolic one. Theoretically a Predicate can be given a state and it will output whether or not that state is valid, whereas
a **Predicate** needs a separate solver. Assume for the moment that the \texttt{wp} function is redefined to operate on the **Predicate** lattice, so \( \texttt{wp} : \texttt{Predicate} \rightarrow \texttt{Predicate} \). Because the \texttt{wp} function should be monotonic, it induces a least fixed point on the **Predicate** lattice. This least fixpoint represents the weakest pre-condition. A monotonic function is an order preserving function, e.g. let \( L_1 = (L_1, \sqsubseteq_1) \) and \( L_2 = (L_2, \sqsubseteq_2) \) be partially ordered sets then a function \( f : L_1 \rightarrow L_2 \) is monotone if:

\[
\forall l, l' \in L_1 \cdot l \sqsubseteq_1 l' \Rightarrow f(l) \sqsubseteq_2 f(l')
\]

Additional information on lattice theory can be found, for example, in [23].

While it might sometimes be possible to create a **Predicate** lattice for programs with small finite input spaces it is impossible to do so in general, because most programs have an infinite program state and thus an infinite **Predicate** lattice. Unfortunately, there is also no known lattice for a symbolic representation of predicates.

**Loop Reduction Strategy**

In theorem proving a conventional solution for the loop problem is the loop reduction strategy [24], which has some similarities to the weakest pre-condition reduction strategy. It is based on the pre-condition strengthening rule (definition 3.5), which states that a specification \( \{ P \} \ s \ \{ Q \} \) can be proven by proving a weaker specification \( \{ P' \} \ s \ \{ Q \} \). Using this idea we can also prove definition 3.26 by finding a weaker \( I \) such that:

\[
\{ I \} \text{ while } g \text{ do } s \ \{ Q \} \tag{3.30}
\]

But what kind of \( I \) would satisfy this specification? It is impossible to deduce the weakest pre-condition from the loop alone. Maybe we can come up with an \( I \) that is a little stronger than the weakest pre-condition, but still sufficiently weak enough to prove \( Q \).

When \( I \) holds at the end of every iteration, it will also hold at the end of the entire loop. So if \( I \) implies the post-condition \( Q \), the loop will realise \( Q \) when it ends. Additionally, the guard should hold at the beginning of every iteration, but should be invalid at the end of the loop. Furthermore, notice that if the \( I \) holds at the end of every iteration, it also holds at the beginning of every iteration (except for the first). Therefore it is sufficient to only prove that \( \{ I \} \ s \ \{ I \} \) holds. Basically the loop body must preserve the \( I \). More formally:

\[
\begin{align*}
\models I \land \neg g & \Rightarrow Q \\
\{I \land g\} & \ s \ \{I\} \\
\{I\} & \text{ while } g \text{ do } S(Q) \tag{3.31}
\end{align*}
\]

Definition 3.31 shows how to prove definition 3.30 when the loop terminates. Technically, in order to fully prove the loop it must also be shown that it terminates, but we will skip this. It should be evident that the correctness of definition 3.31 implies the correctness of definition 3.26. Because the predicate \( I \) is preserved by every iteration, it is called a **loop invariant**. Unfortunately, there is no algorithm to construct such invariants, so it has to be specified by the user, which is a major drawback. If the chosen invariant is too strong, the resulting pre-condition will also be too strong. When the specification turns out to be invalid, it is unclear whether the specification or the weakest pre-condition is to blame. However, if the chosen invariant is too weak, the resulting pre-condition will also be too weak and we are unable to prove that the program satisfies its post-condition.
Loops in Unstructured Programs

In BoogiePL there are no structured loops like the while loop. So the above loop reduction strategy can not be applied to BoogiePL directly. Fortunately Barnett and Leino present an alternative in their paper [4]. They introduce a weakest precondition calculus for a language derived from BoogiePL. It is worth noting that they basically handle the non-deterministic goto command the same as is done in definition 3.25. Consequently, they also have the same issues with loops. Loops in BoogiePL are implicit, as opposed to the explicit while loop example above. In order to find all loops in a program it must first be analysed, but for now this knowledge is assumed.

The explanation of Barnett and Leino’s solution requires some knowledge of graph theory. Readers who are unfamiliar with graph theory might first consider reading subsection “Graphs” in section 3.5.2 first. Barnett and Leino propose a transformation that converts a reducible control flow graph into an acyclic control flow graph, effectively removing the loop. They assume that any irreducible control flow graph is first made reducible. The graph must be reducible, because only then is it possible to identify a loop head for each loop. A loop can be identified via its unique back edge and the loop header is the target of this edge. However, a loop header may be associated with multiple loops. Each loop header and back edge pair is called a natural loop.

The basic idea is that for every natural loop the back edge is cut. This is similar to the removal of the while loop in the previous section. However, as with the while loop the body alone is not a sufficient representation of an arbitrary loop iteration. To solve this problem all variables that are updated in the loop body are collected. These variables are called loop targets. Next, all knowledge of the possible values these loop targets have is wiped out using the havoc command, which is introduced at the beginning of a loop head. Since no knowledge of the values of the loop targets means that they can have any value, the body becomes a sufficient representation of any iteration. Unfortunately, in this solution a lot of information and thus precision is lost. So much actually, that it is very likely to end up with a verification condition that is unprovable.

```
Start:
  goto LoopHead;
LoopHead:
  assert I;    // invariant
goto Body, After;
Body:
  assume g;    // guard
  s;          // loop body
goto LoopHead;
After:
  assume ¬g;   // inverse guard
  return;
```

Figure 3.4: Translated loop with invariant
Therefore, Barnett and Leino also assume that a loop is annotated with an invariant to provide extra information. Consider figure 3.4, it contains a BoogiePL listing and control flow graph of the translation of the following while loop with invariant:

\[
\text{while}(g) \text{ invariant } I \{ s \}
\]

Listing 3.4 shows that the translation asserted the invariant at the beginning of the loop header. Consequently, if the back edge is cut and the havoc inserted, the invariant might become impossible to validate (if it contains a loop target), because the **havoc** command is inserted before the invariant. To solve this problem Barnett and Leino first copy the invariant to all the predecessors of the loop header. This has the same effect as the preservation of the invariant in the \{I\} s \{I\} rule from the previous section. Furthermore, since the invariant now holds for any iteration it is also safe to **assume** that it holds after the havoc. Figure 3.5 shows the while loop in figure 3.4 after these transformations. The loop targets of the body \(s\) are represented with \(x_n\). Note that the invariant is copied to the predecessors of the loop header before the back edge is cut.

**Start:**
- **assert** \(I\); //check invariant  
  **goto** LoopHead;

**LoopHead:**
- **havoc** \(x_n\); //reset loop targets  
- **assume** \(I\); //invariant  
- **goto** Body, After;

**Body:**
- **assume** \(g\), \(s\); //body  
- **assert** \(I\); //check invariant  
- **return**; //removed back edge

**After:**
- **assume** \(\neg g\), \(s\);  
- **assert** \(I\);  
- **return**;

Figure 3.5: Eliminated loop

### 3.4 Program Instances

In the previous section two conventional solutions to the loop problem were introduced. Unfortunately, both require additional specifications in the form of loop invariants. This is a major drawback, because it places a large burden on the developer. However, the focus of this thesis is on static testing not on theorem proving and although using the same techniques it would contradict our goal. The goal of a
test-condition is to be more precise, with respect to the test-case compared to conventional verification-conditions. So instead of considering all possible executions of a program a test-case induces only one execution. Furthermore, it implies that only a particular number of iterations need to be considered, namely the number of iterations which is consistent with the corresponding test-case. Consequently, we only have to prove that it holds for a particular test-case instead of proving that the loop holds for every iteration.

3.4.1 General Idea

When a program is executed, it induces a specific path through a program, called a execution path. A single test-case in dynamic testing actually tests a specific execution path through a program. Suppose a program is constructed from this execution path, we call this a program instance. A program S can be viewed as a possibly infinite set of program instances. For example, consider the following program:

```
Start:
...;
  goto Then, Else;
Then:
  assume guard;
...;
  return;
Else:
  assume ¬guard;
...;
  return;
```

it has two possible execution paths and thus two possible program instances, namely:

```
Start:
...;
  goto Then;
Then:
  assume guard;
...;
  return;
```

```
Start:
...;
  goto Else;
Else:
  assume ¬guard;
...;
  return;
```

Refinement

A program instance s is a refinement of S, because its behaviour is a subset of the behaviour of the original program. We intentionally use the term subset to refer to our abstract machine model in definition 3.2. Using $\text{exec}$ the refinement can be defined more formally:

$$s \sqsubseteq S = (\forall \sigma \in \Sigma_S \bullet \text{exec } \sigma \ s \subseteq \text{exec } \sigma \ S) \quad (3.32)$$

The correctness of definition 3.32 is not formally proven, because we have not defined a full semantic for the $\text{exec}$ function nor do we want to. However, with just the rules for the non-deterministic goto and the assume command we can reason about its
correctness.

\[
\text{exec } \sigma \langle \text{goto } [l_1, \ldots, l_n]; \rangle = \bigcup_{i=1}^{n} \text{exec } \sigma \ l_i
\]

(3.33)

\[
\text{exec } \sigma \langle \text{assume } e; \rangle = \sigma \in e \rightarrow \{\sigma\} \ | \ \emptyset
\]

Because of the non-determinism of the goto command the \text{exec} function has to assume that all final states of the branches can be reached, hence the union of the branches. Now consider that the goto command is the only command in BoogiePL that can introduce a branch in a program and that the branches are the cause of multiple execution paths and thus program instances. Then clearly all final states that a program instance can produce are also in the set of final states of the original program. This covers all normal executions, but what about erroneous executions? Assume the program instances of an if-then-else program in the above example. One might expect an error in the “else” instance (the right one), when it is presented with a \( \sigma \) for which the guard holds. However, this program instance is not allowed to produce an error, because \text{goto} command effectively executes all branches with the same \( \sigma \). This means that in order for any program to execute successfully at all, all the branches of that program must be able to execute successfully with the same \( \sigma \). It is the responsibility of the translator to uphold this “law” and produces a valid BoogiePL program. In the case of the if-then-else example this is achieved by the way the guard is translated to the \text{assume} commands. The rule for \text{assume} in (3.33) shows then when the \( \sigma \) is not in the assumed predicate it produces an empty set instead of an error. So it can be concluded that, given a valid program \( S \), there does not exists a \( \sigma \) for which definition 3.32 does not hold.

Verifying all program instances will prove the correctness of the program, because their collective behaviour equals that of the program:

\[
\text{exec } \sigma \ S = \bigcup \left[ \text{exec } \sigma \ s_1, \ldots \right]
\]

(3.34)

However, there are typically infeasibly or even infinitely many program instances. So we limit to some of them. Note that software testing, in general, only tests the correctness of some program instances. For static testing this means that the weakest pre-condition has to be calculated for the selected program instances, instead of the weakest pre-condition for the entire program.

In the \text{exec} model the set of possible final states becomes smaller (stronger) for a program instance. However, a weakest pre-condition of a program instance becomes weaker than that of the original program. This may seem counter intuitive, but definition 3.25 shows that the goto command joins all possible choices through a logical conjunction. Since a program instance does not contain any branches there are no conjunctions, hence a weaker pre-condition. Because a program instance is a refinement of a program, the following equation holds:

\[
s \subseteq S = (\forall q \bullet \wp s q \Leftarrow \wp S q)
\]

(3.35)

Note that the conjunction of all the weakest pre-conditions of the program instances is equivalent to the weakest pre-condition of the program:

\[
\wp S q = \bigwedge [\wp s_1 q, \ldots]
\]

(3.36)

From definitions 3.36 and 3.35 it follows that if the verification condition \( p \Rightarrow \wp s q \) is invalid, then \( p \Rightarrow \wp S q \) will also be invalid. Meaning a program instance does not generate false positives. Or in other words, if an error is found in the program instance it means that same error is in the original program.
Linearization

Up until now a program instance has been an abstract notion, but in order to calculate the weakest pre-condition of a program instance we need a concrete program instance. We can model a program instance as a sequence of BoogiePL language constructs \([c_1, \ldots, c_n]\), where \(c\) is such a construct. The set of all possible sequences of \(S\) is defined as \(\text{Seq}_S\). We call the construction of such sequences linearization.

Which brings us back to the loop problem, in order to construct a sequence we need a way to linearize a loop. A linearized loop is the body of the loop which is copied \(x\) number of times. We can intuitively define it using a function \(\text{unwind}\), where \([l_m, \ldots, l_n]\) is a sequence of statements representing a single iteration of loop \(l\):

\[
\text{unwind } x [l_1, \ldots, l_n] = [l_1^1, \ldots, l_n^1, \ldots, l_1^x, \ldots, l_n^x]
\]

Note that this is not a complete definition of \(\text{unwind}\), we shall define that later on. This raises the question: how can we obtain a sequence that is consistent with the program instance we want to test? We certainly cannot execute the program and monitor it, we want static testing, after all. We can, however, statically construct a sequence. This entails two activities, namely the linearization of loops and branches. At first glance this may not seem very daunting, but linearizing a loop in BoogiePL is complex as we see later on. As for the linearization of branches — apart from the final solution — we shall not discuss them, since they do not pose a significant problem.

3.4.2 Program Transformation

A natural approach to linearization is program transformation. It creates program instances via source to source transformations. A transformation will linearize the branches and loops. However, finding and transforming the loop is complex, because BoogiePL does not have explicit loops. An alternative is to linearize the loops in the source language (Java, C#, C etc.) which usually have explicit loops. However, there are serious drawbacks to this approach:

- If a language is to be supported by our testing framework it — besides translating the source program to BoogiePL — has to implement linearization. This means that for each source language there has to be a separate implementation of linearization.
- Since we will not be building the translators, we lose control over the linearization process in our testing framework. Consequently, we do not know if all translators have the same notion of linearization and if their results can be trusted.
- Building a translator is a challenge in itself, it is cumbersome to also have to implement linearization. Besides, not everybody will build a translator with our testing framework in mind, because BoogiePL is not exclusive to our framework.

Another important drawback of any form of program transformation is its inefficiency. Even though the number of program instances that are considered is limited, all the sources have to be parsed, analyzed, and solved one by one. This is a costly approach, not to mention a hassle to manage.

Although, at first glance, the approach of program transformation is intuitive, its disadvantages are too great to be a viable solution. Therefore we opt for an alternative solution using abstract interpretation.
3.4.3 Abstract Interpretation

The underlying problem of the previous section is that it creates concrete program instances which are expensive to work with, no matter how they are obtained. So instead of constructing concrete program instances we propose to symbolically generate them. In order to do this we need an abstract model representing program instances. First off we label all the elementary language constructs of BoogiePLS with a natural number $\ell \in \mathbb{N}$ (see 3.37). The elementary language constructs of BoogiePLS are the local variable declarations and commands. The transfer of control commands are also labelled, but this is not strictly necessary as we shall see later on.

$$D ::= \langle \text{var } [x_1 : \tau, \ldots, x_n : \tau] \rangle^\ell$$

$$C ::= \langle x := e; \rangle^\ell$$

$$| \langle a[e_1] := e_2; \rangle^\ell$$

$$| \langle \text{assert } e; \rangle^\ell$$

$$| \langle \text{assume } e; \rangle^\ell$$

$$| \langle \text{havoc } [x_1, \ldots, x_n]; \rangle^\ell$$

$$| \langle \text{call } [x_1, \ldots, x_n] := \text{id}([e_1, \ldots, e_2]); \rangle^\ell$$

$$T ::= \langle \text{goto } [l_1, \ldots, l_n]; \rangle^\ell$$

$$| \langle \text{return}; \rangle^\ell$$

Instead of a sequence of concrete language constructs $[c_1, \ldots, c_n] \in \text{Seq}_S$, we construct a sequence of labels $[\ell_1, \ldots, \ell_n]$. We call this a program trace and define the set of all traces of $S$ as $\text{Trace}_S$, which naturally is a subset of $\text{Trace}$ ($\text{Trace}_S \subset \text{Trace}$). However, in the interest of brevity we drop the subscript $S$ unless it is not clear from the context which program we refer to.

We have cheated a little in definition 3.37, note that the call command only has a single label. If we wanted to properly model a call it needs two labels, like so:

$$\langle \text{call } [x_1, \ldots, x_n] := \text{id}([e_1, \ldots, e_2]); \rangle^\ell$$

$$\ell_c$$

$$\ell_r$$

where $\ell_c$ represents the call to the callee and $\ell_r$ the return to the caller. In a trace, between the $\ell_c$ and $\ell_r$ a trace of the callee would resides. Effectively this inlines the callee into the caller. However, recall that we have already stated in the definition of the wp function that we do not want to inline a procedure. Without the possibility of a trace of a callee between the $\ell_c$ and $\ell_r$ the two labels become superfluous and are reduced to a single $\ell$.

Syntactical vs. Semantical

In our previous definition of $\text{Seq}_S$ we were intentionally vague about which sequences exactly were in the set of possible program sequences. There are two distinguishable categories of sequences, namely the syntactical and semantical sequences. Syntactical sequences are consistent with the syntax of BoogiePLS and semantical sequences are consistent with the semantics (meaning) of the program. Note that the first is a superset of the second. Since the program traces model the program sequences, the same distinction holds.

Listing 3.6 contains a program, named $\text{times2}$, with labelling. The program itself is not very interesting; it multiplies its parameter $x$ by 2. The $\text{times}$ program to its left (listing 3.5) is a normal multiplication program. It is almost identical, except that the 2 has been parametrised, so it multiplies $x$ by $y$. From a semantical point of view these two programs are significantly different. The $\text{times2}$ program only
Approach

**Listing 3.5: times**

```plaintext
procedure times (x : int, y : int) returns (res : int)
    requires x ≥ 0 \& y ≥ 0;
    ensures res = old(x) \* old(y);
{
    var i : int;
    init:
        res := 0;
        i := 0;
    loop:
        goto loopTrue, loopFalse;
    loopTrue:
        assume i < y;
        res := res + x;
        i := i + 1;
    goto loop;
    loopFalse:
        assume i ≥ y;
    return;
}
```

**Listing 3.6: times2**

```plaintext
procedure times2 (x : int) returns (res : int)
    requires x ≥ 0;
    ensures res = old(x) \* 2;
{
    var i : int;
    init:
        res := 0;
        i := 0;
    loop:
        goto loopTrue, loopFalse;
    loopTrue:
        assume i < 2;
        res := res + x;
        i := i + 1;
    goto loop;
    loopFalse:
        assume i ≥ 2;
    return;
}
```

has one semantical trace, namely [1, 2, 3, 4, 5, 6, 7, 8, 4, 5, 6, 7, 8, 4, 9, 10]. While the times program has an infinite number of semantical traces, because its parameters range over an infinite domain — or at least the maximal number of int. However, from a syntactical point of view the programs are identical and both programs have infinite number of possible syntactical correct traces.

It should be clear that semantical traces are preverable to syntactical traces. Unfortunately, in general, they are impossible to construct statically, because of the halting problem. The halting problem applies, because we need to know the number of iterations of a loop in order to linearize it. For some programs, however, it is possible to statically determine this. Such as the times2 program, which clearly only iterates two times regardless of its input.

Because we can not generate semantical traces for all programs, this thesis focuses on the generation of syntactical traces. Note that generating semantical traces is basically an optimisation on generating syntactical traces, because it only filters out (or prevents) the generation of non-semantical traces for some programs.

### 3.5 Linearization

In this section we show how to generate all syntactical traces for a program, we call this process linearization. One task of the linearization process is to unwind all the loops in a program. Because BoogiePL does not contain explicit loops, we need to analyse the control flow of a program in order to find all the loops. We can not literally generate all syntactical traces, because (syntactically) all loops in BoogiePL are infinite. Instead, the developer must explicitly specify which iterations of the loop must be considered during linearization. To be able to analyse the control flow of a program, we construct an explicit model of it using the previously introduced labels.
3.5.1 Control Flow

We define a polymorphic function \( \text{flow} \) that constructs the control flow of a program in the form of a set of pairs \( (\mathcal{P}(N \times N)) \). Each pair represents the flow of one command to another. Remember that the abbreviation \( x^n \) is sometimes used to denote a sequence from \( x_1 \) to \( x_n \), and \( [] \) to denote the empty sequence. The \( \text{flow} \) equations use two polymorphic helper functions, namely \( \text{label} \) and \( \text{init} \). The \( \text{label} \) function returns the label of the elementary language constructs:

\[
\begin{align*}
\text{label} & : \text{Decl} \rightarrow N \\
\text{label} & : \text{Command} \rightarrow N \\
\text{label} & : \text{Toc} \rightarrow N \\
\text{label} \langle \ldots \rangle \ell & = \ell
\end{align*}
\]

(3.39)

While the \( \text{init} \) function returns the initial label of a particular language construct, but the \( \text{flow} \) function only uses it on blocks:

\[
\begin{align*}
\text{init} & : \text{Block} \rightarrow N \\
\text{init} \langle l : [] T \rangle & = \text{label } T \quad (3.40) \\
\text{init} \langle l : [C_1 \ldots C_n] T \rangle & = \text{label } C_1
\end{align*}
\]

Program

The control flow of a program is the union of all the control flows of all its procedures. Note that they are not connected, because we have chosen for black box reduction of the call command.

\[
\begin{align*}
\text{flow} & : \text{Program} \rightarrow \mathcal{P}(N \times N) \\
\text{flow} \langle [D_1, P_1, \ldots, D_n, P_m] \rangle & = \bigcup_{i=1}^{m} \text{flow } P_i \quad (3.41)
\end{align*}
\]

Procedure

The control flow of a procedure (3.42) is simply the control flow of its body \( \beta \).

\[
\begin{align*}
\text{flow} & : \text{Procedure} \rightarrow \mathcal{P}(N \times N) \\
\text{flow} \langle \text{procedure } id \{(x_n : \tau) S_n \} \{\beta}\rangle & = \text{flow } \beta \quad (3.42) \\
\text{flow} \langle \text{procedure } id \{(x_n : \tau) S_n \} \text{ returns } (y_n : \tau) S_n \} \{\beta}\rangle & = \text{flow } \beta
\end{align*}
\]

Body

If the body of a procedure only contains blocks, then its control flow is that of the first block (3.43), the entry point of the procedure. However, if the body also contains variable declarations we first tie all the subsequent variable declarations after each other. Next, we tie the last variable declaration to the first block and finally add the control flow of that first block.

\[
\begin{align*}
\text{flow} & : \text{Body} \rightarrow \mathcal{P}(N \times N) \\
\text{flow} \langle [[B_1 \ldots B_m]] \rangle & = \text{flow } B_1 \\
\text{flow} \langle [D_1 \ldots D_n][B_1 \ldots B_m] \rangle & = \bigcup_{i=1}^{n-1} \{(\text{label } D_i, \text{label } D_{i+1})\} \cup \{(\text{label } D_n, \text{label } B_1)\} \cup \text{flow } B_1 \quad (3.43)
\end{align*}
\]
**Approach**

**Block**

It is possible that a block only contains a toc command. In this case the control flow of a block (3.44) returns the control flow of that toc command. Usually the block will also contain other commands. In this case we first tie all the subsequent commands after each other. Then we tie the last command to the toc command and finally add the control flow of the toc command itself.

\[
\text{flow : Block} \rightarrow \mathcal{P}(\mathbb{N} \times \mathbb{N}) \\
\text{flow } (l : [T]) & \quad = \quad \text{flow } T \\
\text{flow } (l : [C_1 \ldots C_n] T) & \quad = \quad \bigcup_{i=1}^{n-1} \{(\text{label } C_i, \text{label } C_{i+1})\} \cup \{(\text{label } C_n, \text{label } T)\} \cup \text{flow } T
\]  

(3.44)

**Toc**

If the toc command is a return command the procedure stops, so we do not have to add anything to the control flow (3.45). However, if the toc command is a goto command we first translate the BoogiePL labels to their respective blocks, using the `block` function. Then we tie the label of the goto command (\(\ell\)) to all the initial labels of the blocks. Finally, we add the flow of all these blocks to the control flow.

\[
\text{flow : Toc} \rightarrow \mathcal{P}(\mathbb{N} \times \mathbb{N}) \\
\text{flow } \text{return}();^\ell & \quad = \quad \emptyset \\
\text{flow } \text{goto } [l_1, \ldots, l_n];^\ell & \quad = \quad \bigcup_{i=1}^{n} \{(\ell, (\text{init} \circ \text{block}) l_i)\} \cup \text{flow } \text{block} l_i
\]  

(3.45)

**Fixing Loops**

Unfortunately, the flow function for the goto command has the same termination issues as the wp function in section 3.3. It does not terminate when the goto command forms a direct or indirect loop, because of the recursive (flow \(\circ\) block) \(l_i\) call. Fortunately, we can fix this problem quite easily with a few modifications, see definitions 3.46 and 3.47.

\[
\text{flow : Toc} \rightarrow \mathcal{P}(\mathbb{N} \times \mathbb{N}) \\
\text{flow } \text{goto } [l_1, \ldots, l_n];^\ell & \quad = \quad \bigcup_{i=1}^{n} \{(\ell, (\text{init} \circ \text{block}) l_i)\}
\]  

(3.46)

Clearly by removing the (flow \(\circ\) block) \(l_i\) call completely we eliminate the circularity, but we also miss a substantial part of the control flow. The missing control flow can be added by modifying the flow function for the body. Instead of adding just the control flow of the first block, we now add the control flow of every block. Note that the new definition also adds dead blocks (if they exist), but this does not pose any problems.

\[
\text{flow : Body} \rightarrow \mathcal{P}(\mathbb{N} \times \mathbb{N}) \\
\text{flow } ([][B_1 \ldots B_m]) & \quad = \quad \text{blocks} \\
\text{flow } ([D_1 \ldots D_n][B_1 \ldots B_m]) & \quad = \quad \bigcup_{i=1}^{n-1} \{(\text{label } D_i, \text{label } D_{i+1})\} \cup \{(\text{label } D_n, \text{label } B_1)\} \cup \text{blocks} \\
\text{where } \text{blocks} & \quad = \quad \bigcup_{i=1}^{m} \text{flow } B_i
\]  

(3.47)
Also note that we can easily reverse the control flow using the flow\textsuperscript{R} equation (3.48).

\[
\text{flow}^\text{R}: \mathcal{P}(\mathbb{N} \times \mathbb{N}) \rightarrow \mathcal{P}(\mathbb{N} \times \mathbb{N}) \\
\text{flow}^\text{R} f = \{ (\ell, \ell') | (\ell', \ell) \in f \} \tag{3.48}
\]

Now that we have constructed an explicit control flow of the program, we need to linearize it. However, generating the traces from an unordered set of control flow pairs is complex. Luckily, the set of control flow pairs can be viewed as the edges of a directed graph.

### 3.5.2 Graphs

This section is strongly related to the design and implementation chapter, specifically section 4.2. This section provides a theoretical basis of graphs and their role in TestBPL, while the technical details will be explained in the implementation section.

A directed graph (or digraph) \( G \) is an ordered pair consisting of a nonempty set of vertices \( V \) and a set of edges \( E \):

\[
G = (V, E)
\]

#### Vertices and Edges

An edge is an ordered pair of vertices \((u, v)\), so the set of edges is a subset of the cartesian product of vertices, \( E \subseteq V \times V \). An edge is directed from \( u \) to \( v \), so vertex \( v \) is a direct successor of vertex \( u \) and vertex \( u \) a direct predecessor of vertex \( v \). The indegree of a vertex \( v \) is the number of direct predecessors of \( v \), denoted as \( \deg^- \), and the outdegree of a vertex \( v \) is the number of direct successors of \( v \), denoted as \( \deg^+ \). If an vertex has an indegree of 0 it is called a source and if it has an outdegree of 0 it is called a sink. Vertex \( u \) and \( v \) are called siblings if they share a same direct predecessor.

Below, some useful functions for graphs are defined. In favour of the readability of the function types, aliases are defined in definition 3.49. Note that because the control flow function in section 3.5.1 models the control flow using natural numbers, the vertices in the graph must also be represented with natural numbers.

\[
\begin{align*}
\text{Vertex} &= \mathbb{N} \\
\text{Edge} &= \text{Vertex} \times \text{Vertex} \\
\text{Graph} &= \mathcal{P}(\text{Vertex}) \times \mathcal{P}(\text{Edge})
\end{align*} \tag{3.49}
\]

Since the flow function essentially returns a set of edges, it is simple to define a function that turns that set into an actual graph, see definition 3.50. Because the edges are already given, the function only has to extract the vertices from the edges.

\[
\begin{align*}
\text{graph} : \mathcal{P}(\text{Edge}) &\rightarrow \text{Graph} \\
\text{graph edges} &= (\text{vertices}, \text{edges}) \\
\text{where} \\
\text{vertices} &= \bigcup \{\{u, v\} \mid (u, v) \in \text{edges}\}
\end{align*} \tag{3.50}
\]

Furthermore, functions that return the sources and sinks of a graph are also easily defined using the \( \deg \) notation, see definitions 3.51 and 3.52.

\[
\begin{align*}
\text{source} : \text{Graph} &\rightarrow \mathcal{P}(\text{Vertex}) \\
\text{source} (V, E) &= \{ v \mid v \in V, \ \deg^- v = 0 \} \tag{3.51}
\end{align*}
\]
**Approach**

sink : \textbf{Graph} \rightarrow \mathcal{P}(\textbf{Vertex})
\begin{equation}
\text{sink } (V, E) = \{ v \mid v \in V, \deg^+ v = 0 \} \quad (3.52)
\end{equation}

The edges of a graph can be classified with respect to their position and orientation in the graph. For example see figure 3.6, it shows back, front, and cross edges.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{classify_edges.png}
\caption{Classify Edges}
\end{figure}

category of the edges in figure 3.6 can be determined visually. All edges that are pointing downwards are classified as forward edges. Edges that are pointing upward are back edges and edges pointing from one “tree” to the other are known as cross edges. Functions returning the edges of a particular category are assumed, e.g. see definition 3.53. The details of this function are not given in this section, but its implementation shall be presented in section 4.2.

back : \textbf{Graph} \rightarrow \mathcal{P}(\textbf{Edge})
\begin{equation}
\text{back } (V, E) = \{ e \mid e \in E, e \text{ is back edge} \} \quad (3.53)
\end{equation}

**Paths and Cycles**

A path in $G$ is a sequence of vertices $[v_1, v_2, \ldots, v_n]$, such that each pair of adjacent vertices is in the set of edges, $(v_i, v_{i+1}) \in E$ for $1 \leq i < n$. Or in other words each pair of adjacent vertices in a path are connected. A type alias for paths is defined as:

\begin{equation}
\text{Path} = \overline{\text{Vertex}} \quad (3.54)
\end{equation}

If there exist a path between a vertex $u$ and another vertex $v$, $u$ is a predecessor of $v$ and $v$ is a successor of $u$. A path is elementary if it contains no vertex twice. A maximal path is a path that starts at a source and ends in a sink. Of course only graphs that have at least one source and sink can have maximal paths.

A circuit or cycle in $G$ is a path that starts and ends with the same vertex. A circuit is elementary if, apart from the first and last vertex, no vertex occurs twice. A directed graph without any circuits is called a directed acyclic graph or dag.

Assuming a function \textbf{path} : $\textbf{Graph} \rightarrow \mathcal{P}(\textbf{Path})$ which returns all possible paths through a graph, it is possible to specify functions that enumerate the elementary paths and circuits of a graph. In definition 3.55 the \textbf{epath} function is specified, which enumerates all elementary paths of a graph. It works as a filter on the \textbf{path} function leaving only elementary paths. For the definition it is assumed that the elements of a sequence $p$ can be accessed through an index, denoted as $p[i]$. Furthermore, the length of sequence $p$ is denoted as $\#p$.

epath : $\textbf{Graph} \rightarrow \mathcal{P}(\textbf{Path})$
\begin{equation}
\text{epath } g = \{ p \mid p \in \text{path } g, \forall i,j \bullet 0 \leq i < \#p \land i \neq j \Rightarrow p[i] \neq p[j] \} \quad (3.55)
\end{equation}
The same strategy can be used to specify a function enumerating all maximal elementary paths. The `emep` function, however, filters out any elementary path that does not start in a source and ends in a sink. Recall that the $\overline{v_n}$ notation allows use to access the vertices $v_1$ and $v_n$ directly.

\[
\text{emep : Graph } \rightarrow \mathcal{P}(\text{Path}) \\
\text{emep } g = \{ \overline{v_n} \mid \overline{v_n} \in \text{epath } g, v_1 \in \text{source } g \land \overline{v_n} \in \text{sink } g \} \tag{3.56}
\]

For the enumeration of elementary circuits of a graph all the back edges of a graph are needed since they introduce the circuits. The `eec` function also follows the generate and filter strategy, but with a little twist. An elementary circuit is basically an elementary path with its start vertex copied and attached behind it. Of course this is not true for all elementary paths, but only for those that are connected the their start vertex via a back edge. So the `eec` function construct its circuits from these elementary paths by adding the start vertex to the end.

\[
\text{eec : Graph } \rightarrow \mathcal{P}(\text{Path}) \\
\text{eec } g = \{ \overline{v_n} \sqcup [v_1] \mid \overline{v_n} \in \text{epath } g, (\omega, \alpha) \in \text{back } g \land \alpha = v_1 \land \omega = v_n \} \tag{3.57}
\]

### 3.5.3 Traces

The generation of the traces of a control flow graph is presented in this section. The basic principles are explained, but the implementation is presented in section section 4.3. Because the goal of TestBPL is to test procedures in isolation, a call command is not inlined by the `flow` function. This results in disjoint control flow graphs of procedures. While the `trace` function is designed for control flow graphs of single procedures, it should also work for a control flow graphs of procedures with inlined call commands.

Conceptually, a control flow graph always has a single source vertex, because a procedure always has a single entry point. Unfortunately, the `flow` function also generates the control flow of dead code. So the control flow of the procedure can have multiple sources. Because a procedure always contains a return command, its control flow graph will also always contain a sink vertex. Consequently, every control flow graph of a BoogiePL procedure always contains at least one maximal path. Note that a trace of a procedure is equivalent to a maximal path of its control flow graph. The generation of all traces is therefore equivalent to the enumeration of all maximal paths. Because a circuit can be repeated an infinite number of times in the path, it is impossible to enumerate all maximal paths.

Since a control flow graph is finite, the set of elementary paths and elementary circuits is also finite. By injecting an elementary circuit (ec) into a maximal elementary path (`mep`), any maximal path and thus trace can be constructed artificially. For example, consider the subgraph $\{1,2,3,4\}$ in figure 3.6. It contains a single maximal elementary path and a single elementary circuit, $[1,2,3,4]$ and $[2,3,2]$ respectively. Given a desired number of iterations of the circuit, a trace can be constructed. Before injecting a circuit, it is linearized using the `unwind` function in definition 3.58, e.g. for 2 iterations; `unwind` $2$ $[2,3,2] = [2,3,2,3,2]$. Then the linearized circuit is injected at the appropriate position in the path, which is the vertex corresponding with the head of the circuit. Such a position is called an injection point. In the running example, the linearized circuit replaces vertex 2 resulting in the following
Approach

\text{trace} \ [1, 2, 3, 2, 3, 2, 3, 4].

\text{unwind} : \mathbb{N} \to \text{Path} \to \text{Path}

\begin{align*}
\text{unwind } 0 \text{ circuit} & = [] \\
\text{unwind } 1 \text{ circuit} & = \text{circuit} \\
\text{unwind } m \ [v_1, \ldots, v_n] & = [v_1] \sqcup \text{cycles} \\
\text{where} \\
\text{cycles} & = \bigcup_{i=1}^{m} [v'_2, \ldots, v'_{n_i}]
\end{align*}

(3.58)

Constructing maximal paths for control flow graphs with a single circuit, like the previous example, is relatively easy. However, in practice, the control flow graph of a procedure often contains more than one circuit. More importantly, some of these circuits will be nested in others. Usually, a nested circuit can and should only be inserted in a path containing its parent circuit. Consider the graph in figure 3.7, it contains one mep \([1, 2, 3, 7]\) and two ecs: the outer \([2, 3, 4, 5, 6, 2]\) and inner \([4, 5, 4]\). Because the mep of the graph does not contain the outer circuit, the inner circuit can not be injected. The solution is to first inject the outer circuit and then inject the inner circuit, which introduces two complications. Firstly, the nesting of the circuits must be determined. Secondly, the result of an injection is a non-elementary path which can introduce multiple injection points for subsequent injections. Consider the graph in figure 3.8, it contains one mep \([1, 2, 3, 4, 5, 6]\) and two circuits: the outer \([2, 3, 4, 5, 2]\) and inner \([4, 3, 4]\). Assuming, that both circuits are linearized to 1 and only 1 iteration, so 0 iterations is not an option, the first injection should be:

\([2, 3, 4, 5, 2] \to [1, 2, 3, 4, 5, 6] = [1, 2, 3, 4, 5, 2, 3, 4, 5, 6]\)

The \(\to\) symbol is temporarily used to indicate an injection. Because the injection of the outer circuit introduced a second 4 in its path, the injection of the inner circuit \([4, 3, 4]\) must replace them both:

\([4, 3, 4] \to [1, 2, 3, 4, 5, 2, 3, 4, 5, 6] = [1, 2, 3, 4, 3, 4, 5, 2, 3, 4, 3, 4, 5, 6]\)

\begin{figure}[h]
\centering
\includegraphics[width=0.4\textwidth]{nested_circuits1.png}
\caption{Nested Circuits}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.4\textwidth]{nested_circuits2.png}
\caption{Nested Circuits 2}
\end{figure}

The function \text{product}, see definition 3.59, linearizes an elementary circuit and injects it into a path. Because it would be inefficient to repeat the entire process
for every desired iteration, the function accepts a set of iterations, the \( \text{its} \) parameter. The number of possible paths that can be constructed increase dramatically, because all combinations of iterations and injection points must be considered. For example, assume \( \text{path} \) to contain two injection points and \( \text{its} \) to be \( \{0, 1\} \), then the following combination of iterations and injection points can be made: \( \{(0, 0), (0, 1), (1, 0), (1, 1)\} \). The pairs in the set represent the injection points. So the first pair states that a circuit linearized to 0 must be inserted at the first and second injection point. Note that the set is the cartesian square of iterations, \( \text{its} \times \text{its} \).

In general the number of different possible injections in a \( \text{path} \) is the \( n \)-ary cartesian product of \( \text{its} \), where \( \text{its} \) is the desired set of iterations and \( n \) is the number of injection points. The \text{product} function uses helper functions and variables, but some of them are intentionally left undefined to hide unimportant details. The \( \text{inj}^n \) variable represents the \( n \)-ary cartesian product of \( \text{its} \). The \( \text{point} \) represent all injection points of a \( \text{path} \) and contains the position of every vertex, \( v_{\text{pos}} \), which equals the head of the circuit. Note that the \( n \) in the \( n \)-ary cartesian product of iterations is the length of the \( \text{point} \) sequence. Furthermore, the function \( \text{insert} \) replaces every injection point with its corresponding linearized circuit in \( \text{path} \). Because the function composition effectively starts its replacements at \( \text{point}_{n} \), \( \text{point}_1 \) through \( \text{point}_{n-1} \) remain the same for the resulting path and thus the subsequent replacements.

\[
\text{product} : \mathcal{P}(\mathbb{N}) \to \text{Path} \to \text{Path} \to \mathcal{P}(\text{Trace})
\]

\[
\text{product} \ \text{its} \ \text{ec} \ \text{path} = \begin{cases} \{\text{path}\} & \text{if } \text{inj}^n = \emptyset \\ \{\text{insert } \text{inj} \ \text{point}_n | \ \text{inj} \in \text{inj}^n\} & \text{otherwise} \end{cases}
\]

where

\[
\text{point}^n = \left[ v_{\text{pos}} | v \in \text{path}, v = \text{head} \ \text{ec} \right] \quad \text{(3.59)}
\]

\[
\text{inj}^n = \text{its}_1 \times \ldots \times \text{its}_n
\]

\[
\text{insert } \text{it}_n \ \text{point}^n = (\text{replace } \text{point}_i (\text{unwind } \text{it}_i \ \text{ec}) \circ \ldots \circ \text{replace } \text{point}_n (\text{unwind } \text{it}_n \ \text{ec})) \ \text{path}
\]

Finally, a function that generates all traces of a graph is defined in definition 3.60. Note that a lot of the details of the \text{trace} function are intentionally hidden. The full implementation and its details are presented in section 4.3. The \text{trace} function contains several undefined functions: \text{fold}, \text{nested}, and \text{id}. It is assumed that the \text{nested} function transforms the set of elementary circuits into a structure representing the nesting of the circuits, for instance a forest. Readers whom are familiar with Haskell will recognise the intention of the \text{fold} function — it represents a generic recursion over the nested structure. The parameters of \text{fold} are a function, a value for the base case, and the nested structure. The strategy of \text{trace} function is to compose a function which, given the set of maximal elementary paths, produces all the paths of a graph. So the \text{generate} function and the base case do not construct concrete values, but functions. The \text{generate} function constructs an anonymous function that uses the previously defined \text{product} function to generate all traces of the current circuit \( \text{ec} \). These paths are passed on to the inner circuits, which ensures that the inner circuits are injected.

\[
\text{trace} : \text{Graph} \to \mathcal{P}(\mathbb{N}) \to \mathcal{P}(\text{Trace})
\]

\[
\text{trace} \ \text{g} \ \text{its} = (\text{fold } \text{generate } (\text{nested } (\text{ec } \text{g}))) \ (\text{emep } \text{g})
\]

where

\[
\text{generate } \text{ec inner} = \lambda \text{paths} \cdot \text{inner} \quad \cup \{\text{product } \text{its} \ \text{ec} \ \text{path} | \ \text{path} \in \text{paths}\}
\]
3.6 Weakest Pre-condition Revisited

The linearization process removes the responsibility of directing the weakest pre-condition calculation to follow a reverse control flow from the \( \text{wp} \) function. Instead it explicitly constructs the order in which the calculation has to take place in the form of the traces. However, these traces use a symbolic notation, numerical labels, to represent a program instance. To be able to calculate the weakest pre-condition, the labels must be translated back to their respective language construct. In section 3.5.1 the \( \text{label} \) function was introduced to translate a language construct to its respective label. Now this function is “upgraded” to bijective function, meaning it can also translate a label to a language construct, see definition 3.61.

\begin{align*}
\text{label} : \text{Decl} & \leftrightarrow \mathbb{N} \\
\text{label} : \text{Command} & \leftrightarrow \mathbb{N} \\
\text{label} : \text{Toc} & \leftrightarrow \mathbb{N}
\end{align*} \tag{3.61}

Because the \( \text{wp} \) function is now separated from the control flow, the definitions from section 3.3 become obsolete, except for those in definition 3.62.

\begin{align*}
\text{wp} : \text{Decl} & \rightarrow (\text{Predicate} \rightarrow \text{Predicate}) \\
\text{wp} : \text{Command} & \rightarrow (\text{Predicate} \rightarrow \text{Predicate}) \\
\text{wp} : \text{Toc} & \rightarrow (\text{Predicate} \rightarrow \text{Predicate}) \\
\text{wp} (\text{goto} [l_1, \ldots, l_n]) & = \lambda q \rightarrow q \\
\text{wp} (\text{return};) & = \lambda q \rightarrow q
\end{align*} \tag{3.62}

Because the \( \text{wp} \) function for \text{Decl} and \text{Command} remain the same, they are not redefined in definition 3.62. In principle, the \( \text{wp} \) rules for \text{Toc} are also obsolete, but the transfer of control commands are still represented in the traces. Therefore, the \( \text{wp} \) rules for \text{Toc} remain, but are redefined to only propagate the post-condition.

Furthermore, a new function has to be defined which applies the \( \text{wp} \) function in the correct order. Given a post-condition, the traces, and the \( \text{label} \) function, the \text{calculate} function does exactly that. Function composition is used to ensure the weakest pre-condition is calculated in the correct order, like the \( \text{wp} \) rule for \text{Block}, see definition 3.19.

\text{calculate} : \text{Predicate} \rightarrow \mathcal{P}(\text{Trace}) \rightarrow \mathcal{P}(\text{Predicate})

\text{calculate \ post \ traces} \quad = \quad \{\text{build \ trace \ | \ trace \ \in \ \text{traces}}\}

\text{where}

\text{build} [\ell_1, \ldots, \ell_n] \quad = \quad (\text{wp} (\text{label} \ell_1) \circ \ldots \circ \text{wp} (\text{label} \ell_n)) \text{ post}

\tag{3.63}

The final step in the process is the construction of the test-conditions. A bijective function is assumed, which maps a linearized verification condition to its corresponding test data, see definition 3.64.

\text{testdata} : \text{Predicate} \leftrightarrow \mathcal{P}(\text{Var} \times \text{Expression}) \tag{3.64}

The \text{tc} function in definition 3.65 ties all the previous functions together and constructs the test-conditions for a procedure. The \text{pre}, \text{post}, \text{traces}, and \text{graph} helper variables should not need further explanation. Basically, the \text{tc} function only has to substitute the variables with its corresponding test data in the linearized verification conditions. These conditions are constructed in the \text{levcs} helper variable and
the sub helper function substitutes all the variables.

\[
\begin{align*}
tc : & \text{Procedure} \to \mathcal{P}(\mathbb{N}) \to \mathcal{P}(\text{Predicate}) \\
tc \langle \text{procedure } id \ (x_n : \tau) & \ S_n \ (\beta) \rangle \ its \\
tc \langle \text{procedure } id \ (x_n : \tau) \text{ returns } (y_n : \tau) & \ S_n \ (\beta) \rangle \ its \\
& = \{ \text{sub lvc (testdata lvc) } | \ lvc \in \text{lwcs}\} \\
\text{where} & \\
pre & = \text{pre } S_n \\
post & = \text{post } S_n \\
traces & = \text{trace graph its} \\
graph & = \text{graph (flow } \beta) \\
lwcs & = \{ \text{pre } \Rightarrow \ wp | \ wp \in \text{calculate post traces}\} \\
\text{sub lvc } \{(v_1, e_1), \ldots, (v_n, e_n)\} & = \text{lvc } [e_1/v_1] \ldots [e_n/v_n]
\end{align*}
\]

(3.65)

3.7 Example

In this section an elaborate example of the entire process described in this thesis is presented. The program used is the bubblesort algorithm for sorting an array. Of course before TestBPL can start generating test-conditions, the program has to be translated from the source language into BoogiePL. Up until now the example source languages included Java and C#, but the source language for this example is plain C. Because the translation from Java or C# has to cope with advanced features such as object orientation, the resulting BoogiePL program becomes complex. Since the C programming language is relatively simple, it is chosen as the source language for this example. Consequently, the translated BoogiePL program is less complex.

The original C bubblesort program is presented in listing 3.7. Since no translator for C to BoogiePL currently exists, the translation was performed manually. The result is presented in listing 3.8. Different translations are possible, but this one was largely derived from the translation of a Java bubblesort program. This “template” translation was performed by the BML2BPL[17] translator, which is able to translate Java bytecode to BoogiePL. However, to reduce complexity the arrays in the C variant were directly translated to arrays in BoogiePL as opposed to a symbolic representation on the heap. Further liberties were taken in order to make the example more readable. Since the C programming language does not have a built-in specification language the one from BoogiePL is used. Also in order to construct a meaningful pre-condition the sizeof function of the C language was allowed in the specification. However, TestBPL does not support unimplemented functions yet, so only a part of the pre-condition is copied to the BoogiePL version.
Approach

// requires sizeof (input)/sizeof (int) = length ∧ length > 1;
// ensures (∀ i:int • i ≥ 0 ∧ i < old(length) ⇒ result[i−1] ≤ result[i]);

int * bubblesort(int input[], int length) {
  int i, j, temp;
  for (i = length - 1; i > 0; i--)
    for (j = 0; j < i; j++)
      if (input[j] > input[j+1])
        temp = input[j];
        input[j] = input[j+1];
        input[j+1] = temp;

  }
  return input;
}

Listing 3.7: Bubblesort C

Despite the measure describe above, the BoogiePL bubblesort program in listing 3.8 shows that even a relative small program, such as the bubblesort program, translates to a large BoogiePL equivalent.

procedure bubblesort (input : [int]int, length : int) returns (output : [int]int)
  requires length > 1;
  ensures (∀ i:int • i ≥ 0 ∧ i < old(length-1) ⇒ output[i] ≤ output[i+1]);
{
  var i : int, j : int, temp : int; [1]

  init :
    i := 0; [2]
    j := 0; [3]
    temp := 0; [4]
    goto outerPre; [5]

  outerPre :
    i := length - 1; [6]
    goto outerLoop; [7]

  outerLoop :
    goto outerLoopTrue, outerLoopFalse; [8]

  outerLoopTrue :
    assume i > 0; [9]
    goto outerBody; [10]

  outerLoopFalse :
    assume i ≤ 0; [11]
    goto exit; [12]

  outerBody :
    goto innerPre; [13]

  outerPost :
    i := i - 1; [14]
    goto outerLoop; [15]

  innerPre :
    j := 0; [16]
    goto innerLoop; [17]

  innerLoop :
    goto innerLoopTrue, innerLoopFalse; [18]

  innerLoopTrue :
    assume j < i; [19]
goto innerBody; [20]
innerLoopFalse:
assume j ≥ i; [21]
goto outerPost; [22]
innerBody:
goto bigger; [23]
innerPost:
j := j + 1; [24]
goto innerLoop; [25]
bigger:
goto biggerTrue, biggerFalse; [26]
biggerTrue:
assume input\[j\] > input\[j + 1\]; [27]
temp := input\[j\]; [28]
input\[j\] := input\[j + 1\]; [29]
input\[j + 1\] := temp; [30]
goto innerPost; [31]
biggerFalse:
assume input\[j\] ≤ input\[j + 1\]; [32]
goto innerPost; [33]
exit:
output := input; [34]
return; [35]
}

Listing 3.8: Bubblesort BPL

The labels which are assigned to all elementary language constructs of BoogiePL, see section 3.4.3, are presented in listing 3.8 between the [ ]. As described in section 3.5.1, the next step is the calculation of the control flow using the flow function. Its result is omitted, instead the graph to which it is converted to is presented in figure 3.9.

Using the emep and eec functions from section 3.5.2 the elementary paths and elementary circuits are ascertained. The graph in figure 3.9 contains a single maximal elementary path

\[1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 33, 34\]

and three elementary circuits:

1. \[8, 9, 10, 13, 16, 17, 18, 21, 22, 14, 15, 8\]
2. \[18, 19, 20, 23, 26, 32, 33, 24, 25, 18\]
3. \[18, 19, 20, 23, 26, 27, 28, 29, 30, 31, 24, 25, 18\]

As described in section 3.5.3, besides the elementary maximal paths and elementary circuits, a set of allowed number of iterations must be provided in order to generate the traces of the graph. To keep the example manageable, it is assumed that the loops may either make a single iteration or none at all. Despite the fact that TestBPL allows an user to control the number of iterations, it is still susceptible to the computational complexity of the program under test. For example, when generating the traces of bubblesort for 0 and 1 iteration 4 traces are construct, but for 0 through 5 iterations 18656 traces are constructed. This dramatic increase is a consequence of the computational complexity of the bubblesort algorithm, which

52
Figure 3.9: Bubblesort Control Flow Graph
is $O(n^2)$. As such, the user must take great care when choosing the number of iterations. When applying the trace function from section 3.5.3 to the graph in figure 3.9 and the set of \{0,1\} iterations it returns four traces:

1. $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 13, 16, 17, 18, 19, 20, 23, 26, 27, 28, 29, 30, 31, 24, 25, 18, 21, 22, 14, 15, 8, 11, 12, 34, 35]$

2. $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 13, 16, 17, 18, 19, 20, 23, 26, 32, 33, 24, 25, 18, 21, 22, 14, 15, 8, 11, 12, 34, 35]$

3. $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 13, 16, 17, 18, 21, 22, 14, 15, 8, 11, 12, 34, 35]$

4. $[1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 34, 35]$

Note that trace 3 is semantically incorrect, it suggest a program instance where the inner loop is never executed and the outer loop is executed once. It is not hard to see that the initialisation and guards of the loops do not allow this, especially in the C version.

Using the `calculate` function from section 3.6 the weakest pre-conditions of the corresponding traces are constructed:

1. $(\forall i: \text{int}, j: \text{int}, \text{temp}\text{xint} \bullet \text{length} - 1 > 0$
   \[\Rightarrow 0 < \text{length} - 1 \Rightarrow \text{input}[0] > \text{input}[0+1] \Rightarrow 0+1 \geq \text{length} - 1\]
   \[\Rightarrow (\text{length} - 1) - 1 \leq 0 \Rightarrow (\forall i: \text{int} \bullet i \geq 0 \land i < \text{length} - 1\]
   \[\Rightarrow \text{repby}(\text{repby}(\text{input}, 0, \text{input}[0+1]), 0+1, \text{input}[0])[i]
   \[\leq \text{repby}(\text{repby}(\text{input}, 0, \text{input}[0+1]), 0+1, \text{input}[0][i+1])\]

2. $(\forall i: \text{int}, j: \text{int}, \text{temp}\text{xint} \bullet \text{length} - 1 > 0$
   \[\Rightarrow 0 < \text{length} - 1 \Rightarrow \text{input}[0] \leq \text{input}[0+1] \Rightarrow 0+1 \geq \text{length} - 1\]
   \[\Rightarrow (\text{length} - 1) - 1 \leq 0 \Rightarrow (\forall i: \text{int} \bullet i \geq 0 \land i < \text{length} - 1\]
   \[\Rightarrow \text{input}[i] \leq \text{input}[i+1])\]

3. $(\forall i: \text{int}, j: \text{int}, \text{temp}\text{xint} \bullet \text{length} - 1 > 0$
   \[\Rightarrow 0 \geq \text{length} - 1 \Rightarrow (\text{length} - 1) - 1 \leq 0\]
   \[\Rightarrow (\forall i: \text{int} \bullet i \geq 0 \land i < \text{length} - 1 \Rightarrow \text{input}[i] \leq \text{input}[i+1])\]

4. $(\forall i: \text{int}, j: \text{int}, \text{temp}\text{xint} \bullet \text{length} - 1 \leq 0$
   \[\Rightarrow (\forall i: \text{int} \bullet i \geq 0 \land i < \text{length} - 1 \Rightarrow \text{input}[i] \leq \text{input}[i+1])\]

The weakest pre-condition corresponding to trace 3 supports the statement that it is semantically incorrect, because of the contradicting condition $\text{length} - 1 > 0 \Rightarrow 0 \geq \text{length} - 1$.

The final step in the process is the construction and verification of the test-conditions. Despite the fact that the prototype TestBPL can not verify the test-conditions, one trace is manually verified below. Since every trace and thus weakest pre-condition is consistent only with a specific program instance, not just any test-data can be used. Taking the program instance corresponding to each trace into account, the test-data should have following characteristics:

- Trace 1 can be tested with any unsorted array containing exactly two elements.
- Trace 2 can be tested with any sorted array containing exactly two elements.
- Trace 3 is incorrect
• Trace 4 can be tested with any array containing exactly one element.

For example, let us verify trace 1 using the following test-data: \( \text{input} = [4, 3] \) and \( \text{length} = 2 \). First the test-condition is constructed using the \( tc \) function:

\[
2 > 1 \Rightarrow (\forall i: \text{int}, j: \text{int}, \text{temp: int} \cdot 2 - 1 > 0
\]
\[
\Rightarrow 0 < 2 - 1 \Rightarrow 4 > 3 \Rightarrow 0 + 1 \geq 2 - 1
\]
\[
\Rightarrow (2 - 1) - 1 \leq 0 \Rightarrow (\forall i: \text{int} \cdot i \geq 0 \wedge i < 2 - 1
\]
\[
\Rightarrow \text{repby}(\text{repby}(\text{input}, 0, 3), 0 + 1, 4)[i]
\]
\[
\leq \text{repby}(\text{repby}(\text{input}, 0, 3), 0 + 1, 4)[i + 1])
\]

In favor of readability all arithmetic calculations are evaluated and the outermost quantification is removed, because it does not contribute anything to this particular predicate:

\[
2 > 1 \Rightarrow 1 > 0 \Rightarrow 0 < 1 \Rightarrow 4 > 3 \Rightarrow 1 \geq 1 \Rightarrow 0 \leq 0
\]
\[
\Rightarrow (\forall i: \text{int} \cdot i \geq 0 \wedge i < 1 \Rightarrow \text{repby}(\text{repby}(\text{input}, 0, 3), 1, 4)[i]
\]
\[
\leq \text{repby}(\text{repby}(\text{input}, 0, 3), 1, 4)[i + 1])
\]

Since the logical implication is right associative, the remaining universal quantification is solved first. This can be achieved by exhaustively rewriting its body for all the allowed \( i \). In this case \( i \) can only be 0, so the \( i \) is substitute by 0 and the quantification is removed. By applying the rewrite rule for \( \text{repby} \), see definition 3.21, the predicate can be verified:

\[
\text{repby}(\text{repby}(\text{input}, 0, 3), 1, 4)[0] \leq \text{repby}(\text{repby}(\text{input}, 0, 3), 1, 4)[1]
\]
\[
= \text{repby}(\text{repby}(\text{input}, 0, 3), 1, 4)[0] \leq (1 = 1) \rightarrow 4 \mid \text{repby}(\text{input}, 0, 3)[1]
\]
\[
= \text{repby}(\text{repby}(\text{input}, 0, 3), 1, 4)[0] \leq 4
\]
\[
= (1 = 0) \rightarrow 4 \mid \text{repby}(\text{input}, 0, 3)[0] \leq 4
\]
\[
= \text{repby}(\text{input}, 0, 3)[0] \leq 4
\]
\[
= (0 = 0) \rightarrow 3 \mid \text{input}[0] \leq 4
\]
\[
= 3 \leq 4
\]

Which leaves us with the following predicate:

\[
2 > 1 \Rightarrow 1 > 0 \Rightarrow 0 < 1 \Rightarrow 4 > 3 \Rightarrow 1 \geq 1 \Rightarrow 0 \leq 0 \Rightarrow 3 \leq 4
\]

It should be clear that this predicate is valid, meaning that the program performs as expected for a unsorted list with two elements.
Chapter 4

Design and Implementation

TestBPL is implemented in the purely functional programming language Haskell and depends heavily on the Haskell Utrecht Tools (HUT)[30]. Because presenting the entire implementation in this chapter would be a bit verbose, only the most essential and interesting functions are discussed. For more information about acquiring and using the source code of TestBPL see appendix A.

This chapter contains three sections. Section 4.1 describes the parser, attributes, etc. In the next section, section 4.2, the graph library which is used in TestBPL is presented. The library is based on an existing library and extended with some custom algorithms. The final section, section 4.3.2, presents how TestBPL generates the traces of a program, which is a critical step in the linearization process.

4.1 Overview

The Haskell Utrecht Tools is a collection of packages:

1. Parser Combinators: the parser combinator library is an embedded domain specific language for defining parsers in Haskell. A combinator is a higher order function which is used as an infix operator. Using these combinators as a parsing technique allows a developer to define a parser much like a grammar, but still have access to the expressiveness and features of the host language. Other common parsing techniques such as parser generators lack this ability.

2. Attribute Grammar System: the Utrecht University Attribute Grammar System (UUAG) is a preprocessor for Haskell. It enables a developer to easily write catamorphisms for arbitrary data types. A well known catamorphisms in Haskell is the foldr on lists. The language of the UUAG system allows a developer to define a catamorphism using the concepts of inherited and synthesized attributes. The UUAG system and parser combinators integrate very well and allow you to write attributes for the abstract syntax tree of program.

3. Pretty Printing Combinators: the pretty print library is an embedded domain specific language for defining a pretty printer for an abstract syntax tree.

For more information about the Haskell Utrecht Tools see [31, 29].

During the thesis project a BoogiePL parser was build using the parser combinators. This parser can parse full BoogiePL not just the BoogiePL$_S$ introduced in section
3.2. We chose to build a parser ourselves, because we very much liked to implement our static test framework using Haskell and the UUAG system. Besides, at the beginning of this thesis project there was no open source BoogiePL parser available.

A lot of the functions that were defined in chapter 3 are actually implemented as attributes, such as the block, wp, label, init, and flow functions.

4.2 Graph Library

The linearization process in TestBPL requires an implementation of two graph algorithms, specifically the enumeration of all maximal elementary paths and the enumeration of all elementary circuits of a graph, see section 3.5. In lazy functional programming languages, such as Haskell, graph algorithms have been a challenge to program efficiently. King and Launchbury [14] introduced an experimental graph library on which TestBPL’s graph library and above mentioned algorithms are based. The algorithms require some functions that were not exported by the original graph library, so the library was copied and extended. The basic concept of King and Launchbury’s work is presented in this section, along with the extensions made for TestBPL.

4.2.1 Representing Graphs

A directed graph is represented using a standard Haskell immutable array and is defined as a table of successors. The array’s index is a vertex and its value is a list of vertices that are connected to it. The vertex itself is represented by an integer.

```haskell
type Vertex = Int

type Table a = Array Vertex a

type Graph = Table [Vertex]
```

Haskell arrays come with some built-in functions, such as an indexing (!) operator, a function that returns all indices, and a function that returns the least and greatest bounds of an array. In the context of the Graph type the indices function effectively returns the vertices of the graph.

```haskell
vertices :: Graph → [Vertex]
vertices = indices
```

Also, a function that returns the edges of a graph can be defined.

```haskell

type Edge = (Vertex, Vertex)

edges :: Graph → [Edge]
edges g = [(v, w) | v ← vertices g, w ← g ! v ]
```

Before any algorithm can be applied to a graph, it must first be built. Because a graph is just a type alias for an array, it can be constructed using the same functions as an array. Nonetheless, a function is defined that conveniently transforms a list of edges into a graph. A consequence of using an array-based data structure is that the bounds of an array are often needed, e.g. for array construction.

```haskell

type Bounds = (Vertex, Vertex)

buildG :: Bounds → [Edge] → Graph
buildG bnds es = accumArray (flip ( : )) [] bnds es
```
For example, the following listing builds the graph in figure 4.1:

```plaintext
graph = buildG (1,10) [ (1,2), (2,3), (3,1), (3,4), (5,4), (2,5), (1,6), (8,6), (7,8), (7,10), (8,9), (8,10) ]
```

### 4.2.2 Depth-First Search

Graph algorithms can roughly be divided into two categories, those based on a depth-first search and those based on a breadth-first search of a graph. Both traversals systematically explore a graph, but do so in a different order. Depth-first search first explores its direct successors before exploring its siblings, while breadth-first search first explores its siblings before exploring its direct successors. Breadth-first search is not presented in detail, because the algorithms presented in this thesis are based on depth-first search. A depth-first traversal can be roughly described as:

**DFS initial vertex**

where

```plaintext
DFS v = do
  mark v as explored;
  ∀s ∈ successors of v ∧ s is unexplored do DFS(s);
```

Typically, a graph algorithm “embeds” its own code into the search algorithm. Efficiency is the big advantage of this approach, but there are also disadvantages: duplication of the search algorithm and entangled code. If we view the depth-first search process of a directed graph as an explicit value, it forms a spanning forest. The forest represents the order in which the depth-first search explores the vertices of a graph. For example, see the graph in figure 4.1 and its respective spanning forest in figure 4.2. Note that each source vertex in a directed graph leads to a separate tree in the forest. A undirected graph, however, always forms a spanning tree.

![Figure 4.1: Example Graph](image1)

![Figure 4.2: Spanning Forest](image2)

![Figure 4.3: Generated Tree](image3)

**Depth-First Forest**

King and Launchbury’s depth-first search generates such a spanning forest which they call a *depth-first forest*. Their graph algorithms operate on the forest instead of being embedded into the depth-first search. A major advantage of this approach is the separation of the depth-first search code and specific graph algorithm code.
Also, the depth-first forest can be reused by different algorithms and as such only has to be computed once. In strict programming languages the same approach would be inefficient, because not all graph algorithms need to explore the entire graph. For example, an algorithm that searches for a specific vertex can stop when it finds it, but the entire depth-first graph would be built nonetheless. However, in a lazy language like Haskell a value is only constructed when it is needed, so only the part of the depth-first forest that is required is built. Therefore, conceptually no penalty is paid in terms of efficiency. However, while Haskell may not explore the entire graph, the usual overhead of laziness still has to be considered. In their work King and Launchbury also took some measurements of a strongly connected components algorithm using the depth-first search, which seemed to indicate that the depth-first search is linear. When they compared it to Tarjan’s biconnected components algorithm in C, it was between 10 and 20 times slower. Keep in mind that this was back in 1993 and that Haskell compilers have improved significantly.

The depth-first forest is represented using the default Data.Tree module from Haskell, which introduces the following types:

```haskell
data Tree a = Node a (Forest a)
type Forest a = [Tree a]
```

Conventionally, depending on the graph algorithm, the depth-first search can consider a vertex before or after its successors. In order to mirror this behaviour the depth-first forest must be traversed in preorder or postorder, respectively. The preorder and postorder traversals are defined as two mutually recursive functions each. They to return an explicit value, a list of vertices in the desired order.

```haskell
preorder :: Tree a -> [a]
preorder (Node a ts) = a : preorderF ts

preorderF :: Forest a -> [a]
preorderF ts = concat (map preorder ts)

postorder :: Tree a -> [a]
postorder (Node a ts) = postorderF ts ++ [a]

postorderF :: Forest a -> [a]
postorderF ts = concat (map postorder ts)
```

### Generate and Prune

The construction of the depth-first forest uses a common strategy in lazy functional programming, namely the generate and prune strategy. This strategy first generates an output that is usually too big, potentially even infinite, and then prunes the output to discard any unwanted parts. Generate and prune exploits laziness to prevent the generation of discarded data. A very simple example of this strategy is the following Haskell expression `take 10 [1..]`, where `[1..]` generates an infinite list starting at 1 and `take 10` is the pruning function. The result of the expression is of course a list ranging from 1 to 10.

In order to define the depth-first search of a graph $g$ a generate function is defined that generates a tree rooted in $v$ containing all vertices of $g$ that are reachable from $v$. Unless the graph already is a tree, the generated tree will contain repeated subtrees. If the tree contains a cycle, the generated tree will be infinite.
Design and Implementation

generate :: Graph → Vertex → Tree Vertex

\[
generate \ g \ v = \text{Node } v (\text{map } \generate \ g \ (g!v))
\]

For example, for the graph in figure 4.1, starting from vertex 1 the function generates the tree in figure 4.3. In this figure the dotted edge and vertex represent the infinite repetition of subtrees starting from vertex 1.

The next step is to prune the tree, turning it into a depth-first tree. The goal is to discard any subtrees whose roots have occurred previously. Note that, depending on the pruning strategy, the generated tree can not only be turned into a depth-first spanning tree, but also in a breadth-first spanning tree. In order to keep track of the vertices that are to be discarded, a set of “marks” is maintained. This is done using a set monad called SetM. In favour of readability, the internal details of the monad are hidden. There are, however, three functions the pruning requires. Firstly, the typical run function that “executes” the monad. Secondly, the contains function that checks if a vertex is a member of the of set. Thirdly, the include function that adds a vertex to the set.

\[
\text{newtype } \text{SetM s a } = \text{SetM} \{ \text{runSetM ::... } \}
\]

\[
\text{run} :: \text{SetM s a } → a
\]

\[
\text{contains} :: \text{Vertex } → \text{SetM s } \text{Bool}
\]

\[
\text{include} :: \text{Vertex } → \text{SetM s } ()
\]

The prune function “runs” the chop function that literally chops off unwanted subtrees. It does so by simply not adding unwanted subtrees to the result. If a vertex was already visited it and its forest is not returned. If it was unvisited the vertex is added to the visited set and it and its forest are added to the result. Note that the ts are the children of the vertex and that the us are its siblings.

\[
\text{prune} :: \text{Forest Vertex } → \text{Forest Vertex}
\]

\[
\text{prune ts } = \text{run } (\text{chop ts})
\]

\[
\text{chop} :: \text{Forest Vertex } → \text{SetM s } (\text{Forest Vertex})
\]

\[
\text{chop } [] = \text{return } []
\]

\[
\text{chop } (\text{Node } v \ ts : \text{us}) = \text{do} \text{visited } ← \text{contains } v \\
\text{if } \text{visited then} \text{chop us} \\text{else do} \text{include } v \\
\text{as } ← \text{chop ts} \text{bs } ← \text{chop us} \text{return } (\text{Node } v \text{ as } : \text{bs})
\]

Depth-First Search

Now that most of the work is done by the generate and prune functions, the depth-first search only needs to glue them together. As determined in figures 4.1 and 4.2 a directed graph can have multiple source vertices inducing multiple depth-first trees. Therefore, besides the graph the dfs function needs a list of “starting” vertices, preferably the source vertices. On the other hand, not every graph has source vertices and not every graph algorithm starts in one if they do. Sometimes it is not important at all, so another function is defined called dff. It assumes that any vertex can be a start vertex, assuring that every vertex is in the resulting forest.
dfs :: Graph → [Vertex] → Forest Vertex
dfs g vs = prune (map (generate g) vs)

dff :: Graph → Forest Vertex
dff g = dfs g (vertices g)

4.2.3 Graph Algorithms

The extensions which were made to the original graph library are presented in this section. Both the enumeration of elementary circuits and the enumeration of elementary maximal paths can be implemented using the generate and prune strategy. Recall that an elementary path is a path in a graph in which no vertex occurs more than once. The path is also maximal if it starts in a source and ends in a sink. An elementary circuit is a elementary path, but with the same start and end vertex. Consequently, if a function can be defined which generates all elementary paths of a graph, it is possible to prune the result of this function and obtain all maximal elementary paths. Using the same function it is also possible to construct all elementary circuits, which effectively are elementary paths of which the start vertex are copied to the end.

Generate Elementary Paths

In order to generate the maximal elementary paths and elementary circuits, it is not required to actually generate all elementary paths. It is sufficient to generate only elementary paths from specific start and end vertices. The function that is responsible for the generation of elementary paths is called rgep, which is an abbreviation for; reversed generation of elementary paths. Meaning that the elementary paths returned by the function are in reverse. The function generates elementary paths that begin with any of the specified starting vertices.

rgep :: Graph → [Vertex] → [[Vertex]]
rgep g starts = foldr build (map (\v → [v]) starts) postord
where build v eps = [w : ep | ep ← eps, w ← g ! v
  , head ep == v
  , not $ w ’elem’ ep
  ] ++ eps
postord = postorderF (dfs g starts)

At the heart of the rgep function is the build helper function. The build function takes two parameters; the vertex currently under consideration v, and all reversed elementary paths constructed so far eps. Conceptually, the function creates new elementary paths from the existing ones that end with v. For every successor of v, the existing paths are copied and extended with the successor unless it already occurred in the path. Since it is cheaper to both add and read from the front of a list instead of from the back, the paths are build in reverse so the end of a path is at the front. Because the build function can only extend elementary paths for the current vertex if its direct predecessors were considered first, it must be scheduled accordingly. In order to achieve the correct scheduling the build function is applied in topological order of the graph. This can be constructed by inverting a postorder traversal of the depth-first forest, see [14]. So by performing a foldr over the postorder traversal the rgep function effectively builds elementary paths in topological order. Because the build function needs an initial set of elementary paths to extend, the starting vertices are converted to the initial elementary paths.
Using the \texttt{rgep} function it is easy to define a function that enumerates all elementary paths of a graph, see \textit{eep}. By providing all vertices of the graph as possible starting vertices all elementary paths are generated. Of course the paths must also be reversed.

\begin{verbatim}
eep :: Graph \to [[Vertex]]
eep g = map reverse \$ rgep g (vertices g)
\end{verbatim}

While the starting vertices can be specified using the \texttt{rgep} function, the end vertices cannot. So another function, \texttt{gep}, is defined which does allow the specification of end vertices. It uses the \texttt{rgep} function to generate all elementary paths from the start vertices and prunes out the ones that do not end in the specified vertices. This pruning is done on the reversed paths, again in favour of efficiency. Again, the paths are also reversed.

\begin{verbatim}
gep :: Graph \to [Vertex] \to [Vertex] \to [[Vertex]]
gep g starts ends = map reverse \$ prune (rgep g starts)
where prune = filter (\ep \to (head ep) `elem` ends)
\end{verbatim}

Maximal Elementary Paths

As stated before, a maximal elementary path is an elementary path from a source to a sink. So in order to be able to generate all maximal elementary paths of a graph, its sources and sinks must be identified first. Recall that a source is a vertex with an indegree of 0 and a sink a vertex with an outdegree of 0, see section 3.5.2. Because both tasks are very similar, a general function \textit{leaf} is defined. Given the degree of all vertices, either in- or outdegree, the \textit{leaf} function returns the sources or sinks, respectively. The \textit{source} and \textit{sink} functions are merely aliases of the \textit{leaf} function providing the correct degree. The \textit{leaf} function itself filters out the vertices with an in- or outdegree of 0. The \texttt{indegree} and \texttt{outdegree} are library functions and are not presented in this thesis.

\begin{verbatim}
leaf :: Table Int \to [Vertex]
leaf degree g =
  fst . unzip $ filter (\(v, count\) \to count == 0) $ assocs (degree g)

sink :: Graph \to [Vertex]
sink = leaf . outdegree

source :: Graph \to [Vertex]
source = leaf . indegree
\end{verbatim}

Using the \textit{source}, \textit{sink} and \textit{gep} functions the enumeration of all maximal elementary paths is trivial:

\begin{verbatim}
emep :: Graph \to [[Vertex]]
emep g = gep g (source g) (sink g)
\end{verbatim}

Elementary Circuits

The enumeration of all elementary circuits is a little more complex. Recall that back edges are the source of circuits in a graph and that a single back edge introduces one or more circuits to a graph. Therefore, in order to enumerate all circuits, all back edges of a graph must first be identified. Luckily, the original graph library
already contains a function that does exactly that, the `back` function. Given a
graph it basically filters out any edge which is not a back edge, returning a graph
with only back edges.

```haskell
back :: Graph \rightarrow \text{Table Int} \rightarrow \text{Graph}
back g post = mapT select g
  where select v ws = [ w | w \leftarrow ws, post ! v < post ! w ]
```

Besides the graph the `back` function also expects a table representing the postorder
traversal of the depth-first forest. The table assigns a number to the vertices which
corresponds to their position in the postorder traversal. The `back` function identifies
back edges using this table. Suppose an edge \((a, b)\) is an arbitrary edge in a graph
\(g\). Normally in the postorder traversal the source \((a)\) of the edge appears after the
target \((b)\) of the edge. Except when \((a, b)\) is a back edge, then the source appears
before the target. This relation identifies back edges. For example, see the graph
in figure 4.1 and its depth-first spanning forest in figure 4.2. The corresponding
postorder traversal table of this spanning forest can be found in table 4.1.

<table>
<thead>
<tr>
<th>Vertex</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>10</td>
<td>9</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 4.1: Postorder Traversal

Consider the edge \((1, 2)\) of the graph in figure 4.1. The postorder traversal table
show that the position of the vertex 1 is bigger than the position of vertex 2, so it
is not a back edge. This is true for all the edges of the graph in figure 4.2 except for
one. The edge \((3, 2)\) is a back edge, because the position of vertex 3 in the postorder
traversal is 2 which is smaller than the position of vertex 1 which is 6, meaning that
the source appears before the target.

Once all the back edges have been identified, all the circuits of every back edge can
be constructed. In principle a circuit does not have a start or end, but in order to
describe the `ecc` algorithm the source of a back edge is called the end and its target
is called the start of a circuit. So in the above example the 3 is the end vertex and
the 1 is the start vertex of all the circuits of the back edge \((3, 1)\). Note that all the
circuits of a back edge can be constructed by generating all elementary paths from
the start vertex to the end vertex and adding the start vertex to the end of the
resulting elementary paths. This is exactly what the `ecc` function does, for every
back edge it uses the existing `gep` function to generate all elementary paths for the
start to the end vertex and adds the start vertex in the end.

```haskell
ecc :: Graph \rightarrow [[\text{Vertex}]]
ecc g = concatMap (\((u,v)\) \rightarrow circuits v u) (edges backG)
  where circuits start end =
        map (\(c \rightarrow c ++ [\text{start}]\) \# \$ \text{gep} g [\text{start}] [\text{end}]
        backG = back g (postArr (bounds g) (diff g))
```

### 4.3 Generating Traces

In section 3.5.1 the `flow` equations were presented. Their implementation shall
not be presented, because there is no significant differences between them besides
the implementation language. However, some introduction of the involved types is
required. The label with which a program is annotated, see 3.4.3, is represented by
the type `Lab` and the resulting control flow is represented by the `Flow` type.
Design and Implementation

```haskell
type Lab = Int
type Flow = [(Lab, Lab)]
```

Of course the control flow must be converted to a graph and while the buildG function is presented in section 4.2.1 a more convenient function can be defined. Because a graph is represented with an array the buildG function requires a bounds. The bnds function extracts these bounds from the control flow, by finding the smallest and largest vertex. The toGraph function combines the bnds and buildG function to convert the control flow to a graph.

```haskell
bnds :: Flow → (Vertex, Vertex)
bnds fs = (head ordered, last ordered)
  where ordered = (sort . (λ(x, y) → x ++ y) . unzip) fs

toGraph :: Flow → Graph
toGraph [] = error "..."
toGraph edges = buildG (bnds edges) edges
```

4.3.1 Nested Loops

The eec function in section 4.2.3 enumerates all elementary circuits of a control flow graph, but unfortunately it does not provide any knowledge of the nesting of these circuits. This information is required in order to generate correct traces. This raises two questions:

1. Given two elementary circuits, how do we determine if one is nested in the other?
2. Given all elementary circuits of a graph, how do we represent their nesting?

The first question is answered by the compare function. Given two elementary circuits ec1 and ec2, the function returns an Ordering, which is used to represent the nesting. If ec1 is nested in ec2, ec1 is smaller (LT) then ec2. If ec1 is a parent of ec2, so ec2 is a nested circuit of ec1, then ec1 is bigger (GT) then ec2. Finally, if the circuits are equal, it means that they were not nested. In order to determine the nesting, the compare function needs to know the positions of the circuits in the graph. More specifically, it needs to know the source and target position of the back edge of the circuit. These positions can be determined using the topological ordering of the graph.

```haskell
compare :: [Vertex] → [Vertex] → [Vertex] → Ordering
compare topsort ec1 ec2 = order (start ec1, end ec1) (start ec2, end ec2)
  where topArr = tabulate (minimum topsort, maximum topsort) topsort
    start ec = topArr ! head ec
    end ec = topArr ! (last . init) ec
    order (ph1, pl1) (ph2, pl2)
        | ph1 > ph2 && pl1 < pl2 = LT
        | ph1 > ph2 && pl1 > pl2 = EQ
        | ph1 > ph2 && pl1 == pl2 = LT
        | ph1 < ph2 && pl1 < pl2 = EQ
        | ph1 < ph2 && pl1 > pl2 = GT
        | ph1 < ph2 && pl1 == pl2 = GT
        | ph1 == ph2 && pl1 < pl2 = LT
        | ph1 == ph2 && pl1 > pl2 = GT
        | ph1 == ph2 && pl1 == pl2 = EQ
```

65
Consider the graphs A, B, and C in figure 4.4. These graphs show the three most basic examples of nested circuits. Graph A shows the most obvious nested circuit, which starts after and end before its parent circuit. The first LT and GT in the order function match these circuits, depending of course on the order in which the circuits are presented to compare. Graph B and C show nested circuits that start or end in the same vertex as their parent circuit, which match the remaining LTs and GTs. That leaves the EQ ordering, which is shown in graphs D and E of figure 4.4. Graph E show two unrelated circuits, that both the start and end before or after the other.

Graph D shows a unstructured control flow graph which can be constructed in BoogiePL, because of the goto command. However, since the program will commonly be translated from a structured programming language, its corresponding BoogiePL program will also be structured. Therefore, it is assumed that TestBPL will not be presented with such control flow graphs. Consequently no action is taken to handle them. In the off case that an unstructured control flow is presented, no guarantee is given over the validity of the generated traces.

The next step is to represent the nesting of the elementary circuits. This is done using the previously introduced Forest data type. The basic idea is that the list of elementary circuits is converted to a forest, where a nested circuit is a subtree of its parent circuit. The nested function takes care of this. Because the list of elementary circuits (ecs) is presumed to be in arbitrary order, it is first sorted. The sorting is done using the compare function and ensures that a nested circuit always comes before its parent circuit. Because a nested circuit can have more than one parent, the nested function uses a foldr on the sorted list to construct the forest. This ensure that when a nested circuit is added all its parents are already in the forest.

\[
\text{nested} :: [\text{Vertex}] \rightarrow [[\text{Vertex}]] \rightarrow \text{Forest} [\text{Vertex}] \\
\text{nested topsort ecs} = \text{foldr add} [] \text{ sorted} \\
\text{where} \\
\text{add ec forest} \\
| \text{null forest} = [\text{Node ec}] \\
| \text{isOuter ec forest} = \text{Node ec} [] : \text{forest} \\
| \text{otherwise} =
\]
Design and Implementation

let cons = ( : )
    nil = []
    node root nested
        | isChild ec root = Node root (Node ec [] : nested)
        | otherwise = Node root nested
    in foldForest (cons, nil, node) forest

sorted = sortBy (compare topsort) ecs
isChild ec root =
    compare topsort ec root == LT && head ec 'elem' root
isOuter ec forest =
    all (\t \rightarrow compare topsort ec (rootLabel t) == EQ) forest

When adding a circuit to the forest there are three situations that can occur. Firstly, the forest can be empty and the current circuit is the first to be added. Note that because the list is sorted, the first circuit will always be a “outermost” circuit. Secondly, the current circuit can be an outermost circuit and the forest is not empty. This is determined by the isOuter function which checks if the current circuit is not nested in any existing tree. If it is not, the circuit is added to the top level trees. Finally, if the current circuit is not an outermost circuit, it must be a nested circuit and it must be added somewhere in the forest. In order to do find the correct positions, a recursive function is needed. Because the Data.Tree module does not contain a generic fold for the Forest data type, one is defined:

type ForestAlg a b c = (a → b → b, b, c → b → a)
foldForest :: ForestAlg a b c -> [Tree c] -> b
foldForest alg@(cons, nil, _) trees =
    foldr cons nil (map (foldTree alg) trees)
foldTree :: ForestAlg a b c -> Tree c -> a
foldTree alg@(node, node) (Node root forest) =
    node root (foldForest alg forest)

Using the fold and the isChild function the current circuit is added to all its parent circuit(s).

Considering the graphs A to E in figure 4.4, the right-hand side of the conjunction in the isChild function might seem superfluous. However, the compare function can not distinguish between a circuit that is directly or indirectly nested. For instance see the graph in figure 4.5, the compare function will state that the innermost circuit is nested in the outermost circuit. While this is correct, it should only be added to its direct parent, the middle circuit. Therefore, the extra check is required.

Furthermore, a branch introduces multiple elementary circuits from a single back edge, see the graph in figure 4.6. For the outer back edge these circuits are [2, 3, 4, 6, 7, 2] and [2, 3, 5, 6, 7, 2] and for the inner back edge these are [3, 4, 6, 3] and [3, 5, 6, 3]. Note that both inner circuits are nested in both outer circuits, so they are added to the forest under both circuits.

4.3.2 Generate Paths

With the forest of elementary circuits and the foldForest function, the generation of traces can be implemented as it was described in section 3.5.3. First an equivalent for the unwind function, see definition 3.58, is needed:
One of the details that was intentionally left undefined in section 3.5.3 was the replace function. Its implementation splits the path \( s \) at the given injection point \( i \) and adds the circuit \( r \) in the middle. In order to prevent duplication of the vertex at the injection point, only the tail of the second half of the split is taken.

\[
\text{replace} :: \text{Int} \rightarrow [a] \rightarrow [a] \\
\text{replace} \ i \ r \ s = (\text{fst} \ \text{split}) ++ r ++ (\text{tail} \ . \ \text{snd} \ $ \ \text{split})
\]

where \( \text{split} = (\text{splitAt} \ i \ s) \)

Next, the equivalent of the \textit{product} is implemented in the \textit{pathProduct} function. Note that the \( n \)-ary cartesian product is defined using the \textit{sequence} monad.

\[
\text{pathProduct} :: [\text{Int}] \rightarrow [\text{Vertex}] \rightarrow [\text{Vertex}] \rightarrow S.\text{Set} [\text{Vertex}]
\]

\[
\text{pathProduct} \ \text{its} \ \text{ec} \ \text{path} = S.\text{fromList} $ \ \text{map} \ \text{insert} \ \text{injections}
\]

where \( \text{insert} \ \text{injection} = \text{foldr} \ f \ \text{path} $ \ \text{zip} \ \text{points} \ \text{injection} \)

\[
\begin{align*}
\text{f} (\_ , 0) & \text{ path} = \text{path} \\
\text{f} (ix , it) \text{ path} & = \text{replace} \ ix \ (\text{unwind} \ \text{it} \ \text{ec}) \ \text{path} \\
\text{points} & = \text{elemIndices} (\text{head} \ \text{ec}) \ \text{path} \\
\text{injections} & = \text{sequence} $ \ \text{replicate} (\text{length} \ \text{points}) \ \text{its}
\end{align*}
\]

The \textit{generateTraces} function corresponds to the \textit{trace} function, see (3.60). In the implementation the role of the fold which was assumed to exist by \textit{trace} is actually fulfilled by the \textit{foldForest} function. The \textit{generateTraces} function uses this catamorphism to traverse the forest of nested circuits and generate the traces.

\[
\text{generateTraces} :: \text{Graph} \rightarrow [\text{Int}] \rightarrow S.\text{Set} [\text{Vertex}]
\]

\[
\text{generateTraces} \ \text{g} \ \text{its}
\]

| \( \text{mep} == [] \) = \( S.\text{empty} \) |
| \( \text{nec} == [] \) = \( S.\text{fromList} \ \text{mep} \) |
| \( \text{otherwise} \) = |
| \( \text{let cons hd tl} = \backslash \text{paths} \rightarrow (\text{hd} \ \text{paths}) \ 'S.\text{union}' (\ \text{tl} \ \text{paths}) \) |
Design and Implementation

\[ \text{node ec forest } = \]
\[ \text{\ backquote \ paths \ \rightarrow \ \text{forest} \ \text{\&} \ S.\text{fold} \ (\text{combine ec}) \ S.\text{empty paths} \]
\[ \text{\&} \ S.\text{union} \ (\text{pathProduct its ec path}) \]
\[ \text{\ where \ mep } = \ \text{emep} \ \text{g} \]
\[ \text{nec } \ = \ \text{nested} \ (\text{topSort g}) \ \text{eec g} \]

Finally, the implementation of the trace function only needs to perform some conversions.

\[ \text{trace} \ :: \ \text{Flow} \rightarrow \text{[Int]} \rightarrow \text{[[Vertex]]} \]
\[ \text{trace flow ns } = \ S.\text{toList} \ \text{\&} \ \text{generateTraces (toGraph flow) ns} \]

4.4 Optimisation

In section 4.2.2 it was explained how the depth-first search generates a depth-first forest. An advantage of this approach is that the forest can be shared between different graph algorithms. However, in section 4.2.3 the eec and emep functions, directly or indirectly, performed their on depth-first search. So when the generateTraces function applies these functions the depth-first search is executed twice. The functions were defined this way in favour of readability, but in the actual implementation of TestBPL a optimisation is performed. In the implementation the functions which are defined in 4.2.3 have a slightly different name. They all have a prime behind it, but this is not the only difference. None of the functions perform a depth-first search, instead it is parameterised. Because all algorithms require a postorder traversal of the depth-first forest, the traversal was parameterised as opposed to the depth-first forest. Since the rest of the function stays the same, it is not presented again.

\[ \text{rgep} \ :: \ \text{Graph} \rightarrow \text{[Vertex]} \rightarrow \text{[Vertex]} \rightarrow \text{[[Vertex]]} \]
\[ \text{rgep g postord starts } = \ ... \]

\[ \text{gep} \ :: \ \text{Graph} \rightarrow \text{[Vertex]} \rightarrow \text{[Vertex]} \rightarrow \text{[Vertex]} \rightarrow \text{[[Vertex]]} \]
\[ \text{gep g postord starts ends } = \ ... \]

\[ \text{emep} \ :: \ \text{Graph} \rightarrow \text{[Vertex]} \rightarrow \text{[[Vertex]]} \]
\[ \text{emep g postord } = \ ... \]

\[ \text{eec} \ :: \ \text{Graph} \rightarrow \text{[Vertex]} \rightarrow \text{[[Vertex]]} \]
\[ \text{eec g postord } = \ ... \]
\[ \text{where} \]
\[ \text{backG } = \ \text{back g (tabulate (bounds g) postord)} \]

The original function names are also still part of the graph library and perform postorder traversal of the depth-first search before applying their primed counterparts.

\[ \text{rgep} \ :: \ \text{Graph} \rightarrow \text{[Vertex]} \rightarrow \text{[[Vertex]]} \]
\[ \text{rgep g starts } = \ \text{rgep g postord starts} \]
\[ \text{where postord } = \ \text{postorderF (dfs g starts)} \]

\[ \text{gep} \ :: \ \text{Graph} \rightarrow \text{[Vertex]} \rightarrow \text{[Vertex]} \rightarrow \text{[[Vertex]]} \]
\[ \text{gep g starts ends } = \ \text{gep g postord starts ends} \]
\[ \text{where postord } = \ \text{postorderF (dfs g starts)} \]
\[\text{emep} :: \text{Graph} \rightarrow \exists \text{Vertex} \]
\[\text{emep} \ g = \text{emep}' \ g (\text{postorderF} \ \bowtie \ \text{dff} \ g)\]

\[\text{ecc} :: \text{Graph} \rightarrow \exists \text{Vertex} \]
\[\text{ecc} \ g = \text{ecc}' \ g (\text{postorderF} \ \bowtie \ \text{dff} \ g)\]

However, the real advantage becomes apparent in the \textit{generateTraces} function. It is only a small change, but it performs a single depth-first search for all graph algorithms. Note that the \textit{topSort} function is also replaced using the same depth-first search. Again only the difference is presented.

\[\text{generateTraces} :: \text{Graph} \rightarrow \exists \text{Int} \rightarrow \exists \text{Set} \ \exists \text{Vertex}\]
\[\text{generateTraces} \ g \ its = \ldots\]
\[\text{where} \ \text{postord} = \text{postorderF} \ \bowtie \ \text{dff} \ g\]
\[\text{mep} = \text{emep}' \ g \ \text{postord}\]
\[\text{necc} = \text{nested} (\text{reverse} \ \text{postord}) \ (\text{ecc}' \ g \ \text{postord})\]

Another optimisation that is performed in the actual implementation is based on the observation that the traces generated for a program often have a common tail. For instance, reconsider the traces generated in the example in 3.7, they all share the tail \([8, 11, 12, 33, 34]\). So when the weakest pre-conditions for these traces are calculated this part of the calculation is the same for all of them, meaning that the calculation can be shared. Therefore, the list of traces is converted to a forest in which they are maximally shared. The trees in these forests are reversed, meaning that the roots of the trees start with the final labels and work their way down to the initial label.

\[\text{share} :: \exists \text{Vertex} \rightarrow \exists \text{Forest} \ \exists \text{Vertex}\]
\[\text{share} [] = []\]
\[\text{share} \ ts = \text{foldl insert} (\text{toForest} \ \bowtie \ \text{head} \ \text{traces}) (\text{tail} \ \text{traces})\]
\[\text{where} \ \text{insert} \ \text{forest} \ \text{trace} =\]
\[\text{head} \ \text{trace} /= (\text{rootLabel} \ \bowtie \ \text{head} \ \text{forest}) = \text{toForest} \ \text{trace} ++ \ \text{forest}\]
\[\text{otherwise} = (\text{foldForest shareAlg} \ \text{forest}) \ \text{trace}\]
\[\text{traces} = \text{map reverse} \ \text{ts}\]

\[\text{shareAlg} = (\text{cons}, \ \text{nil}, \ \text{node})\]
\[\text{where} \ \text{cons} \ \text{hd} \ \text{tl} = \backslash \ \text{trace} \rightarrow (\text{hd} \ \text{trace}) : (\text{tl} \ \text{trace})\]
\[\text{nil} = \backslash_\rightarrow []\]
\[\text{node} \ \text{vertex} \ \text{forest} \ \text{trace} =\]
\[\text{let} \ \text{trace'} = \text{cut trace vertex}\]
\[\text{root} = \text{Node vertex} (\text{childs} ++ (\text{branch vertex} \ \text{trace'} \ \text{childs}))\]
\[\text{childs} = \text{forest} \ \text{trace'}\]
\[\in \ \text{root}\]
\[\text{branch root} \ \text{trace'} \ \text{childs}\]
\[\text{trace'} /= [] \ \&\& (\text{not} \ \bowtie \ \text{any} (\backslash \text{Node child} _\rightarrow \text{head} \ \text{trace'} \ == \text{child} \ \text{childs}) = \text{toForest} \ \text{trace'}\]
\[\text{otherwise} = []\]
\[\text{cut trace root}\]
\[\text{trace} /= [] \ \&\& \text{root} = \text{head} \ \text{trace} = \text{tail} \ \text{trace}\]
\[\text{otherwise} = []\]

Instead of iterating every trace like the \textit{calculate} function, see 3.63, the actual implementation of \textit{calculate} traverses the maximally shared forest. Meaning that it
Design and Implementation

calculates the weakest pre-conditions for all the traces in a single traversal. However, in the implementation the label function is a \textit{Map} from a label to its respective BoogiePL construct.

\begin{verbatim}
data Statement a b c = Var a  
   | Com b  
   | Toc c

type Statements = Map Lab (Statement LocalVarDecl Command 
   TocManifesto)

calculate :: Statements \rightarrow Predicate \rightarrow Forest Vertex \rightarrow [Predicate]
calculate stmts post traces = foldForest (cons, nil, node) traces post

where cons hd tl q
   | tl q == [q] = hd q
   | otherwise = (hd q) ++ (tl q)

nil = \{q \rightarrow [q]\}

node v ts = \{q \rightarrow ts \& wp v q\}

wp l q = case (lookup l stmts) of
   Just (Var var) \rightarrow wp_LocalVarDecl var q
   Just (Com comm) \rightarrow wp_Command comm q
   Just (Toc toc) \rightarrow wp_TocManifesto toc q
   Nothing \rightarrow q
\end{verbatim}
Chapter 5

Conclusion

5.1 Concluding Remarks

In section 2.3 we introduced the following research question.

To what extent is static testing a feasible testing technique?

Because, at the start of this thesis project, it was unclear how and if it was possible to construct a static testing framework, a prototype tool was build. This tool at least shows it is possible to statically test a procedure and is a strong indication that static testing is feasible. However, to be able to give a definitive answer to the above question more research and development is required. In this section we reflect upon this prototype tool and discuss its properties, advantages, and disadvantages.

5.1.1 Soundness and Completeness

TestBPL generates test-conditions using the weakest pre-condition calculus, but is this solution sound and complete? Let us first informally introduce the notion of soundness. For a solution to be sound it means that any error it finds is an actual error in the program, so it does not generate false positives. For TestBPL this means that every violation found by a test-condition is a violation of the specification \( \{ P \} s \{ Q \} \), where \( P \) is pre-condition, \( s \) a program, and \( Q \) the post-condition. Since a test-condition is an instance of a linear verification condition, it is sound if a linear verification condition is sound. Soundness of the linear verification condition can be defined as, where \( \hat{s} \) represents a program instance:

\[
P \not\subseteq \text{wp} \hat{s} Q \Rightarrow (\exists \sigma \in P \cdot \text{exec } \sigma s \not\subseteq Q)
\]  

(5.1)

Because a program instance is a refinement of a program, see (3.32) and (3.35), any violation found by a linear verification condition is real violation. So the linear verification conditions and thus test-conditions are sound.

Informally, the notion of completeness can be defined as; a solution is complete if it guarantees that when it does not find any error the execution is proven to be valid. For TestBPL this would mean that when its test-conditions are proven valid, the specification \( \{ P \} s \{ Q \} \) is also proven valid. Completeness of the linear verification condition can be defined as, where \( \hat{s} \) again represents a program instance:

\[
P \subseteq \text{wp} \hat{s} Q \Rightarrow (\forall \sigma \in P \cdot \text{exec } \sigma s \subseteq Q)
\]

(5.2)
Again because a program instance is a refinement of a program, it can not prove the correctness of the entire program. Note that this is also not the goal of software testing. Instead software testing builds confidence that the entire specification is correct.

5.1.2 Code Coverage

An commonly used metric to determine if a program has been tested adequately is code coverage. Code coverage is a measurement that describes the degree to which a program is tested. In section 2.1.1 three kinds of code coverage were presented: statement, branch and path coverage. While no extensive research was performed comparing the code coverage of TestBPL to normal dynamic testing frameworks, it is possible to reason about it.

Recall that the traces which TestBPL generates are a symbolic representation of program instances, which can be viewed as execution paths of a program. Note that given a set of desired iterations TestBPL generates all corresponding traces, thus execution paths. So TestBPL gives total control over the path coverage of a program. This also gives all little control over statement and branch coverage. TestBPL’s default behaviour is to generate all the traces of a branch, except for loops. So branch coverage is already pretty good, but when TestBPL is run with at least the desired iterations of 0 and 1 it guarantees complete branch coverage. This also means complete statement coverage, except of course for dead code.

5.1.3 Dynamic versus Static

In section 2.1.1 some issues of conventional (dynamic) testing were presented. Lets compare these issues for dynamic and static testing:

1. Isolation: the first and most important issue is the problem of isolation. In dynamic testing it is hard to achieve proper isolation, because the unit under test is executed in a foreign runtime environment. In this environment the developer often has to mimic the “normal” environment using dummy objects, see 2.1. In TestBPL proper isolation is less to almost no concern for the developer. Basically the developer only has to make sure that any procedure called by the procedure under test has been specified. TestBPL isolates the procedure under test because it does not consider the implementation of a callee, see the \( wp \) function for the call command in section 3.3. Instead it uses its specification. This advantage is partially due to specification based testing in general. However, dynamic specification based testing can not prevent the execution of a call. So this kind of isolation can only be achieved through static testing.

2. Improving Test-adequacy: in dynamic testing determining the code coverage is quite easy, but improving it is not. As we have seen in section 5.1.2, TestBPL guarantees full branch and statement coverage. Also path coverage can easily be improved by adding more desired iterations to the set.

3. Efficiency: at this point it is difficult to make any claim how static testing compares to dynamic testing. While static testing can maximally share the calculation of the weakest pre-condition for different traces, the cost of its overhead is unknown. Therefore, in order to determine if it is more or less efficient than dynamic testing further research is required.
4. Portability: because static testing does not execute the program under test, the results are not in any way platform dependent. Therefore, the tests performed on one platform do not have to be repeated on another.

5.2 Future Work

In this section some possible extensions for TestBPL are discussed and several suggestions are made for future research. In the first section, section 5.2.1, some small extensions are discussed. These are basically enhancements which were planned for this thesis project, but due to time constraints were not implemented. The subsequent sections are more profound and require more research.

5.2.1 Enhancing the Prototype

As mentioned earlier, the prototype tool TestBPL does not have the capability to derive test-conditions from the linear verification conditions and solve them. However, with the approach described in this thesis this should not be a problem to a future version.

Furthermore, in section 3.2.2 the “sub” language of BoogiePL, BoogiePLS was presented. The prototype tool can only cope with BoogiePLS, which basically is just the Hoare triple part of the language. While the TestBPL’s parser can already parse full BoogiePL, support for full BoogiePL should be added to a future version.

5.2.2 Improving Test-conditions

In section 3.4.3 the difference between syntactical and symantical traces is presented. At this moment TestBPL generates syntactical traces, meaning that it sometimes generates a trace that does not represent a valid program instances. Such traces do not pose any problems, but it is inefficient to consider them in the first place. An research project could explore the possibility of eliminating some or all non-semantical traces from the generation process. This might be possible to determine through automatic program analysis techniques. Another option is a generate and prune strategy. However, defining a filter can be quite complex. Our intuition is that a non-semantical trace can be identified by its guards, which will contradict each other.

Barnett and Leino [4] describe a technique to prevent duplication of the weakest pre-condition in different branches of a non-deterministic choice structure. Unfortunately, their technique requires that a program is pacified. It might be interesting to explore if the same techniques can be applied TestBPL in order to reduce the size of its test-conditions, without compromising their precision.

5.2.3 Automatic Testing

In chapter 2 two main challenges were identified for static testing, namely the automatic generation of test-data and the automatic generation of test-conditions. While the latter was pursued during this thesis project, the first also makes for an interesting research topic. In BoogiePL the expected behaviour of a procedure is expressed using a specification. However, in order for TestBPL to check a linear verification condition it needs a set of test-data. This seems redundant, because a
developer not only provides an abstract representation of the expected behaviour (the specification) but also a concrete representation (the test-data). Considering projects like TorX [32] or even QuickCheck [8], it should be possible to generate test-data for the linear verification conditions from the specification.

An alternative to automatic test generation and the generation of test-conditions is the automatic proving of the linear verification conditions. The major concern for this type of research would be to show that verifying an arbitrary linear verification condition is decidable, which is not the case for conventional verification conditions. Automatic proving of the linear verification conditions should still be regarded as static testing, because only a finite amount of program instances are verified. One might consider this approach to be an equivalent to bounded exhaustive testing, where the bounds are determined by the generated program instances.

5.2.4 Control and Feedback

Because TestBPL is a prototype, the creation of a proper user interface did not have priority. Consequently, the user interface is minimalistic to say the least — TestBPL does not even have a command line interface. Currently, it has to be controlled using the interpreter from the Glasgow Haskell Compiler, see appendix A. However, ideally TestBPL would be integrated in an IDE and would not require much of an user interface of its own.

From a usability point of view, several improvements/extensions can be made to TestBPL:

1. Customise Loop Handling: Recall that TestBPL requires a set of “number of iterations” which it has to consider during linearization. At this moment TestBPL only accepts a single set, meaning that all loops are linearized using the same set. It would be preferable to be able to specify the “number of iterations” on a loop to loop basis.

2. Customise Call Handling: TestBPL does not inline a call to a procedure, instead it uses its specification. We could add a feature that allows a developer to change this default behaviour and inline certain calls. Note that when a call is allowed to be inlined, it introduces the problem of recursive calls. Like the loops these must be linearized. Unfortunately, this is not the same problem entirely. Instead of a single back edge an recursive call introduces another back edges, the tail.

3. Customise Branch Handling: When linearizing a procedure, TestBPL considers all possible branches. With branches we mean all gotos which do not introduce a cycle. Like the loops and calls, it would be nice if it was possible to deviate from this default behaviour and allow a developer to disable certain branches. Note that this introduces the possibility that no semantical trace is generated, but this is the responsibility of the developer.

Of course, all these features should also be incorporated into the user interface somehow.

An important aspect of a user interface and possibly interesting research project is generating feedback for TestBPL, more specifically error reporting. Consider that a violation is found only when a test-condition is solved. This is a long way from the original program with which the developer works, or even the BoogiePL representation of that program. In order for the user interface to be practical, some
solution must be found to trace a violation back to the original program and provide the user with sensible feedback.
Appendix A

Installation and Configuration

A.1 Code Availability

The source code of the TestBPL prototype tool can be found at:

http://www.cs.uu.nl/wiki/Students/ElmarKeij

A.2 Usage

In order to build TestBPL the following tools/packages must be installed:

1. The Glasgow Haskell Compiler 6.6.1 or higher
2. The Utrecht University Attribute Grammar System, in uuagc-0.9.6 or higher
3. Parser Combinators and Pretty Print Library, in uulib-0.9.3 or higher

Because the TestBPL prototype tool does not yet have a user interface it has to be run in the ghc interpreter. You can start TestBPL by running make run in the TestBPL directory. In the interpreter a BoogiePL program can be tested using the execute function. For example, execute "test/basic/cf1.bpl".
Bibliography


