Text

Description automatically generated

Formatting and Checking Weakest Precondition Proofs in Dafny

Project Proposal

by

**Jacob Freeman**

School of Information Technology and Electrical Engineering, University of Queensland

Submitted for the degree of Bachelor of Software Engineering (Hons)

Under the supervision of Graeme Smith

Submitted: 24/3/2022

Table of Contents

[1 Introduction 1](#_Toc97719381)

[1.1 Topic 1](#_Toc97719382)

[1.2 Purpose 2](#_Toc97719383)

[1.3 Goals 2](#_Toc97719384)

[1.4 Scope 2](#_Toc97719385)

[2 Background 3](#_Toc97719386)

[2.1 Formal Methods 3](#_Toc97719387)

[2.2 Weakest Precondition Proofs and Hoare Logic 4](#_Toc97719388)

[2.2.1 Hoare Triples 5](#_Toc97719389)

[2.2.2 Assignment Axiom 5](#_Toc97719390)

[2.2.3 Precondition Strengthening 5](#_Toc97719391)

[2.2.4 Postcondition Weakening 6](#_Toc97719392)

[2.2.5 Sequencing Rule 7](#_Toc97719393)

[2.2.6 Forwards and Backwards Reasoning 7](#_Toc97719394)

[2.2.7 Conditional Rule 8](#_Toc97719395)

[2.2.8 While Rule 9](#_Toc97719396)

[2.2.9 While Rule for Total Correctness and Termination 10](#_Toc97719397)

[2.2.10 Additional Equivalence Rules 12](#_Toc97719398)

[2.3 Teaching of Formal Methods 13](#_Toc97719399)

[2.3.1 Advantages of Tool Support 13](#_Toc97719400)

[2.3.2 Disadvantages of Tool Support 14](#_Toc97719401)

[2.4 Program Verification Architecture 14](#_Toc97719402)

[2.4.1 Overview 14](#_Toc97719403)

[2.4.2 Verification Conditions 15](#_Toc97719404)

[2.4.3 Suggested Architecture 16](#_Toc97719405)

[2.5 Preliminary Software Architecture Diagrams 17](#_Toc97719406)

[2.5.1 Student Extension 17](#_Toc97719407)

[2.5.2 Educator Extension 18](#_Toc97719408)

[2.6 Prototype User Interface – Student Extension 19](#_Toc97719409)

[2.6.1 Weakest Precondition Input 19](#_Toc97719410)

[2.6.2 Logic or Rule Input 19](#_Toc97719411)

[2.6.3 Syntax Error Indicator 19](#_Toc97719412)

[2.6.4 Syntax Error Hover 19](#_Toc97719413)

[2.7 Prototype User Interface – Educator Extension 20](#_Toc97719414)

[2.7.1 Verification Function Call 20](#_Toc97719415)

[2.7.2 Verification Display 20](#_Toc97719416)

[2.7.3 Verification Method Button 20](#_Toc97719417)

[2.7.4 Verification Method Display 20](#_Toc97719418)

[3 Conclusion 21](#_Toc97719419)

[4 Project Tasks 21](#_Toc97719420)

[4.1 Project Deliverables 21](#_Toc97719421)

[4.2 Student Tool Development 21](#_Toc97719422)

[4.3 Educator Tool Development 22](#_Toc97719423)

[4.4 Extension Educator Tool Development 22](#_Toc97719424)

[5 Project Plan 22](#_Toc97719425)

[6 Risks 22](#_Toc97719426)

[6.1 Time Management 22](#_Toc97719427)

[6.2 Lack of Expertise 23](#_Toc97719428)

[6.3 Worldly Inhibitors 23](#_Toc97719429)

[7 Appendix 24](#_Toc97719430)

[8 Bibliography 24](#_Toc97719431)

List of Figures

[Figure 1: Hoare Logic Notation [2] 4](#_Toc97719432)

[Figure 2: Hoare Triple Notation [2] 5](#_Toc97719433)

[Figure 3: A Hoare Triple 5](#_Toc97719434)

[Figure 4: Hoare Assignment Axiom Notation [2] 5](#_Toc97719435)

[Figure 5: Precondition Strengthening Notation [2] 6](#_Toc97719436)

[Figure 6: Example of Precondition Strengthening 6](#_Toc97719437)

[Figure 7: Postcondition Weakening Notation [2] 6](#_Toc97719438)

[Figure 8: Example of Postcondition Weakening 6](#_Toc97719439)

[Figure 9: Sequencing Rule Notation [2] 7](#_Toc97719440)

[Figure 10: Example of the Sequencing Rule [2] 7](#_Toc97719441)

[Figure 11: Backwards Reasoning [3] 8](#_Toc97719442)

[Figure 12: Conditional Rule Notation [2] 8](#_Toc97719443)

[Figure 13: If/Else Weakest Precondition Proof [3] 9](#_Toc97719444)

[Figure 14: While Rule Notation [2] 10](#_Toc97719445)

[Figure 15: Loop weakest precondition proof 10](#_Toc97719446)

[Figure 16: While Loop for Total Correctness 11](#_Toc97719447)

[Figure 17: Example of Total Correctness 11](#_Toc97719448)

[Figure 18: Variable Declaration Syntax 12](#_Toc97719449)

[Figure 19: Method Assignment Syntax 12](#_Toc97719450)

[Figure 20: Example of Method Assignment [4] 13](#_Toc97719451)

[Figure 21: A System to Check Proofs [2] 15](#_Toc97719452)

[Figure 22: Example of Verification Condition 16](#_Toc97719453)

[Figure 23 - Student Extension Diagram 17](#_Toc97719454)

[Figure 24 - Educator Extension Diagram 18](#_Toc97719455)

[Figure 25 - Student Extension Prototype Interface 19](#_Toc97719456)

[Figure 26 - Protype Syntax Error Hover 20](#_Toc97719457)

[Figure 27 - Educator Extension Prototype Interface 20](#_Toc97719458)

# Introduction

## Topic

Formal verification is slowly growing in industry adoption. As education institutions continue to offer courses related to formal methods and students are continually educated on the importance of formal methods, its popularity will increase further. However formal methods are not the simplest topic to teach and requires intricate mechanisms to be taught to students in a short amount of time. This is made even more difficult by learning the verification techniques by hand. These handwritten proofs can be difficult to format and analyze, not only for students but for educators to verify and mark in an assessment context.

This thesis project delves into the topic of formal methods. More precisely, how can a tool be developed to support students format their proofs and how can educators check these proofs in the Dafny programming language. The proofs investigated are weakest precondition proofs and forms the backbone of many formal verifiers. This proposal introduces formal methods and examines Hoare logic as this is what weakest precondition proofs are reasoned with. The tool will need to analyze and verify this rigorous mathematical logic. The Dafny programming language is explored and its accompanying constructs and architecture. An analysis into the benefits of tool support in formal method teaching to provide the rationale behind this project. Overall, this project makes the connection between formal methods and tool supports in teaching.

Ultimately, students and educators will use the tool under different circumstances. Students will need to use the tool when completing tutorial exercises and assignments, the tool will provide them with assistance for formatting their proofs checking that their proofs are syntactically and semantically correct, strategically neglecting to check the correctness of these proofs. Furthermore, educators will need to verify the student’s proofs for correctness to save time when marking assessments. This tool may also be utilized in the teaching environment e.g. when an example of a proof is shown in class. These different use cases will need to be separated from one another so that students cannot access the tools capabilities available only to educators.

## Purpose

The purpose of this project is to develop a tool or multiple tools to support in the teaching of formal methods using the Dafny programming language, meeting both students and educators needs and requirements.

## Goals

The goal is to develop tools that provide the following functionality:

* Provide context and motivations as to why this tool is needed
* Provide the relevant background knowledge, for technical understanding
* Implement a user-friendly format for students to enter their weakest precondition logic into VSCode
* The syntax and semantics of the weakest precondition logic will be checked for errors and provide suggestions to the user in VSCode
* Verify the correctness of the weakest precondition proof entered
* Provide output as to what weakest precondition logic is correct and why it may not be correct in VSCode
* In case of verification errors, allow educators to see how the weakest precondition proofs were verified so that manual marking is still possible.

Extension:

* Implement a way to mark weakest precondition proofs entered in an online format such as an online quiz

## Scope

The scope of this project is to provide a tool support for writing and verifying weakest precondition proofs to students and educators. These proofs are written in Dafny and no other programming language. Dafny is taught using the development environment bundled in VSCode Studio, the tool must be an extension of this environment, so that students and educators can easily integrate the new tool into the existing environment. The use of third-party verifiers may need to be implemented into this extension architecture. The weakest precondition proof entails the following formal method topics:

* Assignment Axiom
* Precondition Strengthening
* Sequencing Rule
* Conditional Rule
* While Rule
* Termination

Students should not be able to access the educators tool abilities, so distribution needs to be limited and maintained. The scope, of this project is anything that can be included in a VSCode extension to assist in the tool’s creation. An Extension task is to provide marking to students answers on online quiz’s which have the same formal method topics back different software pathways into Dafny for verification, which is yet to be determined.

# Background

## Formal Methods

Since the birth of computer programs, and all the development that has gone into these increasingly complex programs, why aren’t these programs error free. Most programs are affected by parameters that are hard to define and control. In an IT sense, the development process brings faults and bugs into the program. Formal methods allows programmers to reduce the amount of these faults. Formal methods are “mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems”, simply formal methods are used to verifying that programs do what they are supposed to do. This means that program specifications of statements use mathematical logic and formal verification consists of arduous deductions in this logic. The power of formal methods is that it can deduce verification over the entire state of the program and that these deductions hold for all possible inputs. However formal methods cannot be used throughout the entirety of the development process due to the complexity of the code or lack of tool support. Formal methods are typically reserved for high-level design on safety and security critical mechanisms.

When a program is developed, the first step is to generate a specification that describes the programs desired behavior and should be correct and unambiguous. This specification is then translated into code by the programmer. This is when human error gets introduced into the program as misinterpretation of the specification can occur. Then there is also the mass size of some programs, which are difficult to specify. Typically, the program is tested during and after its implementation so that no errors or faults are present in the program. Testing large programs can be very time-consuming resulting in it being unfeasible. In terms of critical systems, the correct functionality needs to be guaranteed, which needs mass testing or a way to prove the program matches the specification. Formal specifications leave no room for misinterpretation. This is where formal methods are introduced to guarantee the correctness of a program, given the formal method used doesn’t contain faults of its own or that the specification is incorrect. In summary, formal methods are used to detect errors more frequently and earlier as it is done at computer speeds compared to human speeds. [1]

## Weakest Precondition Proofs and Hoare Logic

A formal proof of a program is a combination of hypotheses statements culminating with the conclusion statement, such that each statement is an example of an axiom or continues from a previous statement by a rule of logical inference. If denotes the entire program of statements, then denotes that S can be proven, also known as a theorem. Simply put if is a mathematical expression without formal justification. To prove such assertions certain methods of how to do so must be known. The axioms of Hoare logic are specified in by schemas detailed below and logical inference rules can be specified with the notation in Figure 1.

Figure 1: Hoare Logic Notation [2]

This demonstrates that the conclusion can be proved from which are hypotheses. [2]

Simply weakest preconditions are logically obtained, to prove that the precondition given for the program is a subset of the weakest precondition, hence verifying the program for correctness.

### Hoare Triples

Weakest precondition proofs are assembled on top of Hoare triple logic. Simply, a Hoare triple is written as:

Figure 2: Hoare Triple Notation [2]

Where is a precondition, is a statement and is a postcondition. A Hoare triple is valid when S is executable when is true and can also be proven as true afterwards. Formally, Hoare triples follow the syntax convention of using curly brackets around assertions. As a whole, weakest precondition proofs chain together multiple, more complex Hoare triples or assertions to ascertain the weakest precondition in larger bodies of code. [3]

### Assignment Axiom

The assignment axiom shows that the variable after executing the assignment , equals the expression . Witness that if statement is true after the assignment then the statement generated by replacing for in must be true before execution of the statement. Hence, is defined as substituting all occurrences of with in . This logical axiom is the building block for backwards reasoning and is denoted as:

Figure 4: Hoare Assignment Axiom Notation [2]

### Precondition Strengthening

Hoare logic allows preconditions to be strengthened following backward reasoning. This is denoted as:

Figure 5: Precondition Strengthening Notation [2]

The precondition specification for is , so becomes . As backwards reasoning the typically used to verify programs, precondition strengthening is commonly used.

### Postcondition Weakening

Opposite to precondition strengthening is postcondition weakening, where Hoare logic allows postconditions to be weakened following forward reasoning.

Figure 7: Postcondition Weakening Notation [2]

Precondition strengthening and precondition weakening are often called the rules of consequence. [2]

### Sequencing Rule

The assignment axiom is the foundation for weakest precondition proofs but can only be used to evaluate individual statements. As shown in the example below, statements and can be sequenced together because has postcondition , and has precondition , resulting in a formal sequence. Intuitively, larger programs can be analyzed using the assignment axiom by sequencing together multiple statements and the pre and postconditions accompanying these statements, making them Hoare triples.

Figure 9: Sequencing Rule Notation [2]

### Forwards and Backwards Reasoning

Both forward and backward reasoning can be used to reason a programs correctness. Forward reasoning follows each line in the program in the order it would be executed. The disadvantage of this is that assertions will accumulate everything that is known about the program, limiting any proofs as these logical assertions grow larger. This occurs because it is unknown what is being proven. Typically, programmers have a good idea of what needs to be true after a program executes, i.e. the postconditions of the program. Therefore, if the postcondition is known, this postcondition can be proven given an accurate precondition.

Backwards reasoning is often more beneficial for proving programs correctness. Backwards reasoning is the opposite of forward reasoning, wherein a postcondition is given with assertions being made backwards through the statement until the beginning of the program. This guarantees that if the precondition is fulfilled before the execution of the program than the postcondition must be valid. An example of backwards reasoning is:

What if the actual precondition is , which would also prove the postcondition, however not as useful as it is more restrictive. Typically, the precondition that allows correctness for the largest set of inputs, is regarded as the weakest precondition. Stated differently, the weakest precondition represents the most general precondition needed to prove the postcondition, with a stronger precondition representing a smaller subset of precondition assertion. The weakest precondition function can be written as . [3]

### Conditional Rule

The examples shown until now have only involved linearly sequenced statements. Although, conditional statements such as if/else have multiple possible branches adding complexity to the Hoare triple. The conditional rule follows the following notation:

Figure 12: Conditional Rule Notation [2]

An example of this can be seen below, further backwards reasoning can occur once the state of the above notation is reached:

### While Rule

When analyzing loops in weakest precondition logic, while loops are used instead of the more widely used for loop, due to their simplicity in verification. While loops separate the initial statements, loop guard and step statement, used in for loops. Loops are more complex to prove as it is unknown how many times the loop will be executed or when the loop terminates. Therefore, the postcondition can be satisfied with the use of a loop invariant, denoted . A loop invariant can be considered as a precondition and a postcondition for the loop as it needs to hold immediately before and after the loop, as well as in every point during the loop’s execution. Ultimately when the while loop is exited the loop guard, denoted won’t hold. When writing a loop invariant, the programmer typically gets inspired by either the postcondition or the loop guard. The while rule follows the following notation:

Figure 14: While Rule Notation [2]

### While Rule for Total Correctness and Termination

The correctness of a program is defined by whether it terminates. If a program does not terminate than unexpected results can happen, and errors can be thrown. Using the Hoare triple notation of , if program is started in any state then is satisfied and will terminate to a state satisfying . This is known as total correctness, when the program always terminates to the correct results. Partial correctness of a program occurs when the program terminates with the right result but terminate isn’t precisely known. Hence a termination proof is need for total correctness.

Most loops typically terminate. So, when proving a loops correctness, a proof involving the termination metric needs to be computed. In this notation, is known as the loop variant and changes as the loop iterates. strictly decreases as after on a domain set , typically dependent on the loop guard . As is finite and is strictly decreasing, t cannot decrease forever as it is restricted by . In Hoare logic the specifications only hold if a loop terminates. Termination metrics can be proven using the following notation:

Figure 16: While Loop for Total Correctness

Termination proofs are typically grouped in with the weakest precondition proof using ghost variables. Ghost variables can be assigned with the body of the program without being included in the code’s compilation.

The dummy variable is assigned to at the base of the while loop, as is the loop variant. represents the termination metric that while be assigned at the beginning of the while loop. As the definition of is changed in the loop body, then can assign the loop variant to the specification in a form that strictly decreases so that the termination predicate in the specification can be satisfied, hence when satisfies proving termination.

### Additional Equivalence Rules

There are a wide range of equivalence rules that can be used in the formal verification. However, the rules highlighted here add extra syntax and semantics to the proof.

#### Variable Declarations

When a variable is declared in the program, all instances of that variable are assigned in the precondition. This is denoted by the syntax:

Figure 18: Variable Declaration Syntax

#### Method Assignments in Recursion

If a method is assigned to a variable, in the case of recursion, then both the pre- and postconditions of the method are used in the adjoining precondition in the program. This assignment follows the following notation, where denotes the methods precondition, R denotes the methods postcondition, is the return value of the method and denotes postcondition of the program:

Figure 19: Method Assignment Syntax

## Teaching of Formal Methods

### Advantages of Tool Support

In the context of using formal methods in software engineering, tool support is critical for increasing productivity and reducing mistakes in development. Often students are afforded the chance to use tool support when completing small projects. However, this is usually allowed after students have had to learn formal methods by hand, which is typically difficult at first. This leads to students enjoying the high automation that the formal method tool support provides.

As formal methods consist of complex mathematical concepts, students tend to take extra time to solidify their knowledge, slowing down the teaching process. However, many formal method courses offered at tertiary educational institutions are like a software project in the context that there are time constraints. This limits the extensiveness of the teaching content, restricting the course of more complex formal method topics. Hence courses use tool support to decrease the steep learning curve many students have when learning formal methods, but only after they have learnt formal methods from hand. By exposing students to tool supports for formal methods, and helping facilitate the tools development will lead to a higher adaption of formal methods in industry. [8]

Formalization and its accompanying tool supports are essential for software analysis. It is very popular to analysis software in more informal ways by using diagrams. As this is generally accepted in the programming community, providing rigid formal specifications of the programs intended functionality leads to more accurate programs. Formal specifications of even the smallest programs may be wrong, without tool supports as hand-written verification attempts are prone to human errors. Therefore, verification proofs regarding the correctness of programs must me mechanized. Even a lightweight formalization tool can save time, for example decreasing the amount of test cases. [9]

### Disadvantages of Tool Support

Many students have experience with tool supports when learning programming languages such as C and Java. These tool supports effectually assist students write, execute, and test their programs. Formal methods teachers tend to believe that tool support will also assist students learn formal methods. However, the use of tool supports may not be as effective as first thought, as specified by some educators of VDM and SOFL courses, that delve into formal specification techniques.

Learning formal methods, and the specification language used, involves students to learn the syntax and semantics of this language. It has been sustained that the best way to learn the dynamics of a specification language is to write out the formal specifications by hand, as one would learn English as a second language. This teaching strategy is typically effective as exercises and projects given to students are small and increases student’s memory and depth of knowledge. By not having tools support, removes the student’s ability to ‘copy and paste‘ specifications, without thinking and analyzing the specifications for themselves. The purpose of a formal method specification is not for a computer to directly do this analysis, but for the reader to understand the specification written. Therefore writing by hand forces students to do this analysis in their head without the use of a tool support to help format their specifications. As specification languages don’t need to be compiled and run, not having a tool support won’t cause any major inconvenience. [8]

## Program Verification Architecture

### Overview

A weakest precondition proof can be formalized using a goal orientated method. The verification system can be regarded as a proof checker that also generates proofs. A simple example of this system can be seen below:

Diagram

Description automatically generated

Figure 21: A System to Check Proofs [2]

The system inputs a correctness specification statements that describe the relationships between variables. From these specifications the system generates a set of mathematical statements, known as verification conditions or VC’s. If the verification conditions are provable, the original specification can be deduced from the Hoare logic. The verification conditions are sent to a theorem prover which attempts to automatically prove verification conditions, and if verification can’t occur, advice is sent to the user.

### Verification Conditions

To prove , three things must be done:

1. The program is annotated. This is done by injecting statements that are meant to hold at intermediary points
2. A group of verification conditions are then generated from the annotated specification
3. The verification conditions are proven

As verification conditions are just mathematical statements, step 2 can be seen as ‘compiling’ a verification problem into a mathematical problem.

### Suggested Architecture

#### Step 0: User Input of Weakest Precondition Logic

The user will have the ability to enter their weakest precondition logic into the program. This can be done at any point in the program.

#### Step 1: Annotations

An annotation is a specification made to define the program. These are only made before and after each statement in the program making up a pre- and postcondition for the statement or as an invariant for a while loop. Ultimately, the annotations made should hold when the program reaches that control point. A correctly annotated program specifies . The correct annotations and a multitude of examples are shown in 2.2.

#### Step 2: Generate Verification Conditions

The step provides a procedure that generates verification conditions for . A simple verification condition example is shown below:

This produces the verification condition:

Which is true.

Figure 22: Example of Verification Condition

More complex verifications can be produced following the rules shown in 2.2.

#### Step 3: Prove Verification Conditions

An annotated specification can be proved using Hoare logic, given that the verification conditions are provable. The indicates that the verification conditions are sufficient, but not exactly necessary. The verification conditions are in their weakest form. These verifications can be strengthened and still be accurate. The proof that the verification conditions are sufficient will be done by inductive reasoning on . Induction has 2 parts, the first is the basis induction which shows that the results hold for assignments. The second is the step induction which shows that holds for non-assignments and if the result holds for constituent statements. [2]

#### Step 4: Comparing Users Weakest Precondition Logic to the Verification Conditions

The purpose of this project is to allow users to enter their weakest precondition proof logic into Dafny to be formatted and verified. Their logic input should resemble the verification conditions generated in step 2. Verification can still occur if the logic represents a subset of the appropriate verification condition. Advice and suggestions will then be outputted back to the user, on how to improve their logic.

## Preliminary Software Architecture Diagrams

### Student Extension

The student extension will follow the following software architecture:

Diagram

Description automatically generated

Figure 23 - Student Extension Diagram

The text entered by the user for the weakest precondition proof will be sent in String format for syntax analysis, which will require a rigorous algorithm. The errors determined by this analysis will then be send to the user interface for visual display in the form of squiggly lines and pop-up boxes.

### Educator Extension

The educator extension will follow the following software architecture:

Diagram

Description automatically generated

Figure 24 - Educator Extension Diagram

When the user calls the verification function (shortcut F10), the text of all the weakest precondition proofs will be sent for syntax analysis, similar to the student extension. Depending on the outcome of this analysis, a negative verification status may be sent to the user interface. Otherwise, if syntax standards are met, the weakest precondition proofs are sent for verification condition analysis, which will be a complex algorithm explained in 2.4. This will return either a positive or negative verification status to be displayed on the user interface. The educator will then be presented with a button for verification method viewing, which will return the analysis done to determine that verification condition, so that the educator can manually resolve any misinterpretations of the analysis, if these were to occur.

## Prototype User Interface – Student Extension

The following prototype user interface provides an example for what the student extension can potentially do.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 25 - Student Extension Prototype Interface

The important components of the extension are explained below:

### Weakest Precondition Input

The weakest precondition logic can be added into a new line anywhere within the Dafny program text without affecting the program. This is done by using comments to hide the added logic text from the Dafny text, in the format of ‘//{ logic }’. This format also maintains the syntax highlighting that Dafny has across the weakest precondition logic, such as highlighting functions yellow. From here, syntax errors can be sent to the weakest precondition logic separate from the Dafny program.

### Logic or Rule Input

Each weakest precondition logic input will be manipulated by some form of rule the simplify the logic. The names of these rules can be added in comment format after the weakest precondition proof for tutors to mark.

### Syntax Error Indicator

The syntax indicator seen in the interface above is a yellow squiggly line, the same as an error in the Dafny language. This indicator will highlight exactly the area of text that has the error. There is the potential for different types of errors to be displayed by different colors.

### Syntax Error Hover

VSCode enables the user to hover over the syntax error indicator to interpret exactly what the error is, this can be seen in the image above. These can be customized to whatever the specific error is enabling the user a precise debugging tool.

Graphical user interface

Description automatically generated with medium confidence

Figure 26 - Protype Syntax Error Hover

## Prototype User Interface – Educator Extension

The following prototype user interface provides an example for what the educator extension can potentially do.

Text

Description automatically generated

Figure 27 - Educator Extension Prototype Interface

The important components of the extension are explained below:

### Verification Function Call

Once the student’s logic is obtained, the educator can call the verification function by simply pressing F9 on the keyboard. This will trigger an event where all the weakest precondition logics will be analyzed, and a verification status will be determined.

### Verification Display

This will display the verification status of all the weakest precondition logic in a colored box after the weakest precondition input. It will either display verified or unverified, with the colors triggering an immediate indication of right and wrong to the educator.

### Verification Method Button

If the verification status is unusual or cannot be determined, then the educator has the ability to press a verification method button positioned after each verification condition displayed, this is not shown in the prototype image above.

### Verification Method Display

Once the verification method button is pressed a hover will appear with the Dafny logic that was used to analyze the weakest precondition proof for errors and anomalies.

# Conclusion

Many formal verification tools are built on top of Hoare logic to reason about a program. To verify these programs a set of verification conditions are generated and verified for correctness. As educators teach students these formal methods using weakest precondition reason, there teaching methods are done by hand. When their teachings are translated to working with programming language and IDE’s students struggle to reason their programs, amongst complex verification architecture and logic hidden ‘under the hood’ in many programs source code. Currently there are many tools that verify programs for users but no tools that verify the users weakest precondition reasoning often used in education. Therefore, the development of this tool will provide support to the teaching of formal methods, specifically weakest precondition proofs.

# Project Tasks

The project can be split into distinct parts, the deliverables the tool support available for the student and the tool support available for the educator. These distinct parts can be split up further into smaller tasks.

## Project Deliverables

The project has deliverables that are due by certain deadlines, these deadlines are provided below:

* Thesis Project Proposal (TBA)
* Thesis Seminar Presentation (TBA)
* Thesis Final Presentation + Poster (TBA)
* Thesis Document (TBA)

## Student Tool Development

The student tool support has the following tasks:

* Implement a user-friendly input for students to enter their weakest preconditions
* The syntax and semantics of the weakest precondition logic will be checked for errors
* Provide suggestions for fixing syntax and semantic errors to the user
* Provide suggestions for weakest precondition proof formatting to the user

## Educator Tool Development

The educator tool support has the following tasks:

* Verify the correctness of the weakest precondition proof entered in VSCode
* Provide output as to why the weakest precondition logic is correct and why it may not be correct

## Extension Educator Tool Development

* Verify the correctness of the weakest precondition proof entered in an online quiz format
* Provide output as to why the weakest precondition logic is correct and why it may not be correct

# Project Plan

* Thesis Project Proposal (S1W5)
* Thesis Seminar Presentation (S1W11)
* Student Tool (S1W13)
* Educator Tool (S2W6)
* Extended Educator Tool (S2W12)
* Thesis Final Presentation + Poster (S2W12)
* Thesis Document (S2W13)

# Risks

This project will be presented with several risks, however as this project has no practical/physical component the risks are minimal. The risks are outlined below:

## Time Management

Due to the unknown aspects of the project, even after extensive research is conducted the time need to complete the project to a sufficient standard is hard to predict. As this project has several deliverables with deadlines, these deadlines will need to be met with no barriers to their completion. This can be done by using extensive time management, a plan for time management is displayed in section 5.

## Lack of Expertise

This project will be conducted under the supervision of Dr Graeme Smith with credentials: Bachelor of Engineering (Hons) and Doctor of Philosophy. His fellow contacts will also be utilized. As Dr Smith is an expert in the field of formal methods with industry and educational experience, the projects level of sophistication needed can be met.

## Worldly Inhibitors

The February 2022 flood in Brisbane where teaching at UQ was cancelled for a weak could have impacted the timeline of the project, however other formats for communicator were used for meetings with supervisor Graeme Smith. Another COVID-19 outbreak / variants could cause subsequent lockdowns and border closers, however as this has been ongoing for the past 2-3 years this should be nothing that can’t be handled.

# Appendix

# Bibliography

[1] M. K. Ioana Rodhe, "Overview of formal methods

in software engineering," 2015.

[2] M. Gordon, "Background reading on Hoare Logic," 2016.

[3] H. P. Krysta Yousoufian, Michael Ernst, David Notkin, and Dan Grossman. "Reasoning About Code, Part 1." <https://www.cs.williams.edu/~freund/cs326/ReasoningPart1.html> (accessed.

[4] G. Smith, "Tutorial 3 Solutions," ed, 2021.

[5] K. R. M. L. Jason Koenig, "Getting Started with Dafny: A Guide," 2012.

[6] R. L. F. K. Rustan M. Leino, David R. Cok, *Dafny Reference Manual*, 2021.

[7] V. W. K. Rustan M. Leino, "The Dafny Integrated Development Environment," 2014.

[8] K. T. Shaoying Liu, Toshinori Hayashi, Toshihiro Nakayama, "Teaching Formal Methods in the

Context of Software Engineering," 2014.

[9] R. B. Wolfgang Ahrendt, and Reiner H¨ahnle, "Integrated and Tool-Supported Teaching

of Testing, Debugging, and Verification," pp. 125-143, 2009.

[10] R. L. F. K. Rustan M. Leino, David R. Cok, "This is Boogie 2," 2008.

[11] K. R. M. L. Claire Le Goues, Michał Moskal, "The Boogie Verification Debugger (Tool Paper)," 2011.

[12] K. R. M. L. Cormac Flanagan, Mark Lillibridge, Greg Nelson, James B. Saxe, Raymie Stata, "Extended Static Checking for Java," 2002.

[13] A. P. Jean-Christophe Filliâtre, "Why3 — Where Programs Meet Provers," 2013.