Proving Programs: An extension to the UTP theorem proving assistant Saoithín

Andrew Anderson
B.A. (Mod) Computer Science 2011
Supervisor: Dr. Andrew Butterfield

April 11, 2011
Abstract

This project investigates the verification of correctness conditions for computer programs through the use of automated verification condition generation. Building on the existing capability for specification of programming languages within the Saoithín proof assistant, we implement an extension which allows input and manipulation of programs written in such languages. We also implement a verification condition generator and investigate the application of verification condition generation to a simple programming language under weakest-precondition semantics.
Declaration

I hereby declare that this thesis is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

Andrew Anderson

Date
Permission to Lend

I agree that the Library and other agents of the College may lend or copy this thesis upon request.

Andrew Anderson
Acknowledgements

I would like to thank my supervisor, Dr. Andrew Butterfield, for his guidance and patience during the course of this project, and also for many humourous and enlightening conversations on any number of topics. Thanks also go to Dr. Gavin Doherty for taking the time to read this project.

Last but most certainly not least, my sincere gratitude to all of the friends who provided unwelcome distraction from important work at critical moments, thereby preserving a degree of sanity in the report writer.
Contents

1 Introduction 7
   1.1 Project Aims .............................................. 8
   1.2 Project Execution ......................................... 8
   1.3 Report Structure .......................................... 9

2 Background 10
   2.1 What is Formal Methods? ................................. 10
      2.1.1 Formal Semantics .................................... 11
      2.1.2 Logical Calculi ....................................... 12
      2.1.3 Hoare Logic ........................................... 13
      2.1.4 UTP .................................................... 13
   2.2 Correctness in UTP ......................................... 15
      2.2.1 Total vs. Partial Correctness ...................... 15
   2.3 Verification Conditions ................................. 16

3 Project Motivation 18
   3.1 Overview .................................................. 18
      3.1.1 Tool Structure ....................................... 19
   3.2 Goals ....................................................... 19
5 Conclusions

5.1 Learning Outcomes

5.1.1 Development

5.1.2 Theory

5.2 Difficulties Encountered

5.2.1 Architecture and Automation

5.2.2 Documentation

5.3 Critical Summary of Progress

5.3.1 Interface

5.3.2 Theory Support

5.3.3 Verification

5.4 Future Work

5.4.1 Interface

5.4.2 Program Logic

5.4.3 Verification

Appendix A: Source Code
Chapter 1

Introduction

Theoretical work in formal methods takes the form of design and proof of properties of systems of mathematical logic[6]. Pen-and-paper proof work is often impossible in the large theoretical frameworks needed to reason about computing, leading to the development of tool support for such work, and particularly, the development of theorem proving assistants. This project undertakes an extension to the proof assistant Saoithin, developed primarily by Dr. Andrew Butterfield, who also supervised this work.

Tools

A proof assistant in the context of this report means a program which is used as an interface to proof work. The proof assistant manages all the information about the proof undertaking, usually keeping a description of the entire theoretical framework within which the proof takes place. The theory is manipulated by the user through the tool’s interface.

Proof assistants are used to aid in proofs that are infeasible by manual methods, or which require rigorous verification of their correctness. Because the proof assistant is the sole interface to the proof, it is possible to ensure that the user does not take erroneous proof steps, and to perform various sanity checks. Using a proof assistant can thus be thought of as undertaking a human-guided machine-checked proof, in the sense that the proof assistant allows the user to apply only valid transformations to the objects under scrutiny.
Benefits of Tool Support

Proof assistants are helpful are in numerous ways. Interesting proofs usually proceed within theoretical frameworks consisting of sets of axioms, previously proven theorems, observation variables, type systems relating inhabitants of the theory and other concerns. Managing this information becomes more and more difficult as the theory grows, and is a hindrance to proof work.

Moreover, because proof assistants are an interface to proof, they can aid with tedious manipulation tasks. Undertaking textual proofs of any sort usually involves a lot of mental filtering, pattern-matching, and generally sifting through the assembled information looking for useful constructs, which can be greatly assisted and sped up by a tool to allow the user to concentrate better on the details of the problem at hand.

1.1 Project Aims

As set out in the abstract, the project’s main undertaking is the development of an extension to the theorem proving assistant, Saoithín. This extension adds to the existing system a capability for inputting computer programs to the system, and for the automatic generation of the proof obligations concerning a computer program under scrutiny.

1.2 Project Execution

Saoithín is written in the functional programming language Haskell, and the program’s user interface was built on the wxHaskell GUI library. These two technologies are therefore central to the execution of the project. The extension to the user interface necessary to allow programs to be input was developed on wxHaskell, and both this extension and the verification support logic were written in Haskell.

Haskell

Haskell is a purely functional, statically typed, lazy programming language, with extensive type inference[1]. In practical terms, this means that programs are not
expressed in terms of stateful computations effecting some imperative change to
variables, but in terms of referentially transparent functions, always returning
the same result for a particular input. Haskell is a lazy language, which means
that no computation actually takes place until the result of an expression is asked
for. This allows, among other things, to represent cyclic and infinite data struc-
tures efficiently, and to operate upon them without errors.

wxHaskell

wxHaskell is a graphical user interface library. It’s a Haskell wrapper around the
cross platform wxWidgets interface library, which is written in C++. The wxWid-
ggets library uses the native API of the platform it is on to provide an interface
that looks and feels just like the rest of the system[3]. Saoithin is constructed
with wxWidgets to enable its use on multiple platforms[2].

1.3 Report Structure

The report first discusses the necessary background and theoretical concepts in-
volved with the project, before describing its overall motivation and goals. Fol-
lowing this, the design of the programmatic extension to the system is laid out,
and the implementation of that design presented and discussed. The report then
describes an example programming language and a formulation of correctness,
and demonstrates the use of the system in proving properties of some simple
programs. The conclusions and closing remarks follow, as well as a discussion of
learning outcomes of the project, a critical summary of the progress made, and a
brief discussion of possible future work.
Chapter 2

Background

This chapter lays out an overview of the theory involved in this project, and gives definition of some technical terms that might not be familiar to the reader. It is assumed that the reader has at least a passing familiarity with the areas of

- Programming
- Mathematical logic
- Syntax and semantics of programming languages

2.1 What is Formal Methods?

Formal methods in the context of software engineering is the application of mathematical techniques to the problem of developing provably correct software, for some definition of correctness.

If we imagine a piece of software as a mapping from some problem domain to a solution domain, then the discipline of formal methods is concerned[6] with:

A) **Specification**, or obtaining a rigourous definition of the problem domain,

B) **Design**, or deriving a valid mapping from problem to solution, and

C) **Verification**, or proving that a piece of software is a correct implementation of a design
The usual approach to full-on formal development of software is to express a specification of the problem in a formal language, and mathematically derive a design (or program) from that specification. The formal languages used for this purpose are often just small, rigorously specified programming languages. There are several schools of thought regarding the specification of such languages, though all methods share a common structure.

Generally, a formal system used in the production of verified software consists of a set of language constructs with which to express programs (syntax) and a set of inference rules detailing how execution of statements in the language should proceed (semantics).

### 2.1.1 Formal Semantics

**Operational Semantics**

Operational semantics gives meaning to a formal language by associating with each language construct a set of rules, which lay out either

A) the individual steps of execution of a computation, or

B) how the overall result of the computation is obtained

The former method is known as small-step semantics, since it details how execution proceeds within single expressions. The latter is known as big-step semantics, since it relates an expression directly to its result without detailing how execution proceeds within the expression. In both cases, execution proceeds globally by repeatedly applying to a program one of the semantic rules of the system, stating how it may take a step of execution, or achieve a result. For a more in depth discussion of operational semantics, see Hennessy[7].

**Denotational Semantics**

Where operational semantics provides a direct description of the execution of a language in terms of the expansion of expressions (in essence, it is a set of rules for the interpretation of the language), denotational semantics describes the meaning of statements by translating them into a denotation, which is usually an abstract mathematical object. Denotations are said to belong to some domain, or mathematical formalism. Usually, statements of a programming language are
translated into statements in a mathematical formalism (objects of the semantic domain), in which state they can better be manipulated by formal means, and have their properties proven.

Denotational semantics has an important tenet known as compositionality, which essentially means that the “meaning” of any statement under a denotational semantics should be exactly the sum of the meanings of all of its parts. Crucially, this means that we can replace statements with others which have the same "meaning" (i.e. which map to the same element of the domain) under a denotational semantics.

Denotational semantics is, roughly speaking, the compilation to the operational semantics’ interpretation - it provides a set of rules, not for execution of the language, but for its translation into a different form. Languages defined by a denotational semantics only have meaning relative to the target language (or domain) of the semantics, in contrast to the direct meaning conferred by an operational semantic definition. A thorough explanation of denotational semantics can be found in chapter 4 of Turbak[11].

**Axiomatic Semantics**

Where operational semantics describe how to execute a language, and denotational semantics describe how to translate it into an abstract domain, axiomatic semantics give it meaning by relating statements of the language to theorems or axioms in some logic. In effect, the language is given meaning by a description of how its statements behave under the laws of the logic. The meaning of any statement is exactly what can be proven about it. Hoare logic[8] is the canonical example of an axiomatic semantics.

### 2.1.2 Logical Calculi

As laid out previously, a common approach taken in formal methods is to provide a definition of a language in which to describe the problem, and a set of inference rules, or semantics, concerning that language. Such a tuple is often called a formal system, a logical calculus, or sometimes simply a formalism. There are many such systems in existence, usually specialised to describe a particular type of problem.

A popular such system is the lambda calculus, first set out by Alonzo Church in the 1930s. The lambda calculus and its variants form part of the theoretical basis
for a multitude of programming languages and formal methods tools, such as the Haskell programming language, and the Z formal specification language.

2.1.3 Hoare Logic

The foundation of the logical calculus used in this project is the Hoare Logic. A precise definition of this calculus can be found in [8] and an excellent summary in [10], so here we focus only on the portion relevant to this background exposition, which is the introduction of a widely used representation of assertions about programs and their environments.

**Definition** (Hoare Triple) A Hoare triple has the structure

\[
\{\text{Pre}\} \text{Prog} \{\text{Post}\}
\]

Where \text{Pre} is an assertion about the state of the environment before the execution of a program, \text{Prog}, and \text{Post} is an assertion about the environment after the execution of \text{Prog}.

2.1.4 UTP

UTP is the formalism used in the tool at the core of this project. It was put forward by Hoare & He in their 1998 book, “Unifying theories of programming”[9]. They aim to unify existing approaches to programming language specification within a single theoretical framework. To this end, their formalism combines concepts from operational, denotational, and axiomatic semantics. The formalism is a refinement of the earlier Hoare logic.

The goal of UTP, stated throughout the book, is to provide a small and complete calculus upon which any formalism of programming can be based. The authors characterise the system in their preface thus: “This is a presentation within predicate calculus of Tarski’s theory of relations, enriched with his fixed point theory, and applied to Dijkstra’s simple non-deterministic sequential programming language.”

UTP represents programs as mathematical *predicates* over a set of observation variables, representing the program’s environment, or the real world. Each predicate is a relation between the initial and final values of the observation variables,
i.e. the state of the real world before and after the predicate is evaluated. Sequential programs are built up in UTP by using combinators to join predicates. These predicates correspond to individual items in the triples of Hoare logic.

UTP provides a denotational semantics, in the sense that constructs are described by equating them with their expansion in terms of simpler constructs. The UTP conditional construct, for example, is described denotationally as a more complex statement in first-order predicate logic.

**Definition** (Condition) A conditional branch in UTP has the structure

\[ P \triangleleft b \triangleright Q \equiv (b \land P) \lor (\neg b \land Q) \]

Where \( P \) and \( Q \) are predicates (read: programs) and \( b \) is some discrete Boolean condition. The statement can be read as: “Conditionally \( P \) or \( Q \) depending on \( b \) is equivalent to the truth of \( b \) and the behaviour of \( P \) or the falsehood of \( b \) and the behaviour of \( Q \).”

UTP provides an axiomatic semantics in that predicate logic is the foundation of the system, and predicates are given meaning by describing how they interact with each other in an equational logic setting. When defining a new construct, it is axiomatically defined how it associates, distributes and commutes within the logical system. For example, one interaction of the sequential combinator, ; with the conditional would be as follows.

**Definition** (Condition-Sequence right distribution) The distribution of the sequential combinator to the right of the conditional is defined as

\[ (P \triangleleft b \triangleright Q); R \equiv (P; R) \triangleleft b \triangleright (Q; R) \]

Which states simply that we can validly rewrite either side of the equation as the other. Since program \( R \) is inevitably going to be executed, we can move it inside the conditional statement, but we must place it at the end of both branches.

It is in this sense that UTP is a unification of axiomatic and denotational semantics: constructs are defined denotationally, but their interactions with other objects in the system are described axiomatically.
2.2 Correctness in UTP

To verify a program written in any formalism, we need to be able to express the properties to be proven. In UTP, this expression of correctness is as a predicate transformer semantics over the core language. Predicate transformer semantics are a way of defining the semantics of a programming language by associating with every statement of that language a predicate transformer which describes the effect of the statement as a function between two predicates. These “before” and “after” predicates are commonly known as the precondition and postcondition of the statement, or program, in question. In UTP, the system used is known as the weakest-precondition semantics, and was introduced in [5]. It specifies for each statement of the programming language a function which maps a predicate representing a postcondition that holds after the program’s execution to a predicate representing the weakest possible precondition which must be true before execution so that the postcondition is established by the program.

Statements in the semantics are of the form

Definition (weakest precondition)

\[ wp(\text{Prog}, \text{Post}) \rightarrow \text{Pre} \]

This states that the weakest precondition, \( Pre \) of the program \( \text{Prog} \) follows from the postcondition \( \text{Post} \), or in other words: if \( Pre \) is established before the program executes, then the execution of the program will establish \( Post \). Importantly, this relies on the termination of the program to establish the postcondition, or in other words, we’re aiming for a demonstration of total correctness.

2.2.1 Total vs. Partial Correctness

Proofs of total correctness of a program show that the program both terminates and does so with the correct result. Proofs of partial correctness, on the other hand, show that if the program terminates, then it does so with the correct result. This means that if a program is only proven partially correct, there is the explicit possibility that it might not terminate for some input. Weakest-precondition is a calculus expressing total correctness. The corresponding calculus of partial correctness is known as weakest liberal precondition.
Partial correctness, being predicated on the termination of the program and therefore a weaker statement than total correctness, is usually easier to prove - in effect, proofs proceed on the assumption that the program terminates.

The outcomes of total or partial correctness proofs, respectively, are statements like "It works as expected and terminates" versus "It works as expected if it terminates".

### 2.3 Verification Conditions

The weakest-precondition semantics can be used to express correctness conditions for programs, by fixing the total postcondition, and recursively applying the rules of the semantics to a program, obtaining a total precondition. Given a program, and a postcondition stating the property we would like to hold after execution for the program to be deemed correct, we can derive the precondition under which the program behaves as we want. It is then up to us to prove that this precondition is a tautology. We call such preconditions, and the intermediate proof obligations resulting from their derivation, the **verification conditions** of the program.

Sequential programs of any size are composed in such a way as to have long sequences of predicates, where the precondition derived for one predicate serves as the postcondition for other predicates. By repeatedly applying the rules of the predicate transformer semantics to the statements of the program, we end up with a collection of verification conditions, and eventually, a derived precondition for the program as a whole. However, because of the structure of the semantics (the specification is almost always recursive), it is tedious and time-consuming to manually apply to a program of any length.

It’s common[10] to automate the process of deriving the verification conditions, handing off the burden of analysis to a tool known as a verification condition generator (VCG). After running the program and a correctness condition through a VCG, one ends up with a collection of verification conditions, or proof obligations, which must be shown to hold in order for the program to satisfy the correctness condition.

An important point to raise here is that Saoithín is not designed for autonomous proof, but for assisted proof only. This verification support does not actually further the proof process by itself, it is rather an automation of the process of inspecting the program and generating the proof obligations. The tool user is still
tasked with discharging the proof obligations just as they would undertake a regular proof. As laid out in [10], usually a verification condition generator would be used in conjunction with some kind of automatic proof system. However, proof obligation generation is also a very useful tool in manual proof, reducing the burden of building up a theoretical description of the problem, an often tedious and error-prone task.
Chapter 3

Project Motivation

This chapter lays out an overview of the tool structure necessary for explaining the justification for the project, as well as the individual goals of the project.

3.1 Overview

Saoithín is a proof assistant in the sense laid out previously. It allows a user to build up theories describing formal systems through a graphical interface, and provides support systems which handle matching proof sections against the laws and theorems present, transforming proof sections according to user choices, tracking the provenance of theoretical objects (have they been proven, or simply asserted?), and providing general support for theoretical work. However, the tool is only a proof assistant, not an automated prover. The user has to manually undertake each proof step. The tool streamlines some proof processes, but it does not take part in any reasoning by itself, as distinct from systems which undertake automated proof.

Saoithín, although it allows the description of programming languages, currently lacks the ability to input and manipulate whole programs easily, as well as any tool assistance for the verification of those programs, although verification could be undertaken by deriving and inputting all the verification conditions (and the program) manually. The user would essentially assume the role of a parser, entering each program statement as a separate predicate in a theory, and wiring together their control flow. This is clearly undesirable and infeasible in the case of larger programs.
3.1.1 Tool Structure

Theories in Saoithín, as has been explained, contain a description of some mathematical objects and rules for their manipulation. For the purposes of this project, verifying programs, we need only consider some key features, since the rest we don’t need to use.

Specifying a programming language starts out with deciding on the syntax of some language constructs. Saoithín allows the definition of these constructs, their arities and precedences, and whether they associate left, right, or not at all. A precedence is associated with a construct to clarify the order of evaluation of items in an expression. Language constructs and their precedence and associativity are entered in the Language and Precedence tabs.

Any observation variables of the theory can be entered in the Obs tab. Observation variables are global, visible from every theorem, expression or program in the theory. Observation variables are used to represent global properties, like whether the program under examination has started, or terminated, or as state variables. They can also be used, as they are in later sections, in the definition of specific language constructs.

The constructs of the language interact with each other, and programs will be expressed as combinations of language constructs. The semantics of the programming language are specified by adding to the laws section of the theory, under the appropriately named Laws tab. Laws describe axiomatically how language constructs interact. For an example of such a law, see expression 1.1 previously. Language constructs can also be given a denotation here, by stating in a law that they are equivalent to some other construct. For an example of such a law, see expression 1.2 previously.

3.2 Goals

3.2.1 Programs

To be able to reason about programs, we first need to represent them in the tool. This is the first goal of the project: to have programs as an entity within the proof assistant, and support program entry from the GUI. In particular, the tool requires an interface to support the following objectives:

- Entry and editing of program source code
• Loading and saving of program source code

• Parsing of the program source into a theoretical representation of the program

• Entry of conditions for verification

• A way to trigger the verification condition generation process

Fortunately, the tool already has a parser for textual representations of theoretical objects; it has access to the specification where the user defined the syntax of the programming language. However, parsing whole programs is not something that the text entry interface was designed for. Presently, each statement in the program would have to be entered manually as a discrete predicate. This is clearly infeasible for programs of any size. It makes better sense to allow the user to enter programs through a standard text editor, which is a familiar interface to all computer users. This editing window can also be used as the anchor for the various functions that pertain specifically to programs entered into the proof assistant for manipulation, laid out in the list just previous.

Armed with a representation of programs in the proof assistant, and an interface allowing users to both enter and manipulate their source code, and invoke the various tools provided by the project on the program, we can move on to what those tools provided by the project are.

### 3.2.2 Verification Condition Generator

It is tedious and error-prone to manually derive the verification conditions for even a small program. Thus, the second goal of the project is, given a language and a definition of a predicate transformer semantics over that language, to perform automated verification condition generation. The verification support needs to take:

- a user specified transformer semantics
- a user specified programming language
- a user specified postcondition
- a user specified program
...and apply the transformer semantics to the user’s program, given their postcondition. The result of this process is that the program is parsed into a new theory in the tool, and the proof obligations that would establish the given postcondition are entered into that theory as conjectures. The user can then manipulate these conjectures according to whatever theory of programming underpins their work, and eventually either prove or disprove the correctness of the program.

3.2.3 Examples

The third goal of this project is then to investigate verification condition generation by developing some example theories. In particular, it would be useful to define a simple programming language, and a corresponding weakest-precondition semantics, and then verify some example programs in that language.

3.2.4 Summary

In summary, this project proposes to augment the tool as laid out in the Tool Structure section with an interface to simplify the input and manipulation of programs. On top of this, an investigation into the generation of verification conditions from those programs will be undertaken, comprising the development of a simple verification condition generator, and an evaluation of it. The evaluation will consist of the development of a theoretical model of a sample programming language / transformer semantics pair, and the verification of some simple programs with the tool.
Chapter 4

Project Implementation

4.1 Interface

The first objective tackled in this project was to extend the tool’s interface with one that allowed programs to be manipulated as source code, as laid out previously in section 3.2 of the report. At a minimum, we require the ability to load and parse programs written in whatever programming language the user has designed in the tool. However, it would be tedious to have to use a text editor to fix mistakes in source code and switch back and forth between the tool and the editor, so integrating some sort of text manipulation facility to the tool itself seemed an obvious way to sidestep that particular problem.

4.1.1 Design

The interface design is that of a simple text editor, with a standard menu bar and text entry area. This editing window is the entry point for the source code of the program to be verified, whether loaded from a file or written in situ. The menu is divided into two subsections, Actions and Debug. Standard text editor functions like loading and saving of files are available from the Actions submenu as well as the option Finalise, which triggers the glue logic that builds the program code entered into a theory and runs the verification condition generator.

The Debug menu contains some debugging options Sync and Show, which are sanity checks for stages of the process. The sync option synchronises the state of the editing window with the internal representation of the program in the
tool. The show option displays the result of parsing the current contents of the editor window, and can be used to check if the entered program parses against the language defined in the user theory of programming.

![Program Editor](image1.png)

Figure 4.1: Entry window

Once the program has been entered, whether loaded from some existing source file or entered manually, the next step in the process is building a description of it into one of the tool’s theories. The program is parsed and individual lines become named predicates within the theory created to hold the program. The verification generator is then run over this set of predicates given a correctness condition for the program, which is supplied by the user. What actually happens is that the user selects Actions→Finalize and is presented with a dialog box in which to enter the postcondition for the program.

![Postcondition Text](image2.png)

Figure 4.2: Condition entry dialog

The glue code now runs and builds the program and postcondition into a theory, then runs the verification condition generation process with the supplied postcondition, resulting in a theoretical description of the program, ready for proof work to begin.
Implementation

The code implementing these interface features is pretty standard GUI boilerplate and would not serve much purpose to include in this report. It can be found in the source file “SaoithinProg.lhs” on the included CD. The only notable feature of the implementation is that safeguards have been put in place against accidental closing of the interface during program entry or losing the window. All of the actions that lead to the abandonment of the program entry process are captured and a dialog pops up to present the user with a choice to quit or continue their work. As well as this, the editor window state is synchronised into the program state in the tool at each opportunity to do so, following user actions.
4.2 Program Logic

This section covers the design of the logic gluing together the various aspects of the extension.

4.2.1 Representation of Programs

Programs are represented as theories within the tool, in the sense that they are a collection of statements of some logic. Each individual statement in the source program, after it has been parsed, is represented in the proof assistant as a predicate of the language the user has created. This means that once a program has been entered into the tool, the source code of the program is no longer required. Any changes which need to be made to the program as a result of errors can be made to the individual erroneous statements via the predicate editing interface already provided. This may seem like a slight drawback, but if an error is discovered in the program after it has been finalised, the source code needs to be reparsed in any case.

As a further design consideration, typically program source code lives on a disk where immediate access to the code is expected by the developer. Often, programming projects live inside a version control system and have automated build scripts which process them into a form to be shipped to end users. It is a reasonable expectation, therefore, that the program source code will not be gobbled up by the verification system, but rather that verification can take place on programs without the verification system owning the source code in any way. With these considerations in mind, it was decided that the verifier should be able to load and save program source code freely, but would not track it like the tool does with, for example, the on-disk representation of theories.

Justification

There are several reasons for the choice of the existing "Theory" tool datatype as a representation of programs. The first, and most obvious, is that a program is semantically very similar to a mathematical theory, which is what that datatype is intended to represent. If we look at the structure of a typical program to verify, it has associated with it several kinds of theoretical objects.

- observed (or global) variables
• statements of some language, usually concerning both global and local variables

• verification conditions for the program (predicates over language statements)

• an initial and final environment (configurations of variables)

This is similar to any standard tool theory, in that it is a collection of predicate logic statements and variables, about which the user would like to prove some properties.

Another reason for this representation is that we would like to be able to save the user’s progress towards verification of a program at any point during the process, and that capability is already provided for theories. It would be tedious to have to use the system to re-parse and regenerate the program and verification conditions every time the proof assistant was started, and representing the program as merely source code would mean that we’d have to do this.

Leveraging the existing representation of theories is a low-cost means of representing programs and their corresponding data and proof objectives within the tool, and also gives us access to the preexisting support code written for that representation.

### 4.3 Program Entry Process

The process of taking an existing program and getting it into the tool as a collection of theoretical objects for verification is quite straightforward from the user’s perspective. The workflow for doing so proceeds as follows.

• Open the proof assistant and load the required theories

• Select the theory representing the appropriate language and choose to create a program over it

• In the extension GUI which pops up, navigate to the actions menu and choose to load a program

• Choose the program to verify from the file chooser

• Look over the program’s source code and make any necessary changes from the extension GUI
• When the program is ready, choose to finalize the process from the actions menu
• Enter a global postcondition for the correctness of the program and hit OK

At this point, the machinery of the extension takes over and processes the program. In this step, the program source is first parsed into a sequence of named predicates, and these predicates are inserted into the theory which holds the program. The program’s total postcondition is also parsed, having been entered by the user, and is inserted into the theory along with the program source. The verification condition generator is then invoked upon the parsed program and total postcondition, and the intermediate verification conditions are generated. These are also inserted into the theory as a system of named predicates. As part of this generation process, a total precondition for the program’s correct behaviour is established. As the Examples section of this report shows, this precondition is initially quite vague, relying on numerous unproven intermediate observations, but it becomes more concrete as the user’s proof process advances.

From this point on, the user may begin manipulating the theoretical representation of the program in order to satisfy the total precondition of the program. Since we are using the weakest precondition semantics to perform the verification, the program can be said to have been proven correct with regard to the total postcondition once the total precondition can be shown to follow from it. If we are using a theory which incorporates some notion of the environment of the program, such as the theory of designs, as set out in [9], then frequently we have an extra assertion added to the theory, which is that the total precondition does, in fact, hold.

4.4 Verification Condition Generation

The verification condition generator developed for this project is quite straightforward. As laid out in the verification section of the Goals chapter, there were several objectives it needed to satisfy.

4.4.1 User-specified Transformation

A major design goal of the verification support system was to allow the user to provide their own theoretical description of the verification calculus, weakest-precondition in this case, for verifying code. This is necessary because the user
will need to provide a formulation of the wp-calculus for each language in which they want to construct verifiable programs. However, some constraints must be imposed so that the verification support can rely on a consistent interface across different theories.

The VCG presumes that the user has specified their weakest-precondition semantics as a system defining a binary operator, \( wp \) over predicates. The left operand of the operator is presumed to be the program and the right operator the postcondition.

\[
\text{Prog } wp \text{ Post } \rightarrow Pre
\]

The result of application of the operator is the weakest precondition implied by the given program and postcondition. The operator can be typed:

\[
wp : P \times P \rightarrow P
\]

4.4.2 Programming Language

Alongside a theory containing a suitable definition of weakest preconditions, the user must also necessarily provide a theoretical description of the object language of verification, the programming language in which the predicates operated upon by the weakest precondition combinator are written. There are no restrictions imposed upon this language by the verification system, since it operates over arbitrary predicates. The only restrictions are imposed by the limits of what can be defined within the tool.

4.4.3 Bringing it all together...

Given a theory containing a formulation of weakest preconditions, and a corresponding theory containing a description of some programming language, the verification support system takes a program constructed in that language, and a total postcondition for that program. Thus the left and right operands of the weakest precondition combinator are filled, and the semantics provided by the user can be invoked to generate a precondition for the program. To verify the program correct, the user must then show by some means that this (generated) total precondition is a tautology, or follows from some external circumstance.
4.4.4 Recursion

Sequential programs, such as those under examination in this project, have a structure that necessitates a recursive specification of the weakest precondition semantics. This is to do with the nature of weakest preconditions and sequences of operations: if two programs are run in sequence, the precondition of the latter program is the postcondition of the former. Logically, since the programs are run one after the other, the state of the world after the first has terminated must be the same as the state of the world just before the second commences, otherwise the operation of sequencing them would have no meaning - if the world can change arbitrarily between steps in a sequence, then the sequence does not have any real impact on the execution of the overall program.

The verification support system breaks apart sequences into chains of discrete named predicates, representing each step in the execution of a program.

4.5 Theory Support

In order to demonstrate the extension to Saoithin, we need a simple programming language and a predicate transformer semantics for it. We take the simple sequential programming language "XYZ-Design", for which a specification comes built-in to Saoithin, and extended it with the capability to express unbounded conditional loops (the familiar programming concept of a while loop). This language was chosen primarily because of its simplicity. We provide a complete definition of a weakest precondition semantics over this language.

4.6 Language

The language "XYZ-Design" as laid out below expresses programs of three or fewer variables. These are the observational variables of the theory of programming, and can be thought of as registers from the point of view of the programmer.

The language consists purely of control-flow constructs, with the exception of the assignment construct. Because of the nature of UTP, the language constructs operate upon arbitrary predicates. The tool user can formulate a theory of the computation of numerical expressions in UTP, for example, and then use it in conjunction with this theory of programming to construct programs performing
numerical analysis. Examples of this will be shown in later sections, using theoretical formulations of arithmetic and equality over integral values built into the proof assistant.

4.6.1 Observational Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ok, ok'</td>
<td>$\mathbb{B}$</td>
</tr>
<tr>
<td>x, y, z, x', y', z'</td>
<td>$\mathbb{Z}$</td>
</tr>
</tbody>
</table>

The variables $ok$ and $ok'$ represent the state of the program’s execution, where $ok$ is true if the program has started, and false otherwise, and $ok'$ is true if the program has terminated, and false otherwise.

The variables $x$, $y$, and $z$ are freely modifiable by programs according to the constructs of the language. In this formalism, programs map the initial states of variables ($v$) to the final states $v'$. Formally, we say that programs are predicates parameterized over the state space $\{x, x', y, y', z, z'\}$. This representation of programs comes directly from UTP[9].

4.6.2 Syntax

<table>
<thead>
<tr>
<th>Name</th>
<th>Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assign</td>
<td>$v := e$</td>
</tr>
<tr>
<td>Sequence</td>
<td>$P_1; P_2$</td>
</tr>
<tr>
<td>Skip</td>
<td>$\text{skip}$</td>
</tr>
<tr>
<td>Condition</td>
<td>$P \triangleleft b \triangleright Q$</td>
</tr>
<tr>
<td>While</td>
<td>$b * P$</td>
</tr>
</tbody>
</table>

4.6.3 Semantics

In the usual UTP style, we provide a mixed denotational/axiomatic semantics for "XYZ-Design".

Assignment

$$(v := e) \equiv (\text{ok} \Rightarrow (\text{ok}' \land (v' = e))) \land (\text{Var'} \setminus v' = \text{Var'}\setminus v)$$
The statement “A variable v is assigned the value of expression e” is defined denotationally. The truth of ok (i.e. the program has started) implies that the program terminates, the final value of variable v is the value of expression e, and the other observational variables are unchanged. We write Var\'\(\backslash v\) to denote the set of dashed (or final) observational variables less the element \(v\). Asserting that the set of final variables less \(v\) is equal to the set of initial variables less \(v\) is how we denote that \(v\) is the only variable modified by this update.

Sequence

\((P; Q) \equiv (\exists \bullet VarM : P[VarM//Var'] \land Q[VarM//Var])\)

The sequence of two arbitrary predicates \(P\) and \(Q\) is defined denotationally. We say that the sequence means there exists a set of intermediate variables, \(VarM\) such that the set of final variables of \(P\), \(Var'\), and the set of initial variables of \(Q\), \(Var\), are both equivalent to \(VarM\). We do this by stating that the corresponding name capture avoiding substitutions are valid.

Skip

\((P; skip) \equiv P\)
\((skip; P) \equiv P\)

The \(skip\) construct is defined axiomatically as the right and left unit of sequential composition. \(skip\) represents the program that terminates successfully, but performs no action. We could also have defined \(skip\) denotationally as follows:

\(skip \equiv (ok \Rightarrow ok' \land (Var' = Var))\)

Although both definitions are valid, and the axiomatic characterisation of the operation follows logically from the denotational one, the axiomatic definition is more useful from the point of view of manipulation of programs - we want these \(skip\) operations to reduce easily, since we commonly use them as “bookends” to represent the termination of some branch of a computation. An example of this use can be found in the forthcoming definition of the \texttt{while} construct. We choose the axiomatic version because our primary purpose is not providing a basis for computation, but for proving properties of computable expressions - a subtle but important distinction.
Condition

\[ P \triangleleft b \triangleright Q \equiv (b \land P) \lor (\neg b \land Q) \]

The conditional construct “if \(b\) then \(P\) else \(Q\)” is defined denotationally as the truth of \(b\) and the behaviour of predicate \(P\) or the falsehood of \(b\) and the behaviour of predicate \(Q\).

While

\[ b \triangleright P \equiv (P; b \triangleright P) \triangleleft b \triangleright \text{skip} \]

The while loop is given a operational semantics in terms of the other constructs set out just previously. If the condition is true, the body of the loop executes once, followed by another (recursive) evaluation of the loop; if it is not then the loop terminates. It is common to see the \textit{skip} construct used as a bookend in this fashion.
4.6.4 Weakest Precondition

We provide a weakest precondition calculus over the language set out just previously.

**True & False**

\[ \forall P : P \wp \text{true} \]
\[ \forall P : \neg(P \wp \text{false}) \]

The weakest precondition operator constitutes a partial order over programs. The resultant lattice has a well defined top and bottom, namely the constant boolean predicates `true` and `false`. We explicitly introduce these, and axiomatise the facts that all possible program states satisfy `true`, and no possible program state satisfies `false`.

These two postconditions have weakest preconditions completely independent of their program, and are very useful in proving the weakest precondition of other program and postcondition pairs, by reduction. Dijkstra [5] only includes the latter axiom, but the former is also extremely useful, because it allows us to conclude that if we can reduce a postcondition to truth independent of the program, then any precondition is valid.

**Assignment**

\[ x := e \wp P \equiv P[x/e] \]

After the assignment of `e` to `x`, their values are identical. Logically, the weakest precondition of the program `x` is assigned `e` with the postcondition `P` is that the postcondition must still hold when all occurrences of `x` have been replaced with `e`, as is suggested by the assignment.

The notation used here for the substitution, `P[x/e]`, as previously mentioned represents the name capture avoiding substitution of `e` for `x` in the predicate `P`.

More specifically, it is the substitution of `e` for free variables, `x`, while avoiding any name collisions with existing variables in `P`. The tool performs alpha-renaming as appropriate to ensure that the substitution does not affect the result except by the laws of the semantics.
Skip

\[ \text{skip wp } P \equiv P \]

\textit{skip} represents the program that does nothing but terminate successfully. FITtingly, the weakest precondition of \textit{skip} with respect to some postcondition \( P \) is simply \( P \) (\textit{skip} does nothing; for \( P \) to hold after \textit{skip}, it must hold before).

Sequence

\[ P_1; P_2 \text{ wp } P \equiv P_1 \text{ wp } (P_2 \text{ wp } P) \]

The weakest precondition of a sequence of two predicates is a nesting construct. Intuitively, if the total postcondition \( P \) is to be established by the program, then it must at the very least be established by the final statement of the program; hence \((P_2 \text{ wp } P)\). However, by definition, the only way \( P_2 \) will be able to establish \( P \) is if \((P_2 \text{ wp } P)\) is true. That statement, therefore, must be the postcondition of the previous item in the sequence. Since we are only considering a binary sequential combinator, that item is \( P_1 \), hence \((P_1 \text{ wp } (P_2 \text{ wp } P))\).

Condition

\[(P \preceq b \succeq Q) \text{ wp } R \equiv (b \Rightarrow P \text{ wp } R) \land (\neg b \Rightarrow Q \text{ wp } R)\]

The rule for calculating the weakest precondition of a conditional statement closely mirrors the semantics of the statement itself. We simply swap the conjunctions for implications, saying that the truth of \( b \) implies the weakest precondition of the true branch, and the falsehood of \( b \) implies the weakest precondition of the false branch.

While

\[ b \ast P \text{ wp } Q \equiv I \]

\[ \land (I \land b \Rightarrow P \text{ wp } I) \]

\[ \land (I \land \neg b \Rightarrow Q). \]
The weakest precondition of the repetition construct is more complex than for the previous constructs, because the behaviour of the construct needs characterisation in two new ways, with a loop variant and a loop invariant. In words, the definition above reads:

The weakest precondition of “while \( b \) do \( P \)” is that:

A) The loop invariant holds

B) The loop invariant and the loop condition imply the weakest precondition of the loop body and the loop invariant

C) The loop invariant and the termination of the loop imply the total loop post-condition

where the termination of the loop is denoted by the falsehood of the loop condition, \( b \).

The \texttt{while} construct is usually parameterized with a \texttt{loop invariant}, here denoted \( I \). This is because it is much easier to prove properties of loops given some fact that can be depended upon to be true on entry to the loop, on exit from the loop, and at all points during the execution of the loop.

\textbf{Definition} (Loop Invariant)

The loop invariant is exactly that property of the loop which, in conjunction with the loop condition, \( b \), implies the weakest precondition of the body of the loop, \( P \), and itself \((P \text{ wp } I)\). \( I \) is the postcondition of the loop body, but also implies its weakest precondition, so necessarily, \( I \) cannot be changed by the loop body. Therefore, \( I \) is \texttt{invariant} in the loop.

The loop invariant is usually carefully chosen to minimize the amount of work required to establish the fact to be proven about the loop. Of course, the loop invariant itself must be shown to hold for the proof depending upon it to be valid.

It’s not enough to simply provide an invariant for the \texttt{while} loop, however. Because we provide a weakest precondition semantics, we are concerned with total correctness, i.e. proof of termination as well as correct operation. To prove termination, a common approach is to characterise the execution of one iteration of the loop in terms of a \texttt{loop variant}. 

35
We can then prove that the loop terminates by showing that the loop variant is decreased by each iteration of the loop, reaching a zero value at which the loop condition becomes false. The loop variant must form a well-founded (i.e. having a minimum) relation on the state space of the program.

We can reformulate the weakest precondition of while in terms of a variant function $f$ as:

\[
 b * P_{wp} Q \equiv I
\]
\[
 \land \forall \bullet i : (I \land b \Rightarrow P_{wp} (I \land f < i))
\]
\[
 \land (I \land \neg b \Rightarrow Q).
\]

The variant relation we denote as $<$. The modifications to the previous formulation of weakest preconditions of while are that we explicitly quantify over all initial values, $i$, of the variant function.

To the statement that the loop invariant is preserved by the execution of the loop $(I \land b \Rightarrow P_{wp} I)$ we add that that the value of the variant function, $f$, must have decreased under the relation $<$ after execution of the loop body from the initial value, $(f < i)$.

Importantly, because the variant relation is well-founded, we have that the value of the function will eventually reach a minimum. An in-depth discussion of the properties of well-founded relations of this type can be found in chapter 3 of [12].

The abstract of [10] presents succinctly three different schemes for proof systems involving weakest-precondition style verification. Because of how the calculus transforms while loops, loop invariants must either be provided by the user through some form of annotation of the program, or be inferred somehow by an additional invariant generator.

The problem of inferring loop invariants is, however, formally undecidable in even simple cases, as shown in [4]. Therefore, the approach taken here is to generate placeholders for loop invariants, and have the user of the system perform the annotation required to complete verification.
4.7 Verification

Now that we have a theoretical programming language and a way to express the correctness of programs written with it, we can use the extension to prove properties of simple programs. This section contains a brief walkthrough of the use of the tool to prove a fact about a simple example program.

4.7.1 An Example Program

We consider a simple program consisting of state update after some piece of arithmetic has been computed.

\[
\begin{align*}
  x &:= 3 \\
  y &:= x + y.
\end{align*}
\]

We enter this program into the editor and finalise it, giving it the postcondition \( y = 6 \). This necessitates that \( y \) has the initial value 3.

![Figure 4.4: Generated objects](image-url)
4.7.2 Results from the VCG

Running the verification condition generator gives us a theory with this program inside it, conveniently carved up into pieces which we can manipulate in a proof. The program itself is split apart, with each source statement represented as a separate predicate, the postcondition is represented explicitly as a named predicate, and two weakest precondition statements about the program are generated.

An automated theorem proving system would utilise a constraint system similar to that of Lundblad[10] to reduce these conditions to truth, however the user of the system in this case must do it manually - Saoithin is a proof assistant, not a fully automated proof system. It is possible to use the matching capabilities of the tool to reduce these conditions to friendlier and more readable ones, but the problem of deciding which of multiple valid matches for an expression will enable better progress towards the proof goal is well outside the scope of the tool, for the moment - doing so is part of the actual proof process.

4.7.3 Discharging Proof Obligations

We can discharge the proof obligations that have been generated in the usual fashion any logical system is manipulated with the proof assistant.

![Conjecture - Correctness](image)

Figure 4.5: Conjecture - Correctness
The process beyond this point is interacting with the proof assistant using core functionality that was not modified by this investigation. We would like to prove that the program will run correctly, and terminate with the result \( y = 6 \) in the case that the initial state is \( y = 3 \).

We write this as a simple conjecture, saying that the initial state implies the weakest precondition of the program and the total postcondition. This statement has already been generated for us by the verification condition generator, so we simply write that \(( y = 3 ) \Rightarrow wP P2\) in this case.

Finally, after the verification condition generator has done its job, the user can begin the proof, reducing the conjecture to truth according to the semantics that they have previously supplied for the weakest precondition.
Chapter 5

Conclusions

5.1 Learning Outcomes

5.1.1 Development

Language

This project built on the class on the Haskell programming language taken in the first semester of the year. As part of this project, my familiarity with the language and its libraries was significantly increased, particularly in investigation of the existing project code. I’ve gained a more in-depth understanding of the language and platform as a result of work on this project.

Tools

The wxHaskell graphical user interface library was used in this project. It was also covered in the Haskell programming module, but again, a much more in-depth understanding of the library was gained as a result of investigating the existing code and the library facilities required to build the program entry and editing interface.
5.1.2 Theory

Formal Methods

I gained a significant understanding of the scope and foundations of formal methods as part of this project, building on optional modules taken as part of the degree course. In particular, the differences between major schools of theory and a better understanding of the application and usefulness of that theory are clearer to me as a result of the background reading and study necessary to undertake the investigation.

5.2 Difficulties Encountered

5.2.1 Architecture and Automation

There were problems integrating the verification condition generation process into the tool’s current architecture. Essentially, the tool is constructed as a proof aid, and most manipulations of theoretical objects are phrased in terms of proof. The process of verification condition generation proceeds similarly to proof, but is not dependent on user input, while user input is a fundamental component of the proof process, and drives progress from one step to the next. Verification condition generation is a step on the path to autonomous proof, but the tool is designed to assist in proof bookkeeping rather than undertake proof by itself.

The principal difficulty here was that the code which manipulates theoretical objects relies on user input to select between high-level alternatives, for example, which matching law of a given set to apply to transform a particular expression. Constructing new theoretical objects out of old, an integral part of the process of generating verification conditions, is something this investigation attempts to automate. Generated verification conditions are rather verbose and could benefit from some automated simplification - currently this simplification has to be undertaken by the tool user after the generation process has completed.

Solution

A good solution to the problem would involve a fairly extensive refactor of the tool, pulling out all the manipulation of theoretical objects into a generalised form - essentially an API for predicate logic. On top of such a base, both user-
driven and automated proof processes could be implemented. There is already a significant quantity of code that would be useful, requiring only some abstraction from user-driven proof.

5.2.2 Documentation

The tool has quite extensive documentation in terms of source comments and integrated Literate Haskell documentation snippets, however, the documentation is concerned mainly with the specifics of the code in question. It was quite difficult to initially navigate through the codebase and find the relevant source for the numerous functions of the tool. In particular, a large portion of the glue code connecting the extension’s GUI to the rest of the tool relied on using the in-built parsing library to process programs into a collection of theoretical objects. The tool’s parser is a library developed in-house, built on top of Haskell’s Parsec parser combinators, with custom additions.

Solution

More documentation of the project architecture in the abstract, as opposed to the particulars of individual pieces of functionality would significantly ease the learning curve for developers of new extensions to the tool.
5.3 **Critical Summary of Progress**

5.3.1 **Interface**

The interface is functionally complete, allowing the user to load and save programs, edit the source code of programs, perform test parsing and debugging functions. It could certainly benefit from some cleanup however, and an analysis of user workflow would be a good first step towards that goal.

5.3.2 **Theory Support**

A theory of programming and a weakest precondition semantics for it were developed and used as part of this investigation. In principle, any theory of programming and a suitable theory of correctness can be used to perform verification of programs. The theories developed for this investigation have served their purpose of providing example material well.

5.3.3 **Verification**

The verification support developed in the investigation is not terribly advanced, and could benefit from numerous additions and extensions in functionality, as laid out in the **Future Work** section, to follow. However, it has served its initial purpose of demonstrating the process of verification condition generation.
5.4 Future Work

5.4.1 Interface

The program entry interface could benefit from an analysis of the user workflow during the process of verifying code, as mentioned previously. Numerous tweaks and sophistications would make things easier for the user, such as integrating a facility to highlight erroneous syntax, building on the parser integrated into the tool, or providing a more sophisticated visual representation of the program’s source code.

The interface as it stands is functionally complete, in that it allows the user to perform all of the actions they require for verification, as laid out in the project goals section. Not much effort has been spared for making it look good as well, due to the time constraints involved.

5.4.2 Program Logic

Observation

It would be useful to extract the free variables of each statement in the program and add them to the theoretical representation as observational variables. This would allow to, among other things, inform the user if a program references uninitialised variables, but also would allow programming languages without explicit input/output primitives to record the results of operations by the setting of values into these observation variables. The setting of observational variables to values resulting from computation is a low cost solution to the problem of reporting results of computation from functionally pure languages - what happens after the program has terminated is outside of the control of the programming language.

5.4.3 Verification

The verification condition generator could be extended in several ways to provide more functionality, and would indeed already have been, were it not for the time constraints imposed on this project.

To avoid the need to re-parse the source code of programs if errors are discov-
ered, it would be useful to modify the verification condition generation process to allow it to be invoked, not only on program source code, but on any collection of already-parsed predicates in the system. The user could then remedy errors by changing the erroneous predicates and having the verification conditions re-generated automatically, as opposed to correcting the source and re-running the whole process.
Appendix A

Source Code

The source code for this project can be found on the attached CD-ROM. The principal additions are in the source files “SaoithinProg.lhs”, “Verification.lhs” and “Program.lhs” in the “src” directory of the project. A makefile is included which will build the project and place an executable in the “bin” folder, if the dependencies on versions of the Haskell platform and libraries are met. For more information on building the tool, see the project page[2].
Bibliography


